Exponential Algorithms for Scheduling Problems
Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, Vincent T’Kindt

To cite this version:
Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, Vincent T’Kindt. Exponential Algorithms for Scheduling Problems. 2014. <hal-00944382>
Exponential Algorithms for Scheduling Problems

Christophe Lenté
Mathieu Liedloff
Ameur Soukhal
Vincent T'kindt

Ecole Polytechnique de l’Université de Tours
Département Informatique
64 Av. Jean Portalis
37200 TOURS - FRANCE
e-mail : tkindt@univ-tours.fr

February 2014
Research Report n°300
Exponential Algorithms for Scheduling Problems

Christophe Lenté · Mathieu Liedloff · Ameur Soukhal · Vincent T’kindt

Received: date / Accepted: date

Abstract This paper focuses on the challenging issue of designing exponential algorithms for scheduling problems. Despite a growing literature dealing with such algorithms for other combinatorial optimization problems, it is still a recent research area in scheduling theory and few results are known. An exponential algorithm solves optimally an \( \mathcal{NP} \)-hard optimization problem with a worst-case time, or space, complexity that can be established and, which is lower than the one of a brute-force search. By the way, an exponential algorithm provides information about the complexity in the worst-case of solving a given \( \mathcal{NP} \)-hard problem.

In this paper, we provide a survey of the few results known on scheduling problems as well as some techniques for deriving exponential algorithms. In a second part, we focus on some basic scheduling problems for which we propose exponential algorithms. For instance, we give for the problem of scheduling \( n \) jobs on 2 identical parallel machines to minimize the weighted number of tardy jobs, an exponential algorithm running in \( O^*(\sqrt[3]{n}) \) time in the worst-case.

Keywords Exponential algorithms · worst-case complexity · scheduling theory

C. Lenté
University François-Rabelais of Tours, Laboratoire d’Informatique (EA 6300), ERL CNRS OC 6305, 64 avenue Jean Portalis, 37200 Tours, France
E-mail: christophe.lente@univ-tours.fr

M. Liedloff
LIFO, Université d’Orléans, 45067 Orléans Cedex 2, France
E-mail: mathieu.liedloff@univ-orleans.fr

A. Soukhal
University François-Rabelais of Tours, Laboratoire d’Informatique (EA 6300), ERL CNRS OC 6305, 64 avenue Jean Portalis, 37200 Tours, France
E-mail: ameur.soukhal@univ-tours.fr

V. T’kindt
University François-Rabelais of Tours, Laboratoire d’Informatique (EA 6300), ERL CNRS OC 6305, 64 avenue Jean Portalis, 37200 Tours, France
E-mail: tkindt@univ-tours.fr
1 Introduction and Issues of Exponential Algorithms

Scheduling consists in determining the optimal allocation of a set of jobs (or tasks) to machines (or resources) over time. Since the mid 50’s, scheduling problems have been the matter of numerous researches which have yield today to a well-defined theory at the crossroad of several research fields like operations research and combinatorial optimization, computer science and industrial engineering. Most of the scheduling problems dealt with in the literature are intractable problems, i.e. $\mathcal{NP}$-hard problems. Consequently, an optimal solution of such problems can only be computed by super polynomial time algorithms (unless $\mathcal{P} = \mathcal{NP}$). Usually, the evaluation of the efficiency of such algorithms is conducted through extensive computational experiments and the challenge is to solve instances of size as high as possible. But, theoretically speaking, several fundamental questions remain open: for exponential-time algorithms can we establish stronger conclusions than their non polynomiality in time? For instance, is it possible to derive upper bounds on their average complexity or their worst-case complexity? This is a task which is usually performed for polynomially solvable problems: when we provide an exact polynomial-time algorithm we usually also provide information about the number of steps it requires to compute an optimal solution. Why not for $\mathcal{NP}$-hard problems and exponential-time algorithms?

The interest in studying the worst-case, or even average, time complexity of such algorithms is beyond the simple interest of counting a number of steps. It is related to establishing properties of $\mathcal{NP}$-hard problems: assume we deal with a $\mathcal{NP}$-hard optimisation problem for which a brute-force search requires $n!$ steps, with $n$ the size of the input, to compute an optimal solution. The question is: can this problem admit an exponential algorithm with a worst-case time complexity lower than that of this enumeration algorithm? Can we solve it using, for instance, $2^n$ steps? Such a property would give an indication on the expected difficulty of a problem, and also challenge the design of efficient optimal algorithms: their efficiency should be still evaluated via computational experiments, but they would have also to not exceed the upper bound on the worst-case complexity established on the problem.

It also has to be noted that fixed-parameter tractable algorithms are strongly related to exponential-time algorithms: the former are capable of solving to optimality $\mathcal{NP}$-hard problems within a time complexity bounded by a function exponential in a parameter $k$ of the instances. Fixed-parameter tractable algorithms are out of the scope of this paper, and the interested reader is kindly referred to Niedermeier [2006], among others.

In this paper, we make use of the notation $O^*$ for worst-case complexities: an exponential algorithm is said to have a $O^*(\alpha^n)$ worst-case complexity iff there exists a polynomial $p$ such that the algorithm is in $O(p(n),\alpha^n)$. The study of exponential-time algorithms solving $\mathcal{NP}$-hard optimisation problems has been the matter of a recently growing scientific interest. The first exponential-time algorithms date back from the sixties and seventies. Most well-known algorithms are Davis-Putnam’s and Davis-Logemann-Loveland’s algorithms for deciding the satisfiability of a given CNF-SAT instance, i.e. a propositional logic formulae being in conjunctive normal form (Davis and Putnam [1960], Davis et al [1962]). Algorithms solving restricted versions of SAT have also attracted a lot of attention, e.g. the best-known randomized algorithm solves 3-SAT in time $O^*(1.3210^n)$ (Hertli et al [2011]). Exponential-time algorithms for $\mathcal{NP}$-hard graph problems have been also established. The Traveling Salesman Problem can be solved trivially in $O^*(n!)$ time by enumerating all possible permutations of the $n$
cities. Based on a dynamic programming approach, Held and Karp gave in 1962 an \(O^*(2^n)\) time algorithm for solving the problem on arbitrary graphs. Then, the problem has been studied for bounded-degree graphs (see e.g. Björklund et al [2008], Iwama and Nakashima [2007]). However, up to 2010, no improvement has been done for arbitrary graphs. An attempt is due to Björklund [2010] who presented a Monte Carlo algorithm deciding the existence of an Hamiltonian circuit in a graph in \(O^*(1.657^n)\) time.

Another well-studied graph problem is called the maximum independent set problem: given a graph \(G = (V, E)\), it asks to compute a maximum-size subset \(S \subseteq V\) such that no two vertices in \(S\) are adjacent. The problem can be solved in \(O^*(2^n)\) by enumerating all possible subset of vertices. Tarjan and Trojanowski [1977] gave an \(O^*(1.259^n)\) time algorithm which has been improved by a sequence of papers. By now, the best known algorithm is due to Bourgeois et al [2011] and has a worst-case running time of \(O^*(1.2114^n)\). To complete this short list of graph problems, we mention the problem of coloring a graph with a minimum number of colors such that adjacent vertices have different colors. Lawler [1976] showed that the problem can be solved in time \(O(2.4423^n)\) and a major improvement has been achieved by Björklund et al [2009]. Thanks to an inclusion-exclusion formula approach, they proposed an \(O^*(2^n)\) time algorithm. Finally, we mention the knapsack problem: Horowitz and Sahni [1974] gave an \(O^*(1.4142^n)\) time algorithm based on an approach called Sort & Search. In the last decade, the design and analysis of exponential-time algorithms saw a growing interest. Several books and surveys are devoted to the subject (Fomin and Kratsch [2010], Woeginger [2003, 2004]).

For problems involving graphs, the relevant size measure is typically a cardinality, such as the number of vertices or edges in the instance. The scheduling problems studied in the present paper are more complicated in the sense that their instances involve cardinalities (the number of jobs to schedule and/or the number of machines) and values (like processing times of jobs). Intuitively, it seems less easy to correlate the worst-case complexity of an exponential-time algorithm only to the size of the instances. In this paper we consider a set of basic scheduling problems which share the following definition. A set of \(n\) jobs has to be scheduled on a set of \(m\) machines. Each job \(i\) is made up of, at most, two ordered operations specified by processing times \(p_{i,1}\) and \(p_{i,2}\). More particularly, we study several configurations:

- **Single machine problems** for which \(m = 1\) and each job \(i\) has one operation of processing time \(p_i\) (the second index is omitted),
- **Parallel machine problems** for which \(m\) is arbitrary and each job \(i\) has one operation of processing time \(p_i\). This operation can be processed by any machine,
- **Interval scheduling problems** for which \(m\) is arbitrary, each job \(i\) has one operation and can be only processed by a given subset of machines. These problems have the particularity that each job \(i\) is only available during a time interval \(I_i = [r_i, \tilde{d}_i]\) with \(p_i = \tilde{d}_i - r_i\),
- **2-machine Flowshop problems** for which 2 machines are available and each job \(i\) has two ordered operations. For each job, the first operation is processed on the first machine before the second operation is processed on the second machine. Besides, without loss of optimality for the considered problems, we assume that the sequence of jobs on the first machine is the same than on the second machine.
The aim of these scheduling problems is to allocate optimally the jobs to the machines in order to minimize a given criterion and, possibly, under additional constraints. Let us define by $C_i(s)$ the completion time of the last operation of job $i$ in a given schedule $s$. Besides, let us refer to $f_i$ as the cost function associated to job $i$ and depending on the value of $C_i(s)$. It can be interesting to minimize two general cost functions $f_{\text{max}}(s) = \max_{1 \leq i \leq n} f_i(C_i(s))$ or $\sum f_i(s) = \sum_{i=1}^n f_i(C_i(s))$. Notice, that from now on the mention of schedule $s$ in the completion time notation will be omitted for simplicity purposes, except when it will be unavoidable in the text.

Particular cases of the maximum cost function $f_{\text{max}}$ are the makespan criterion defined by $C_{\text{max}} = \max_{1 \leq i \leq n}(C_i)$, the maximum tardiness criterion defined by $T_{\text{max}} = \max_{1 \leq i \leq n}(\max(0; C_i - d_i))$ and the maximum lateness criterion defined by $L_{\text{max}} = \max_{1 \leq i \leq n}(C_i - d_i)$. The data $d_i$ is the due date of job $i$. Similarly, particular cases of the total cost function $\sum f_i$ are the total weighted completion time defined by $\sum w_i C_i$, the total weighted tardiness defined by $\sum w_i T_i = \sum w_i \max(0; C_i - d_i)$ and the total weighted number of late jobs defined by $\sum w_i U_i$ with $U_i = 1$ if $C_i > d_i$ and $U_i = 0$ otherwise. The data $w_i$ is the tardiness penalty of job $i$. For the tackled interval scheduling problem the aim is not to minimize one of these criteria but only to decide of its feasibility. The above particular cases of $f_{\text{max}}$ and $\sum f_i$ criteria share the implicit property that the $f_i$’s are non-decreasing functions of the completion times $C_i(s)$. There exists other particular cases for which this property does not hold as for instance the total earliness criterion defined by $\sum E_i = \sum \max(0; d_i - C_i)$.

The scheduling problems dealt with in this paper are referred using the classic 3-field notation $[\alpha|\beta|\gamma]$ introduced by Graham et al. [1979], with $\alpha$ containing the definition of the machine configuration, $\beta$ containing additional constraints or data and $\gamma$ the criterion which is minimized. For instance, the notation $1|d_i| \sum w_i U_i$ refers to the single machine problem where each job is additionally defined by a due date $d_i$ and for which we want to minimize the total weighted number of late jobs $\sum w_i U_i$. The particular interval scheduling problem tackled in this paper will be only referred to as IntSched.

For more information about scheduling theory, the reader is kindly referred to basic books on the field (see Brucker [2007] and Pinedo [2008] among others).

Before synthesing the results that are provided in this paper, we need to introduce an additional property of some scheduling problems.

**Definition 1** A schedule $s$ on a single machine is said to be *decomposable* iff $C_{\text{max}}(s) = \sum_{i \in s} p_i$.

**Definition 2** A schedule $s$ on parallel machines is said to be *decomposable* iff $C_{\text{max}}(s_j) = \sum_{i \in s_j} p_i$, $\forall j = 1, \ldots, m$, with $s_j$ the sub-schedule of $s$ on machine $j$.

The class of decomposable schedules is dominant for several scheduling problems, as for instance the $1|d_i| T_{\text{max}}$ problem. This means that, for such problems, there always exist at least one optimal schedule which answers the decomposability property. Examples of problems for which this is not the case, are scheduling problems with jobs having distinct release dates. When dealing with problems for which we explicitly restrict the search for optimal solutions to decomposable schedules, we mention in the $\beta$-field of the problem notation the word *dec*.

Another important motivation of this paper is related to the novelty of the study: up to now, the establishment of worst-case complexities for $\mathcal{NP}$-hard scheduling problems has been the matter of few studies in the literature. Woeginger [2005] presented...
a pioneer work (also given in the book of Fomin and Kratsch [2010]) on a single machine scheduling problem with precedence constraints, referred to as $1|\text{prec}| \sum w_i C_i$. He gave a dynamic programming algorithm running in $O^*(2^n)$ and suggested that such dynamic programming also enables to derive a $O^*(2^n)$ exponential-time algorithms for the $1|d_i| \sum w_i U_i$ and $1|d_i| \sum T_i$ problems, and a $O^*(3^n)$ exponential-time algorithm for the $1|r_i, \text{prec}| \sum C_i$. Later on Cygan et al [2011] provided, for the $1|\text{prec}| \sum C_i$ problem, an exponential algorithm in $O^*((2-10^{-10})^n)$ time.

Table 1 presents a synthesis of the results proved later on in this paper and the results established by Woeginger [2003] and recently by Fomin and Kratsch [2010]. The first column contains the problem notation for which is indicated in the second column the worst-case complexity of the brute-force search algorithm. The third column shows the worst-case complexities of proposed exponential-time algorithms and the fourth column refers to the publication or section of this paper which contains the proofs of the results.

As the $1|\text{dec}| \sum f_i$ problem generalizes the $1|d_i| \sum w_i T_i$ and $1|d_i| \sum w_i C_i$ problems, they can be solved in $O^*(2^n)$. When turning to the problems with parallel machines the same generalizations can be established.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Enumeration</th>
<th>Exp. Time Alg.</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1</td>
<td>\text{dec}</td>
<td>f_{\text{max}}$</td>
<td>$O^*(n!)$</td>
</tr>
<tr>
<td>$1</td>
<td>\text{dec}</td>
<td>\sum f_i$</td>
<td>$O^*(n!)$</td>
</tr>
<tr>
<td>$1</td>
<td>\text{prec}</td>
<td>\sum C_i$</td>
<td>$O^*(n!)$</td>
</tr>
<tr>
<td>$1</td>
<td>\text{dec}</td>
<td>\sum w_i C_i$</td>
<td>$O^*(n!)$</td>
</tr>
<tr>
<td>$1</td>
<td>d_i</td>
<td>\sum w_i U_i$</td>
<td>$O^*(n!)$</td>
</tr>
<tr>
<td>$1</td>
<td>d_i</td>
<td>\sum T_i$</td>
<td>$O^*(n!)$</td>
</tr>
<tr>
<td>$1</td>
<td>r_i, \text{prec}</td>
<td>\sum w_i C_i$</td>
<td>$O^*(n!)$</td>
</tr>
<tr>
<td>$\text{IntSched}$</td>
<td>$O^*(2^n \log(n))$</td>
<td>$O^*(1.2132^{nm})$</td>
<td>Section 3</td>
</tr>
<tr>
<td>$P</td>
<td>\text{dec}</td>
<td>f_{\text{max}}$</td>
<td>$O^*(m^n n!)$</td>
</tr>
<tr>
<td>$P</td>
<td>\text{dec}</td>
<td>\sum f_i$</td>
<td>$O^*(m^n n!)$</td>
</tr>
<tr>
<td>$P4</td>
<td>\text{dec}</td>
<td>C_{\text{max}}$</td>
<td>$O^*(4^n)$</td>
</tr>
<tr>
<td>$P3</td>
<td>\text{dec}</td>
<td>C_{\text{max}}$</td>
<td>$O^*(3^n)$</td>
</tr>
<tr>
<td>$P2</td>
<td>\text{dec}</td>
<td>C_{\text{max}}$</td>
<td>$O^*(2^n)$</td>
</tr>
<tr>
<td>$P2</td>
<td>d_i</td>
<td>\sum w_i U_i$</td>
<td>$O^*(3^n)$</td>
</tr>
<tr>
<td>$F2</td>
<td>\text{dec}</td>
<td>C_{\text{max}}$</td>
<td>$O^*(2^n)$</td>
</tr>
</tbody>
</table>

Table 1 Synthesis of the best known worst-case complexities

The remainder is organized as follows. Section 2 introduces some of the classic techniques used in the literature to compute worst-case complexities for $\mathcal{NP}$-hard problems. In section 3 we start with the study of a multiskilled interval scheduling problem which is a very particular scheduling problem. In sections 4 and 5 we focus
on basic single machine and parallel machine scheduling problems. Section 6 ends up the study of scheduling problems by focusing on a particular but complex 2-machine flowshop problem. Conclusions and future research lines are next provided.

2 Some Techniques Used to Derive Worst-Case Complexities

The design and analysis of exponential-time algorithms has been recently the subject of a comprehensive monograph (Fomin and Kratsch [2010]). To design exponential-time algorithms, two possibilities are offered to us: find a problem-specific decomposition scheme to break the problem into smaller subproblems, or apply a known general decomposition scheme (technique). For some of the scheduling problems considered in this paper we have proposed exponential-time algorithms based on dedicated decomposition schemes. But we also have successfully applied some known techniques which are mainly Dynamic Programming and Sort & Search.

This section intends to provide the reader with an overview of some classic techniques focusing on the two mostly used in the remainder of the paper. As outlined by Fomin and Kratsch [2010], one common way to derive exponential-time algorithms is to consider branching-based algorithms. A typical example, largely used in the literature, are Branch-and-Bound algorithms which provide optimal solutions with exponential time and, most of the time, polynomial space. But, one of the difficulty induced by such algorithms is to derive a worst-case time complexity better than the brute-force search: this is due, at least, by the bounding mechanism which makes intractable the analysis of their time complexity. A more used technique, called Branch-and-Reduce, has been successfully used to derive exponential-time algorithms. It shares with Branch-and-Bound algorithms the feature of branching to decompose the problem into subproblems. But a Branch-and-Reduce algorithm has no bounding mechanism and does not use dominance conditions. It rather uses a reduction procedure at each node. The underlying idea of such a procedure, for a given node, is to decrease in polynomial time the length of the instance of the subproblem to solve at this node. Consequently, we may be able to analyse that, in the worst case, the size of the search tree is lower than if no reduction procedure was used. Thus, this leads to a decreased worst-case time complexity than that of the brute-force search. An illustration is given in figure 1 in which is pictured the effect of the reduction procedure at a node $\pi$. In this figure $\pi^*$ refers to the “best” node in the subtree $T$ that can be attained from node $\pi$. Besides, node $\pi'$ is on the path from $\pi$ to $\pi^*$ in the search tree. Therefore, the reduction procedure is equivalent to “jump” in polynomial time from $\pi$ to $\pi'$. Replacing $\pi$ by $\pi'$ yields to save nodes in the search for $\pi^*$ and if, for the worst instances, the reduction procedure always applies then the worst-case time complexity of the corresponding Branch-and-Reduce algorithm is lower than that of the brute-force search.

Fig. 1 Illustration of the reduction procedure in a Branch-and-Reduce algorithm
Regarding the literature scheduling problem, Branch-and-Bound algorithms have been often used to efficiently solve them in practice. So, it could appear almost easy to derive from them Branch-and-Reduce algorithms and to analyse their running time. The design of a reduction procedure is far from trivial.

Another way of decomposing the problem to solve consists in applying Dynamic programming. The dynamic programming paradigm is based on breaking down an instance into subproblems. The key idea is to compute only once for each subproblem an optimal solution, to store this solution into a table and to retrieve it each time the corresponding subproblem has to be solved. Dynamic programming has been extensively used in the literature to derive polynomial-time algorithms, pseudo-polynomial time algorithms, polynomial-time approximation schemes (PTAS and FPTAS), …, and it can be also applied to derive exponential algorithms. Typically, exponential algorithms based on dynamic programming require both exponential time and exponential space in the worst case, which is not the case for Branch-and-Reduce algorithms (they usually only require exponential time).

As mentioned by Woeginger [2003], dynamic programming across the subsets enables to derive exponential algorithms. For permutation problems it typically yields to \(O^*(2^n)\) time algorithms against \(O^*(n!)\) for the brute-force search. Dynamic programming across the subsets has been successfully applied by Woeginger on the \(1|\text{prec}|\sum w_i C_i\) problem to build an \(O^*(2^n)\) time and space exponential algorithm. Let \(S\) be a subset of the ground set \([1, \ldots, n]\) such that \(\forall j \in S\) if there exists a precedence relation \(i \rightarrow j\), then \(i \in S\). Let us defined by \(\text{Last}(S) \subseteq S\) the subset of jobs with no successor in \(S\). The recurrence function \(\text{Opt}[S]\) is then defined by:

\[
\begin{aligned}
\text{Opt}[\emptyset] &= 0, \\
\text{Opt}[S] &= \min_{i \in \text{Last}(S)} \{\text{Opt}[S - \{i\}] + w_i P(S)\} \quad \text{with} \quad P(S) = \sum_{i \in S} p_i.
\end{aligned}
\]

It follows that enumerating all subsets \(S\) from the ground set \([1, \ldots, n]\) yields a time and space complexity in \(O^*(2^n)\). Woeginger [2003] also states that this algorithm can be applied to the \(1|d_i|\sum w_i U_i\) and \(1|d_i|\sum w_i T_i\) problems with the same complexity. According to Woeginger, the \(1|r_i, \text{prec}|\sum C_i\) problem can be solved in \(O^*(3^n)\) time using dynamic programming.

Another category of techniques for designing exponential algorithms is based on splitting instances at the cost of an increase in the data. In this category, called Split and List by Fomin and Kratsch [2010], an interesting technique is Sort & Search which has been first proposed by Horowitz and Sahni [1974] to solve the discrete knapsack problem in \(O^*(\sqrt{2^n})\) time and space. The underlying idea is to create a partition, let’s say \(I_1\) and \(I_2\), of a given instance \(I\). Then, by enumerating all possible partial solutions from \(I_1\) and \(I_2\) we may be able to compute the optimal solution corresponding to the instance \(I\). We illustrate this technique on the discrete knapsack problem defined as follows. Let \(O = \{o_1, \ldots, o_n\}\) be a set of \(n\) objects, each one being defined by a value \(v(o_i)\) and a weight \(w(o_i)\), \(1 \leq i \leq n\). We are also given a positive integer capacity \(W\) for the knapsack. The goal is to find a subset \(O' \subseteq O\) such that \(\sum_{o \in O'} w(o) \leq W\) and \(\sum_{o \in O'} v(o)\) is maximum.

The Sort & Search technique suggests to partition \(O\) into \(O_1 = \{o_1, \ldots, o_{\lceil n/2 \rceil}\}\) and \(O_2 = \{o_{\lceil n/2 \rceil + 1}, \ldots, o_n\}\). A first table \(T_1\) is built from \(O_1\) by enumerating all subsets \(O' \subseteq O_1\); a column \(j\) of \(T_1\) corresponds to \(O'_j\) and is associated with the values \(w(O'_j) = \sum_{i \in O'_j} w(o_i)\) and \(v(O'_j) = \sum_{i \in O'_j} v(o_i)\). A second table \(T_2\) is build in the
same way starting from subset \( O_2 \). These two tables have \( O(2^{\frac{7}{2}}) \) columns. Before searching for the optimal solution we perform a sort step on table \( T_2 \): columns \( j \) of \( T_2 \) are sorted by increasing values of \( w(O'_j) \). For each column in position \( k \) after that sorting, we store the index \( \ell_k \leq k \) of the column with maximum \( v(O'_k) \) value i.e. \( \ell_k = \arg\max_{u \leq k} (v(O'_u)) \). This processing, which can be achieved by means of a classic sorting procedure, requires \( O^*(2^{\frac{7}{2}} \log(2^{\frac{7}{2}})) = O^*(\sqrt{2^n}) \) time. Then, a search step is applied to find an optimal solution: for each column \( j \) of table \( T_1 \), we look for the column \( k \) of table \( T_2 \) such that \( w(O'_j) + w(O'_k) \leq W \) and \( v(O'_j) + v(O'_k) \) is maximum. For a given column \( j \), this is achieved by means of a binary search in table \( T_2 \) to find column \( k \) such that \( k = \arg\max_{u \in T_2} (w(O'_j) + w(O'_k) \leq W) \). Then, \( v(O'_j) + v(O'_k) \) is the maximum value of the objective function when objects of \( O'_j \) are put in the knapsack but objects in \( O_1 \setminus O'_j \) are not put in the knapsack. The examination of all \( O'_j \) enables to compute the optimal solution of the problem. The overall search step can be achieved in \( O^*(2^{\frac{7}{2}} \log(2^{\frac{7}{2}})) = O^*(\sqrt{2^n}) \) time. Therefore, this Sort \& Search algorithm requires \( O^*(\sqrt{2^n}) \) time and space. We provide below a numerical example with \( n = 6 \) objects, \( O = \{a, b, c, d, e, f\} \) and \( W = 9 \).

<table>
<thead>
<tr>
<th>( O )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O_1 )</td>
<td>{a, b, c}</td>
<td>{d, e, f}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( O_2 )</td>
<td>{a, b, c}</td>
<td>{d, e, f}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table below presents the result of the search step: for each column \( j \) of \( T_1 \) we indicate the column \( k \) of \( T_2 \) such that \( k = \arg\max_{u \in T_2} (w(O'_j) + w(O'_k) \leq W) \).

<table>
<thead>
<tr>
<th>( j )</th>
<th>( k )</th>
<th>( w(O'_j) + w(O'_k) )</th>
<th>( v(O'_j) + v(O'_k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>{a}</td>
<td>{b}</td>
<td>{c}</td>
</tr>
</tbody>
</table>

Consequently, the optimal solution value is equal to 12 and can be obtained by putting into the knapsack objects \( \{a, b, d\} \) or \( \{b, c, d, e\} \).

The Sort \& Search technique is very powerful to design exponential algorithms and can be applied to a lot of \( \mathcal{NP} \)-hard optimisation problems. Informally speaking, such problems must have the properties that: (1) two partial solutions can be combined in polynomial time to build a complete solution of the initial instance, (2) we must be able to set up a sorting step which enables to perform the searching step in no more time than the building of the tables.

Other techniques, and their analysis, can be found in Fomin and Kratsch [2010].

3 An Introductory Case: The Multiskilled Interval Scheduling Problem

Let us first consider a simple scheduling problem, referred to as IntSched, which serves to introduce several ways for establishing exponential algorithms. IntSched can be
stated as follows. Consider a set of \( n \) jobs to be processed by \( m \) machines. Each job \( i \) is defined by a processing interval \( I_i = [r_i, d_i] \), i.e. starts at time \( r_i \) and completes at time \( d_i \) and, without loss of generality, we assume that \( d_1 \leq d_2 \leq \ldots \leq d_n \). Besides, machines do not all have the same skills or capabilities which implies that to each job \( i \) is defined a subset \( \mathcal{M}_i \) of machines on which it can be processed. The aim of the problem is then to find a feasible assignment of jobs to machines. It is an \( \mathcal{NP} \)-hard problem also referred to as a Ficed Job Scheduling Problem in the literature (Kolen et al [2007], Kovalyov et al [2007]). Notice that when all machines are identical, i.e. \( \forall i, j, \mathcal{M}_i = \mathcal{M}_j \), the problem can be solved in polynomial time since it reduces to a coloring problem in an interval graph.

Let \( \text{Enum} \) be the algorithm which solves the problem \( \text{IntSched} \) by a brute-force search of all possible assignments. This can be achieved in \( O^*(m^n) = O^*(2^n \log_2(m)) \) time. The question is now whether it is possible or not to provide a smaller complexity for the problem \( \text{IntSched} \).

First, consider the dynamic programming algorithm, referred to as \( \text{DynPro} \), defined as follows:

\[
\begin{align*}
\text{Opt}[i, l_1, l_2, \ldots, l_m] &= \text{True} \quad \text{if there exists an assignment of machines to jobs in} \\
\{1, \ldots, i\} \text{ such that } \forall j = 1, \ldots, m, \text{ there is no job} \\
\forall k \in \{1, \ldots, i\} \text{ assigned to machine } j \text{ with } d_k > l_j. \\
\text{Opt}[i, l_1, l_2, \ldots, l_m] &= \text{False} \quad \text{otherwise.}
\end{align*}
\]

In \( \text{Opt} \) the \( l_j \)'s are upper bounds on the completion times of the last jobs from \( \{1, \ldots, i\} \) scheduled on the machines. If we denote by \( \mathcal{M}_i^R = \{ j \in \mathcal{M}_i \mid l_j \geq d_i \} \), then the recurrence function can be rewritten as:

\[
\begin{align*}
\text{Opt}[i, l_1, l_2, \ldots, l_m] &= \forall u \in \mathcal{M}_i^R \text{Opt}[i - 1, l_1, \ldots, l_u = r_i, \ldots, l_m] \quad \forall i = 1, \ldots, n \\
\text{Opt}[0, l_1, l_2, \ldots, l_m] &= \text{True} \quad \forall l_1, \ldots, l_m
\end{align*}
\]

with \( \forall u \in \mathcal{M}_i^R \text{Opt}[i - 1, l_1, \ldots, l_u = r_i, \ldots, l_m] = \text{False} \) if \( \mathcal{M}_i^R = \emptyset \). \( \text{DynPro} \) first calculates all relevant tuples \( (l_1, \ldots, l_m) \) in a recursive way. Starting with \( l_j = d_{\text{max}} = \max_{1 \leq j \leq n} (d_j) \), \( \forall j = 1, \ldots, m, \) all tuples \( (l_1, \ldots, l_u = r_i, \ldots, l_m), \forall u \in \mathcal{M}_i^R \) are calculated. Recursively, for each of these tuples we iterate with \( \mathcal{M}_{i - 1}^R, \mathcal{M}_{i - 1}^R \). \( \text{DynPro} \) next builds \( n \) tables containing the values of \( \text{Opt} \): table \( i \) contains the values for the set of jobs \( \{1, \ldots, i\} \) and is built once table \((i - 1)\) is known. Besides, the columns of table \( i \) are the tuples generated at the \( (n - i) \)th recursion. if \( \forall u \in \mathcal{M}_i^R \text{Opt}[i - 1, l_1, \ldots, l_u = r_i, \ldots, l_m] \) is true then there exists a feasible assignment of jobs to machines, which can be calculated in polynomial time by a backward procedure as usual in dynamic programming.

**Lemma 1** \( \text{DynPro} \) has a worst-case complexity in \( O^*(2^{(m+1) \log_2(n)}) \).

**Proof** To calculate the tables containing the values of \( \text{Opt}[i, l_1, \ldots, l_m] \) we need to consider the set of possible values for the parameters. Each parameter can take at most \( n \) values which implies that there are at most \( n^{m+1} \) values of the recurrence function to calculate. Besides, for any given value \( \text{Opt}[i, l_1, \ldots, l_m] \) we need to evaluate \( \forall u \in \mathcal{M}_i^R \text{Opt}[i - 1, l_1, \ldots, l_u = r_i, \ldots, l_m] \) which is done by accessing to, at most, \( m \)
values $Opt[i-1, l_1, \ldots, l_u = r_1, \ldots, l_m]$ already evaluated. Thus, the time complexity is, at worst, in $O(m \times n^{(m+1)}) = O^*(n^{(m+1)}) = O^*(2^{(m+1) \log_2(n)})$. This is also the space complexity of the algorithm.

From lemma 1 we can see that: (i) whenever $m$ is fixed, the IntSched problem becomes polynomially solvable, (ii) DynPro algorithm offers a better complexity than Enum, whenever $n > m$.

In order to derive exponential algorithms for IntSched, we can also reduce it to known graph problems. Consider the following algorithm, referred to as StaDom, which first transforms an instance of the IntSched problem into a graph. Let $G = (V, E)$ be an undirected graph in which each vertex $v_i \in V$ represents a couple $(I_j, \ell)$ with $\ell \in M_j$. Therefore, for a given job we create as much vertices as machines capable of processing it. We create an edge $e_k \in E$ between two nodes $v_i = (I_j, \ell)$ and $v_p = (I_q, \ell')$ iff $I_j \cap I_q \neq \emptyset$ and $\ell = \ell'$. We also create an edge between two vertices associated to the same job. This yields a graph $G$ with at most $N = nm$ vertices and $M = n^2 m^2$ edges. On this graph, StaDom applies the exact algorithm for the Maximum Independent Set problem in $O^*(1.2132^N)$ (Kneis et al [2009]). The example provided in figure 2 illustrates the reduction of the IntSched problem to the search of an independent set $S$ of maximum size in the graph $G$.

**Fig. 2** Reduction of IntSched to the search of a independent set of maximum size in a graph: a 4-job and 3-machine example

**Lemma 2** StaDom solves the IntSched problem with a worst-case time complexity in $O^*(1.2132^{nm})$ and polynomial space.

**Proof** We first show that if there exists an independent set $S$ of cardinality $n$ in the graph $G$ then there exists a feasible solution to the associated instance of the IntSched problem. For each vertex $v_i \in S$, let $(I_j, \ell)$ be the associated time interval of job $j$ and the machine $\ell \in M_j$. By construction of the graph, there is no other vertex $v_k \in S$ associated to the couple $(I_u, \ell')$ such that one of the two conditions holds:

1. $u = j$,
2. $u \neq j$, $\ell = \ell'$ and $I_u \cap I_j \neq \emptyset$.

Both conditions lead to a contradiction with the fact that $S$ is an independent set of maximum size since there is an edge between $v_i$ and $v_k$. Consequently, as there are $n$ vertices in $S$, one for each job of IntSched and with each machine assigned to a single job at the same time, then $S$ can be easily translated into a feasible assignment for the IntSched problem.

By applying the same argument we can easily show that if there does not exist an independent set $S$ of cardinality $n$ on graph $G$, there does not exist a feasible solution to the associated IntSched problem. □

Now, we establish another result by considering another reduction of the IntSched problem to a graph problem. Consider the following algorithm, referred to as LisCol,
which first transforms an instance of the \emph{IntSched} problem into a graph. Let $G = (V, E)$ be an undirected graph in which each vertex $v_i \in V$ represents a job $i$ and is associated with a set of colors $C_i$: color $\ell \in C_i$ iff machine $\ell \in \mathcal{M}_i$. We create an edge $e_k \in E$ between two nodes $v_i$ and $v_p$ iff $I_j \cap I_q \neq \emptyset$. This yields a graph $G$ with $N = n$ vertices and at most $M = n^2$ edges. On this graph, \emph{LisCol} applies the algorithm for the list-coloring problem with worst-case complexity in $O^*(2^N)$ (Björklund et al [2009]). The example provided in figure 3 illustrates the reduction of the \emph{IntSched} problem to the search of a list-coloring $L$ in the graph $G$. This reduction leads to the result of lemma 3.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{Reduction of \emph{IntSched} to the search of a list-coloring in a graph: a 4-job and 3-machine example}
\end{figure}

\textbf{Lemma 3} \emph{LisCol} solves the \emph{IntSched} problem with a worst-case complexity in $O^*(2^n)$.

The question is now whether one of these four algorithms outperforms, in terms of complexity, the others or not: \emph{Enum} is in $O^*(2^{n \log_2(m)})$, \emph{DynPro} is in $O^*(2^{(m+1) \log_2(n)})$, \emph{StaDom} is in $O^*(1.2132^{nm}) = O^*(2^{\frac{1.2132 \cdot n^2}{m}})$ and \emph{LisCol} is in $O^*(2^n)$. From these complexities we can note that:

\begin{itemize}
  \item \emph{LisCol} has a lower worst-case complexity than \emph{Enum},
  \item For $m \leq 3$, the worst-case running time of \emph{StaDom} is better than \emph{LisCol},
  \item For $m \leq 13$, the worst-case running time of \emph{StaDom} is better than \emph{Enum}.
\end{itemize}

It follows that, among \emph{Enum}, \emph{StaDom} and \emph{LisCol}, the latter has the lowest complexity for values of $m$ higher than 3 whilst \emph{StaDom} is better for $m$ lower than 3. \emph{DynPro} has a complexity which can be better than the one of \emph{LisCol}, depending on the size of the instances; for example, this is the case for any instance with $m \leq 10$ and $n \geq 1000$. But, on the other hand, for any instance with $n \leq 60$ and $m \geq 10$, the worst-case running time of \emph{LisCol} is better than \emph{DynPro}.

In this section we provided an illustration of the notion of worst-case complexity and we showed complexity results by exploiting, for \emph{IntSched}, strong links with graph problems. Unfortunately, most often this manner to show complexity results does not hold since \emph{NP}-hard scheduling problem, in general, involve data related to duration or date (processing times, due dates, \ldots). This makes them harder than classical unweighted graph problems.

\section*{4 Single Machine Scheduling Problems}

\subsection*{4.1 A General Result for Decomposable Problems}

Consider \emph{n} jobs to be scheduled without preemption on a single machine available from time 0 onwards. Each job $i$ is defined by a processing time $p_i$ and completes at time $C_i(s)$ in a given schedule $s$ (whenever there is no ambiguity we omit $s$ in
the notation). Additionally, to each job is associated a cost function $f_i$. We also assume that the decomposability property of definition 1 holds. The aim is to calculate a schedule $s$ (a sequence of jobs) which minimizes either criterion $f_{\text{max}}(s) = \max_{1 \leq i \leq n}(f_i(C_i(s)))$ or criterion $\sum f_i(s) = \sum_{i=1}^{n} f_i(C_i(s))$. We assume that for any given schedule $s$ these criteria can be evaluated in polynomial time. These two problems, which are referred to as $1|\text{dec}|f_{\text{max}}$ and $1|\text{dec}|\sum f_i$, generalize a set of basic $NP$-hard scheduling problems like the $1|d_i|\sum w_iC_i$, $1|d_i|\sum w_iT_i$, $1|d_i, d_i|\sum w_iC_i$, $1|d_i, d_i|\sum w_iT_i$, $1|d_i|\sum w_iU_i$, $1|d_i, d_i|\sum w_iU_i$, $1|d_i, \text{dec}|\sum w_iE_i$ and $1|d_i, d_i, \text{dec}|\sum w_iE_i$ problems.

First, consider the algorithm $Enum$ which solves the problems $1|\text{dec}|f_{\text{max}}$ or $1|\text{dec}|\sum f_i$ by a brute-force search of all possible schedules. As the number of such schedules (sequences of $n$ jobs) is equal to $n!$ the $Enum$ algorithm has a worst-case complexity in $O^*(n!)$ time. It is possible to establish better bounds by means of a dynamic programming algorithm, denoted by $DynPro$ and introduced by Fomin and Kratsch [2010].

For the $1|\text{dec}|\sum f_i$ problem, $DynPro$ works as follows. Let be $S \subseteq \{1, \ldots, n\}$ and $Opt[S]$ the recurrence function calculated on set $S$: $Opt[S]$ is equal to the minimal value of criterion $\sum f_i$ for the jobs in $S$. We have:

\[
\begin{cases}
    Opt[\emptyset] = 0, \\
    Opt[S] = \min_{t \in S} \{Opt[S - \{t\}] + f_t(P(S))\} \quad \text{with } P(S) = \sum_{i \in S} p_i.
\end{cases}
\]

Notice that in the presence of additional constraints, like deadlines $d_i$, the above formulation must be slightly changed as follows: when computing the minimum value over $t \in S$, only jobs satisfying these additional constraints must be considered. In the case of deadlines, only jobs $t$ with $d_t \geq P(S)$ have to be considered. $DynPro$ has a worst-case time and space complexity in $O^*(2^n)$. It can be easily adapted to solve the $1|\text{dec}|f_{\text{max}}$ problem.

In the next section, we refine the worst-case complexity of a particular single machine decomposable problem.

### 4.2 The Problem of Minimizing the Weighted Number of Late Jobs

Consider that each job $i$ is defined by a processing time $p_i$, a due date $d_i$ and a tardiness penalty $w_i$. The aim is to compute an optimal schedule $s$ which minimizes the weighted number of late jobs denoted by $\sum w_iU_i$ with $U_i = 1$ if $C_i(s) > d_i$ and $U_i = 0$, otherwise. This problem, which is referred to as $1|d_i|\sum w_iU_i$, has been shown $NP$-hard in the weak sense (Karp [1972] and Lawler and Moore [1969]). We first show some simple properties.

**Lemma 4** Let $E$ be a set of desired early jobs, i.e. jobs that we would like to complete before their due date $d_i$. Either there is no feasible schedule $s$ in which all jobs in $E$ are early, either there exists an optimal schedule in which all jobs in $E$ are sequenced by increasing value of their due date $d_i$ (Earliest Due Date rule, EDD).

**Proof** The EDD rule has been shown to optimally solve the $1|d_i|L_{\text{max}}$ problem (Jackson [1955]). Let $s_{\text{EDD}}$ be the schedule of jobs obtained by sequencing the jobs in $E$ according to the EDD rule. Since there is no other schedule $s'$ of $E$ with $L_{\text{max}}(s') <$
Lemma 5 Let \( s_{\text{EDD}} \) be the schedule obtained by the EDD rule on a set of early jobs \( E \), with \( L_{\max}(s_{\text{EDD}}) \leq 0 \). There exists a feasible schedule of all jobs in \( E \) starting at time \( t \) iff \( L_{\max}(s_{\text{EDD}}) + t \leq 0 \).

Proof In \( s_{\text{EDD}} \) the first job starts at time \( t = 0 \) and we have \( L_{\max}(s_{\text{EDD}}) = \max_{i \in s_{\text{EDD}}} (C_i(s_{\text{EDD}}) - d_i) \). Now, assume that the first early job of \( E \) starts at time \( t > 0 \). Then, due to the optimality of the EDD rule there exists a feasible schedule in which all jobs in \( E \) remain early and start after time \( t \) iff \( C_i(s_{\text{EDD}}) + t \leq d_i, \forall i \in E \) which is equivalent to \( L_{\max}(s_{\text{EDD}}) + t \leq 0 \).

First, consider the \( \text{Enum} \) algorithm which solves the problem by a brute-force search of all schedules. From lemma 4 we can deduce that \( \text{Enum} \) has only to enumerate all the sets \( E \) of possible early jobs and, for each set \( E \), calculate in polynomial time as suggested in the proof of that theorem an associated schedule \( s \). By keeping the schedule \( s \) with the minimal value of \( \sum w_i U_i \), \( \text{Enum} \) can solve optimally the problem. As there are \( 2^n \) sets of possible early jobs, \( \text{Enum} \) has a worst-case complexity in \( O^*(2^n) \) time. This complexity can also be deduced from the \( \text{DynPro} \) algorithm proposed in section 4.1. The question is whether it is possible or not to establish a better bound. To that purpose we apply the Sort \& Search approach to derive the following optimal algorithm, referred to as \( \text{SorSea} \). Without loss of generality, jobs are assumed to be numbered by increasing order of their due date, i.e. \( d_1 \leq d_2 \leq \ldots \leq d_n \). Let be \( I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor \} \) and \( I_2 = \{ \lceil \frac{n}{2} \rceil + 1, \ldots, n \} \) a partition of the initial instance to solve. Starting from set \( I_1 \), algorithm \( \text{SorSea} \) builds a sequence of early jobs scheduled first, whilst starting from set \( I_2 \) it builds a sequence of desired early jobs scheduled right after the early jobs of \( I_1 \). Let \( s_1^k \subseteq I_1 \) (resp. \( s_2^k \subseteq I_2 \)) be a sequence of early jobs sorted by the EDD rule, and \( \bar{s}_1^k = I_1 - s_1^k \) (resp. \( \bar{s}_2^k = I_2 - s_2^k \)) be the sequence of tardy jobs (in any order). The decomposition of a schedule \( s \) computed by \( \text{SorSea} \) in presented in figure 4. We also define \( P(A) = \sum_{i \in A} p_i \) for any set of jobs \( A \). We have:

\[
\sum_{i=1}^{n} w_i U_i(s) = \sum_{i \in s_1^k} w_i + \sum_{i \in \bar{s}_1^k} w_i.
\]

Fig. 4 Decomposition of a schedule \( s \) for the 1\(|d_i|\sum w_i U_i \) problem

\( \text{SorSea} \) builds a table \( T_1 \) in which each column \( j \) is associated with a sequence \( s_1^j \subseteq I_1 \) of at most \( \frac{n}{2} \) jobs. Therefore, table \( T_1 \) contains at most \( 2^n \) columns. To each column \( j \) we store the values \( P(s_1^j) \) and \( \sum_{i \in s_1^j} w_i \). \( \text{SorSea} \) also builds a table \( T_2 \) in which column \( k \) is associated with a sequence \( s_2^k \subseteq I_2 \) of at most \( \frac{n}{2} \) jobs. In table \( T_2 \) the \( 2^n \) columns are sorted by decreasing values of \( L_{\max}(s_2^k) \). For each column \( k \) we store the values \( L_{\max}(s_2^k) \), \( \sum_{i \in s_2^k} w_i \) and \( wU_{\min}(s_2^k) = \min_{l \geq k} (\sum_{i \in s_2^l} w_i) \).

For a given column \( j \) of \( T_1 \), i.e. with associated partial sequences \( s_1^j \) and \( \bar{s}_1^j \), \( \text{SorSea} \) searches in \( O(n) \) time in \( T_2 \) the column \( k \) such that:
\[ k = \text{argmin}(u \in T_2 \mid P(s^*_1) + L_{\text{max}}(s^*_2) \leq 0). \]

From lemma 5, we can deduce that all columns \( \ell \geq k \) in table \( T_2 \) correspond to all the partial schedules \( s_2^* \) with no tardy job if they are scheduled after \( s_1^* \). The value of the smallest \( \sum_{i \in s_1^*} w_i U_i(s) \) value in a schedule \( s \) starting by the partial sequence \( s_1^* \) of early jobs and with jobs in \( s_1^* \) tardy is then given by:

\[
\sum_{i \in s_1^*} w_i U_i(s) = \sum_{i \in s_1^*} w_i U_i + w U_{\text{min}}(s_2^*).
\]

By computing for each column \( j \) of \( T_1 \) the above value, SorSea computes the optimal solution of the \( 1|d_i|\sum w_i U_i \) problem.

**Theorem 1** SorSea solves the \( 1|d_i|\sum w_i U_i \) problem with a worst-case time and space complexity in \( O^*(\sqrt{2^n}) \).

**Proof** First, SorSea builds table \( T_1 \), thus requiring \( O^*(\sqrt{2^n}) \) time and space. Next, it builds table \( T_2 \) also in \( O^*(\sqrt{2^n}) \) time and space since the sorting of the columns is done in \( O^*(2^{2n} \times \log(2^n)) = O^*(\sqrt{2^n}) \) time. The main part of SorSea algorithm consists in searching for each column \( j \) of \( T_1 \), the column \( k \) in \( T_2 \) such that \( k = \text{argmin}(u \in T_2/ P(s^*_1) + L_{\text{max}}(s^*_2) \leq 0). \) By a binary search, whenever \( j \) is given, the value of \( k \) can be computed in \( O^*(\log(2^n)) = O(n) \) time, i.e. in polynomial time. As there are \( 2^n \) columns in table \( T_1 \), the search for the optimal solution in tables \( T_1 \) and \( T_2 \) can be achieved in \( O^*(\sqrt{2^n}) \) time and space. \( \square \)

### 5 Parallel Machine Scheduling Problems

#### 5.1 A General Result for Decomposable Problems

Consider \( n \) jobs to be scheduled without preemption on \( m \) identical parallel machines available from time 0 onwards. Each job \( i \) is defined by a processing time \( p_i \) and completes at time \( C_i(s) \) on the machine \( j \) which processes it in a given schedule \( s \). To each job is associated a cost function \( f_i \). We also assume that the decomposability property of definition 2 holds. The aim is to calculate a schedule \( s \) (sequences of jobs on the machines) which minimizes either criterion \( f_{\text{max}}(s) \) or criterion \( \sum f_i(s) \). The two problems tackled in this section, referred to as \( P[\text{dec}|f_{\text{max}} \text{ or } P[\text{dec}] \sum f_i \), generalize that of section 4.1 and are strongly \( NP \)-hard. They also generalize some basic scheduling problems like the \( P|[C_{\text{max}}, P|d_i|T_{\text{max}}, P|d_i|L_{\text{max}}, P||\sum w_i C_i, P|d_i|\sum T_i, P|d_i|\sum w_i T_i, P|d_i|\sum w_i U_i, P|d_i, \text{dec}] \sum w_i E_i \) problems and their variant with deadlines.

First, consider the algorithm Enum which solves the problems \( P[\text{dec}|f_{\text{max}} \text{ or } P[\text{dec}] \sum f_i \) by a brute-force search of all possible schedules. A schedule is defined by sets of jobs on machines \( j \), each set leading to \( n_j! \) permutations in the worst-case. For a given assignment of jobs to machines, the number of schedules is given by \( \prod_{j=1}^{m} n_j! \), which is lower than \( n! \). Besides, there are \( n^m \) possible assignments of \( n \) jobs to \( m \) machines thus leading to a worst-case time complexity of Enum in \( O^*(m^n n!) \).

Notice that this complexity is an upper bound on its exact complexity which, to be established, would require to compute the partition of a number \( n \) into \( k \) numbers with \( 1 \leq k \leq m \), as defined in number theory. There does not exist, to the best of our knowledge, a general formulae giving the number of such partitions.
We now show that it is possible to provide a strongly reduced bound, by means of a dynamic programming algorithm and a suitable decomposition of the problem. The resulting algorithm is denoted by DecDP and is presented for the $P_{|dec|} f_{\max}$ problem. However, it can be easily adapted to the $P_{|dec|} \sum f_i$ problem.

The main line of DecDP is to separate recursively the set of machines into two “equal-size” subsets, thus leading to $O(\log_2(m))$ subproblems ($P_i$) to deal with. This decomposition is illustrated in figure 5 in the case of $m = 8$ machines. If $m$ is not a power of 2 then for some subproblems there is an odd number of machines and in ($P_1$) there is a single machine. However, this does not change the functioning of DecDP.

**Fig. 5** Illustration of the recursive decomposition of problems $P_{|dec|} \sum f_i$ and $P_{|dec|} f_{\max}$

We present this algorithm in the case where $m$ is a power of 2. Let us denote by $X^k$ the set of sets of $k$ jobs among $n$ and let be $X = \cup_{1 \leq k \leq n} X^k$. We define ($P_1$) as the problem of scheduling a set $S$ of jobs on $2^k$ machines and we denote by $F_1[S]$ the optimal value of $\sum f_i$ for the jobs in $S$.

First, DecDP solves the problem ($P_0$) which involves a single machine and is denoted by $1_{|dec|} \sum f_i$. The latter can be solved in $O^*(2^n)$ by DynPro presented in section 4.1. This algorithm computes the optimal solution of $\sum f_i$ criterion for all subsets $S \in X$: let be $\sigma_S$ the optimal sequence associated to subset $S$, $\forall S \in X$, then $F_0[S] = \sum f_i(\sigma_S)$ can be computed in $O(1)$ time after running of DynPro.

Next, for each value $t$ from 1 to $\log_2(m)$, we have to compute $F_t[S], \forall S \in X$. This is done by computing $F_t[S] = \min_{S' \subseteq S}(F_{t-1}[S'] + F_{t-1}[S \setminus S'])$. For instance, for problem ($P_1$) and a given $S \in X$, $F_1[S]$ is computed by trying all possible assignments of jobs in $S$ on machines 1 and 2 and by using the values $F_0$ computed by DynPro. Similarly, for problem ($P_2$) and a given $S \in X$, $F_2[S]$ is computed by trying all possible assignments of jobs in $S$ on the couples (machine 1, machine 2) and (machine 3, machine 4) and by using the values $F_1$ previously computed. This process is repeated until we are able to compute $F_{\log_2(m)}[\{1, \ldots, n\}]$.

**Theorem 2** DecDP solves the $P_{|dec|} \sum f_i$ problem with a worst-case time complexity in $O^*(3^n)$ and a worst-case space complexity in $O^*(2^n)$.

**Proof** First, DecDP computes sets $X^k$ and $X$ which can be achieved in $O^*(2^n)$ time and space. This is also the case of DynPro algorithm used to compute $F_0[S], \forall S \in X$. For a given problem ($P_1$), all $F_1[S]$ values can be computed in $O^*(3^n)$ time: for a given set $S$ there are $2^{|S|}$ subsets $S'$ and as there are $\binom{n}{k}$ sets of cardinality $k$, we have to access $O(\sum_{k=0}^{n} \binom{n}{k} 2^k)$ times to $F_{t-1}$ (each access is done in $O(1)$ time). By using the Newton’s binomial formula, $\sum_{k=0}^{n} \binom{n}{k} 2^k$ can be rewritten as $3^n$, thus leading to a time complexity in $O(3^n)$ for computing $F_1[S], \forall S \in X$. The memory space required is in $O(2^n)$.

As there are $\log_2(m)$ problems ($P_i$) to consider, they are all solved in $O^*(\log_2(m)3^n) = O^*(3^n)$ time. Consequently, DecDP requires $O^*(3^n)$ time and $O^*(2^n)$ space. □
In this section we focus on a sub-problem of the makespan minimization. We consider the case where \( m \) is not a power of 2, and there are \( \lceil \log_2(m) \rceil \) problems \((P_t)\) to solve, and the problem \((P_0)\) involves a single machine. Then, it is solved by the \( \text{DynPro} \) algorithm presented in section 4.1 and no problem \((P_0)\) has to be solved. For values \( t \) from 2 to \( \lceil \log_2(m) \rceil \) the recurrence function \( F_t[S] \) does not change.

The same result can be established for the \( P|\text{dec}|f_{\text{max}} \) problem by slightly changing the definition of \( F_t[S] \) by \( F_t[S] = \min_{S' \subseteq S} \{ \max(F_{t-1}[S'], F_{t-1}[S \setminus S']) \} \).

### 5.2 The Two Machine Problem with Makespan Minimization

In this section we focus on a sub-problem of the \( P|\text{dec}|f_{\text{max}} \) problem which is referred to as \( P2||C_{\text{max}} \) and defined as follows. Consider \( n \) jobs to be scheduled without preemption on two parallel identical machines available from time 0 onwards. Each job \( i \) is defined by a processing time \( p_i \) and completes at time \( C_i(s) \) on the machine \( j \) which processes it in a given schedule \( s \). The aim is to calculate a schedule \( s \) (an assignment of jobs to machines) which minimizes the makespan \( C_{\text{max}} \). This problem, which has been shown \( \mathcal{NP} \)-hard in the weak sense (Lenstra et al. [1977]), can be also modeled as a \( \text{SUBSET SUM} \) problem (Garey and Johnson [1979]).

First, consider the algorithm \( \text{Enum} \) which solves the problem \( P2||C_{\text{max}} \) by a brute-force search of all possible schedules. A schedule is defined by a partition of the set of jobs into 2 sets, one for each machine. Therefore, there are at most \( O(2^n) \) partitions and \( \text{Enum} \) requires \( O^*(2^n) \) time. This bound is lower than that of given for the more general \( P|\text{dec}|f_{\text{max}} \) problem. However, we show that it is possible to provide a reduced bound by application of the \( \text{Sort} \& \text{Search} \) method in a similar way than already done by Horowitz and Sahni [1974] for the \( \text{SUBSET SUM} \) problem.

\( \text{SortSea} \) works as follows. Let \( I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor \} \) and \( I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\} \) be a decomposition of the instance. Starting from \( I_1 \) it builds an assignment of jobs at the beginning on machine 1 and on machine 2, whilst from set \( I_2 \) it builds and assignment of jobs at the end of the schedule. Given a set \( s^1_i \subseteq I_1 \) (resp. \( s^2_i \subseteq I_2 \)) of jobs assigned on machine 1 we note \( \tilde{s}^1_i = I_1 - s^1_i \) (resp. \( \tilde{s}^2_i = I_2 - s^2_i \)) the set of jobs assigned on machine 2 (figure 6). We also define \( P(A) = \sum_{i \in A} p_i \) for any set of jobs \( A \), and we have:

\[
C_{\text{max}}(s) = \max(P(s^1_i) + P(s^2_j), P(s^1_i) + P(s^2_j)).
\]

**Fig. 6** Decomposition of a schedule for the \( P2||C_{\text{max}} \) problem

\( \text{SortSea} \) builds a table \( T_1 \) in which each column \( j \) is associated with an assignment \( s^1_j \subseteq I_1 \) of at most \( \frac{n}{2} \) jobs. To each column \( j \) are associated the values of \( P(s^1_j) \) and \( P(s^1_{j+1}) \). Next, \( \text{SortSearch} \) builds a table \( T_2 \) in which each column \( n \) is associated with an assignment \( s^2_n \subseteq I_2 \) of at most \( \frac{n}{2} \) jobs. These one are sorted by non increasing values of \( (P(s^2_n) - P(s^2_{n-1})) \). To each column \( k \) are associated the values \( P(s^2_k) \), \( P(s^2_{k+1}) \), \( \ldots \)
\[ P^d_{\min}(s_j^2) = \min_{k \geq k}(P(s_j^2)) \] and \[ P^g_{\min}(s_j^2) = \min_{\ell \leq k}(P(s_j^2)) \].

For a given column \( j \) from \( T_1 \), i.e. assignments \( s_1^j \) and \( \hat{s}_1^j \), SorSea searches in table \( T_2 \) the indexes \( k \) and \( \ell \) such that:

\[
\begin{align*}
 k &= \arg\min(u \in T_2 | P(s_1^j) - P(s_1^j) \geq P(s_2^j) - P(s_2^j)), \\
 \ell &= \arg\max(u \in T_2 | P(s_1^j) - P(s_1^j) \leq P(s_2^j) - P(s_2^j)).
\end{align*}
\]

Then, SorSea deduces the smallest value of \( C_{\text{max}}(s) \) in a schedule starting by \( s_1^j \) on machine 1 and by \( \hat{s}_1^j \) on machine 2:

\[
C_{\text{max}}(s) = \min(P(s_1^j) + P^d_{\min}(s_j^2), P(\hat{s}_1^j) + P^g_{\min}(s_j^2)).
\]

The optimal value of \( C_{\text{max}} \) is obtained by applying the above search into table \( T_2 \) for each column \( j \) from table \( T_1 \) and by keeping the smallest value \( C_{\text{max}} \) found.

**Theorem 3** SorSea solves the \( P2||C_{\text{max}} \) problem with a worst-case time and space complexity in \( O^*(\sqrt{2^n}) \).

**Proof** First, SorSea builds table \( T_1 \), thus requiring \( O^*(\sqrt{2^n}) \) time and space. Next, it builds table \( T_2 \) also in \( O^*(\sqrt{2^n}) \) time and space since the sorting of the columns is done in \( O^*(2^k \times \log(2^k)) = O^*(\sqrt{2^n}) \) time. The main part of SorSea algorithm consists in searching for each column \( j \) of \( T_1 \) the columns \( k \) and \( \ell \) in \( T_2 \) such that \( k = \arg\min(u \in T_2 | P(s_1^j) - P(s_2^j) \geq P(s_3^j) - P(s_3^j)) \) and \( \ell = \arg\max(u \in T_2 | P(s_1^j) - P(s_2^j) \leq P(s_3^j) - P(s_3^j)) \). By a binary search, whenever \( j \) is given, the values of \( k \) and \( \ell \) can be computed in \( O^*(\log(2^k)) = O(n) \) time. As there are \( 2^k \) columns in table \( T_1 \), the search for the optimal solution in tables \( T_1 \) and \( T_2 \) can be achieved in \( O^*(\sqrt{2^n}) \) time. \( \square \)

### 5.3 The Two Machine Problem with the Weighted Number of Late Jobs

In this section we focus on a sub-problem of the \( P|\text{dec}|\sum f_i \) problem which is referred to as \( P2|d_i|\sum w_i U_i \) and defined as follows. Consider \( n \) jobs to be scheduled without preemption on two identical parallel machines available from time 0 onwards. Each job \( i \) is defined by a processing time \( p_i \), a due date \( d_i \), a tardiness penalty \( w_i \), and completes at time \( C_t(s) \) on the machine which processes it in a given schedule \( s \). Without loss of generality, we assume that jobs are indexed such that \( p_1 \leq p_2 \leq \ldots \leq p_n \). The aim is to calculate a schedule \( s \) (an assignment of jobs on the two machines) which minimizes the weighted number of late jobs \( \sum w_i U_i \). This problem has been shown \( \mathcal{NP} \)-hard in the weak sense (Graham et al [1979]), even in the case \( w_i = 1, \forall i = 1, \ldots, n \).

First, we concentrate on some properties of the problem and the brute-force search Enum algorithm. Lemma 4 (section 4.2) still holds on each one of the machines as far as the sets of early jobs they process are known. From theorem 4 we can deduce that Enum has only to enumerate all the sets of possible early jobs on each machine and, for each set \( E_j \) of early jobs on machine \( j \), to calculate in polynomial time an associated schedule \( s \) (schedule on any machine, at the end, the tardy jobs). By keeping the schedule \( s \) with the minimal value of \( \sum w_i U_i \), Enum can solve optimally the problem. As each job can be either early on machine 1, early on machine 2 or tardy, there are \( 3^n \) sets of possible early jobs and Enum is in \( O^*(3^n) \) time. This complexity is also that of the DynPro algorithm proposed in section 4.1. The question is whether it is possible
or not to establish a smaller bound.
We now state a result which extends lemma 5.

**Lemma 6** Let $E_1$ (resp. $E_2$) be a set of early jobs assigned on machine 1 (resp. machine 2) and $s_{\text{EDD}}$ be the schedule obtained by applying the EDD rule on each machine to sequence $E_1$ and $E_2$. We have $L_{\text{max}}(s_{\text{EDD}}) \leq 0$. There exists a feasible schedule of all jobs in $E_1$ and $E_2$ starting at time $t$ iff $L_{\text{max}}(s_{\text{EDD}}) + t \leq 0$.

**Proof** Follows directly from lemma 5.

As for the $1|d_i|\sum w_i U_i$ problem, we propose a Sort & Search approach, referred to as SorSea. Let $I_1 = \{1, \ldots, n_1\}$ and $I_2 = \{n_1 + 1, \ldots, n\}$ a decomposition of the initial instance (we note $n_2 = |I_2|$). Starting from $I_1$ we build a sequence of jobs “first” on machines 1 and 2, whilst starting from $I_2$ we build a sequence of jobs “second” on that machines. For a given $s^1 \subseteq I_1$ (resp. $s^2 \subseteq I_2$), i.e. a sequence of early jobs assigned “first” (resp. “second”) either on machine 1 or machine 2, we denote by $s^1_i = I_1 - s^1_i$ (resp. $s^2_i = I_2 - s^2_i$) the set of tardy jobs assigned “first” (resp. “second”) either on machine 1 or machine 2. This decomposition of a schedule is illustrated in figure 7. Notice that, with respect to the optimization criterion, we do not care about the position or the machine which processes the tardy jobs: so, they can be scheduled anywhere in a schedule, but after $s^1_i$ and $s^2_i$. We have:

$$\sum w_i U_i(s) = \sum_{i \in s^1_i} w_i + \sum_{i \in s^2_i} w_i.$$

**Fig. 7** Decomposition of a schedule for the $P2|d_i|\sum w_i U_i$

In addition to the above decomposition scheme, SorSea exploits the symmetry induced by the fact that the two machines are identical. Figure 8 shows that, when the partial schedule $s^1_i$ is fixed, we can switch in the partial schedule $s^2_i$ the sequences on machines 1 and 2 to build two schedules. This enables to derive a simple condition to check that there exists a feasible schedule starting with $s^1_i$ and ending with $s^2_i$ in which all jobs are early. We make use of the following additional notations: $\forall \ell = 1, 2$, $s^1_\ell$ (resp. $s^2_\ell$) refers to the sequence of jobs from $s^1_i$ assigned on machine 1 (resp. machine 2). We also define $P(A) = \sum_{i \in A} p_i$, for any given set $A$, $C_{\text{min}}(s^1_\ell) = \min(P(s^1_\ell), P(s^2_\ell))$, $C_{\text{max}}(s^1_\ell) = \max(P(s^1_\ell), P(s^2_\ell))$, $L_{\text{min}}(s^2_\ell) = \min(\max_{i \in s^2_\ell : \{C_i - d_i\}} \max(\max_{i \in s^2_\ell : \{C_i - d_i\}} \max_{i \in s^2_\ell : \{C_i - d_i\}}))$ and $L_{\text{max}}(s^2_\ell) = \max(\max_{i \in s^2_\ell : \{C_i - d_i\}} \max_{i \in s^2_\ell : \{C_i - d_i\}})$.

**Fig. 8** Partial sequences fitting
Theorem 4 Let be \( s_1^k \) (resp. \( s_2^k \)) a partial schedule of early jobs. There exists a feasible schedule starting with \( s_1^k \) and ending with \( s_2^k \) iff the following system holds:

\[
\begin{aligned}
-L_{\text{max}}(s_2^k) &\geq C_{\text{min}}(s_1^k) \\
-L_{\text{min}}(s_2^k) &\geq C_{\text{max}}(s_1^k)
\end{aligned}
\]  

(A)

Proof Without loss of generality, we assume that \( C_{\text{min}}(s_1^k) = P(s_1^{t,1}) \) and \( L_{\text{max}}(s_2^k) = \max_{i \in s_2^k} (C_i - d_i) \) (if this does not hold, by symmetry, \( s_2^{k,1} \) and \( s_2^{k,2} \) can be switched).

Lemma 5 applied on machine 2 states that there exists a feasible schedule of early jobs on that machine iff \( C_{\text{min}}(s_1^k) + L_{\text{max}}(s_2^k) \leq 0 \). Similarly, there exists a feasible schedule of early jobs on machine 1 iff \( P(s_1^{t,1}) + \min_{i \in s_2^k} (C_i - d_i) \leq 0 \), i.e. \( C_{\text{max}}(s_1^k) + L_{\text{min}}(s_2^k) \leq 0 \). This gives system (A).

The current theorem is true since if there is no feasible schedule, a permutation of \( s_2^{k,1} \) and \( s_2^{k,2} \) does not lead to a schedule in which all jobs are early.

SortSea works in a different way than the classic 2-table approach already used in this paper. To the best of our knowledge, this approach does work for the \( P^2 \left| \sum w_i U_i \right. \) problem due to the presence of two inequalities in system (A) and which have to hold during the search step. Consequently, we provide an extension of the Sort & Search technique by using two tables but one being double indexed.

SortSea first builds a table \( T_1 \) in which each column \( j \) is associated with a partial schedule \( s_1^k \subseteq I_1 \) of at most \( n_1 \) early jobs. There are at most \( 3^{n_1} \) columns since each job \( i \in I_1 \) can be either early on machine 1, early on machine 2 or tardy. To each column \( j \) are associated the values of \( C_{\text{max}}(s_1^j) \), \( C_{\text{min}}(s_1^j) \) and \( \sum_{i \in s_1^j} w_i \). Next, SortSearch algorithm builds two double-entry tables \( T_2^S \) and \( T_2^P \) as follows (figure 9).

Notice that there are \( 3^{n_2} \) partial schedules \( s_2^k \). Now, all values \( -L_{\text{min}}(s_2^k) \) are sorted by increasing values and let \( L_{\text{min}}^{[t]} \) be the \( t \)-th value in this order. Similarly, all values \( -L_{\text{max}}(s_2^k) \) are sorted by increasing values and let \( L_{\text{max}}^{[t]} \) be the \( t \)-th value in this order.

We define initial values inside these two tables as follows, \( \forall t, t' = 1, \ldots, 3^{n_2} \):

\[
\begin{aligned}
T_2^S[t, t'] &= \sum_{i \in s_2^k} w_i \text{ and } T_2^P[t, t'] = s_2^k, \quad \text{if there exists } s_2^k \text{ such that } L_{\text{min}}^{[t]} = -L_{\text{min}}(s_2^k) \text{ and } L_{\text{max}}^{[t']} = -L_{\text{max}}(s_2^k), \\
T_2^S[t, t'] &= +\infty \text{ and } T_2^P[t, t'] = \emptyset, \quad \text{Otherwise.}
\end{aligned}
\]

Fig. 9 Illustration of the initial tables \( T_2^S \) and \( T_2^P \)

Notice that in case there are several partial schedules \( s_2^k \) with the same \( -L_{\text{min}}(s_2^k) \) and \( -L_{\text{max}}(s_2^k) \) values, then we only store the one with the minimal \( \sum_{i \in s_2^k} w_i \) value. SortSea next updates tables \( T_2^S \) and \( T_2^P \) in order to guarantee that \( \forall t, t', T_2^S[t, t'] \) contains the lowest \( \sum_{i \in s_2^k} w_i \) value of a schedule \( s_2^k \) appearing in \( T_2^S[u, v] \), with \( u \geq t \) and \( v \geq t' \). This update is done according to the algorithm given in figure 10. An illustration of the updated tables is given in figure 11.
5.4 The Three Machine Problem with Makespan Minimization

In this section we focus on a scheduling problem involving three identical parallel machines and referred to as $P3||C_{\text{max}}$. This problem, which is similar to the one tackled in section 5.2 can be defined as follows. Consider $n$ jobs to be scheduled without preemption on three identical parallel machines available from time 0 onwards. Each job $i$ is defined by a processing time $p_i$ and completes at time $C_i(s)$ on the machine $j$ which processes it in a given schedule $s$. Without loss of generality, we assume that jobs are indexed such that $p_1 \leq p_2 \leq \ldots \leq p_n$. The aim is to calculate a schedule $s$ (an assignment of jobs on the three machines) which minimizes the makespan defined by $C_{\text{max}} = \max_{1 \leq i \leq n}(C_i)$. This problem has been shown $\mathcal{NP}$-hard in the weak sense.
Consider the algorithm Enum which solves the problem P3||C\text{max} by a brute-force search of all possible schedules. A schedule is defined by a partition of the set of jobs into 3 sets, one for each machine. Therefore, there are at most O(3^n) partitions and the algorithm Enum requires O*(3^n) time. This bound is equal to that of obtained for the more general P|dec|f_{\max} problem. However, we show that it is possible to provide a reduced bound by application of the Sort & Search method.

SortSea, which is very similar to the one proposed for the P2|\sum w_i U_i problem (section 5.3), works as follows. Let I_1 = \{1, \ldots, n_1\} and I_2 = \{n_1 + 1, \ldots, n\} be a decomposition of the instance (we note n_2 = |I_2|). Starting from I_1 it builds an assignment of jobs at the beginning on machine 1, machine 2 and machine 3, whilst from set I_2 it build and assignment of jobs at the end of the schedule. Given a set s_1^i = I_1 (resp. s_3^i = I_3), we refer to s_1^iℓ (resp. s_3^iℓ) as the sub-set of jobs from s_1^i (resp. s_3^i) assigned to machine ℓ (figure 12). We have ∪_{\ell=1}^3 s_1^i\ell = ∪_{\ell=1}^3 s_3^i\ell = \emptyset, ∪_{\ell=1}^3 s_1^i\ell = s_1^i and ∪_{\ell=1}^3 s_3^i\ell = s_3^i. We also define P(A) = ∑_{i∈A} p_i for any given set A, and we have:

\[ C_{\text{max}}(s) = C_{\text{max}}(s_1^i, s_3^i) = \max(P(s_1^i, 1) + P(s_2^i, 1), P(s_1^i, 2) + P(s_2^i, 2), P(s_1^i, 3) + P(s_2^i, 3)). \]

Fig. 12 Decomposition of a schedule for the P3||C\text{max} problem

As the three machines are identical, without loss of optimality, SortSea restricts to the schedules s in which C_{\text{max}}(s) = P(s_1^1, 1) + P(s_2^1, 1), i.e. the makespan value is given by the jobs scheduled on machine 3. These schedules are characterized by the following inequalities:

\[
\begin{align*}
\{ & P(s_1^1, 1) + P(s_2^1, 1) \leq P(s_1^3, 1) + P(s_2^3, 1) \\
& P(s_1^1, 2) + P(s_2^1, 2) \leq P(s_1^3, 1) + P(s_2^3, 1) \}
\end{align*}
\]

\[
\begin{align*}
\Leftrightarrow \{ & \delta_1,3(s_1^1) \geq -\delta_1,3(s_1^1) \\
& \delta_2,3(s_1^1) \geq -\delta_2,3(s_1^1) \} \quad (A)
\end{align*}
\]

with \( \delta_{\alpha,\beta}(s_1^1) = P(s_\alpha, \beta) - P(s_\beta, \alpha) \).

By using \( \delta_{\alpha,\beta} \), we can rewrite \( P(s_2^1, 1) = \frac{1}{3}(P(s_2^1, 1) + \delta_{1,3}(s_1^1) + \delta_{2,3}(s_1^1)) \).

**Theorem 6** Let \( s_1^1 \) be a partial schedule of jobs in I_1 on the three machines and let \( O_2(s_1^1) = \{ s_2^1 \subseteq I_2 | \delta_{1,3}(s_1^1) \geq -\delta_{1,3}(s_1^1) \text{ and } \delta_{2,3}(s_1^1) \geq -\delta_{2,3}(s_1^1) \} \) be the set of partial schedules \( s_2^1 \) built from \( I_2 \) such that \( C_{\text{max}}(s_1^1, s_2^1) = P(s_1^1, 1) + P(s_2^1, 1) \). Let be \( s_2^1 \in O_2(s_1^1) \) such that \( \delta_{1,3}(s_2^1) + \delta_{2,3}(s_2^1) = \min_{s_2^1 \in O_2(s_1^1)} (\delta_{1,3}(s_2^1) + \delta_{2,3}(s_2^1)) \) for any given \( s_1^1 \). We have that \( C_{\text{max}}(s_1^1, s_2^1) \) is minimal among all schedules starting with \( s_1^1 \).
Proof As \( s_2^k \in O_2(s_1^k) \), for a given \( s_1^k \), the constraints of system (A) are answered and the schedule \( s \) obtained by appending \( s_2^k \) after \( s_1^k \) is such that \( C_{\max}(s) = P(s_1^{k,3}) + P(s_2^{k,3}) \).

We now have to show that \( C_{\max}(s) \) is minimal. Using the rewritten form of \( P(s_1^{k,3}) \) given above, we can write that:

\[
C_{\max}(s) = P(s_1^{k,3}) + \frac{1}{3}(P(s_2^k) + \delta_{1,3}(s_2^k) + \delta_{2,3}(s_2^k)).
\]

As \( P(s_2^k) \) is a constant and \( P(s_1^{k,3}) \) is fixed, \( C_{\max}(s) \) is minimal iff \( \delta_{1,3}(s_2^k) + \delta_{2,3}(s_2^k) \) is minimal. This is the case as we have chosen \( s_2^k \) such that \( \delta_{1,3}(s_2^k) + \delta_{2,3}(s_2^k) = \min_{s_2^k \in O_2(s_1^k)} \{\delta_{1,3}(s_2^k) + \delta_{2,3}(s_2^k)\}. \]

SorSea for the \( P3||C_{max} \) problem follows the same scheme than the one proposed for the \( P2|d_i| \sum w_i U_i \) problem on a 2-table approach but with one table being double indexed.

First, it builds a table \( T_1 \) in which each column \( j \) is associated with a partial schedule \( s_1^j \) of jobs in \( I_1 \) and there are at most \( 3^n \) columns. To each column \( j \) are associated the values of \( \delta_{1,3}(s_1^j) \) and \( \delta_{2,3}(s_1^j) \). Next, SorSeach algorithm builds two double-entry tables \( T_2^S \) and \( T_2^P \) as for the \( P2|d_i| \sum w_i U_i \) problem except that:

1. Rows are sorted by increasing values of \( \delta_{1,3}(s_2^k) \) and let \( \delta_{1,3}(t) \) be the \( t \)-th value in this order,
2. Columns are sorted by increasing values of \( \delta_{2,3}(s_2^k) \) and let \( \delta_{2,3}(t) \) be the \( t \)-th value in this order,
3. Each cell of table \( T_2^S \) contains a \( (\delta_{1,3} + \delta_{2,3}') \) value if there exists \( s_2^k \) such that \( \delta_{1,3}(s_2^k) = \delta_{1,3}(s_2^k) \) and \( \delta_{2,3}(s_2^k) = \delta_{2,3}(s_2^k) \).

SorSea next updates tables \( T_2^S \) and \( T_2^P \) in order to guarantee that \( \forall t, t', T_2^S[t, t'] \) contains the lowest \( (\delta_{1,3} + \delta_{2,3}') \) value of a schedule \( s_2^k \) appearing in \( T_2^P[u, v] \), with \( u \geq t \) and \( v \geq t' \). This update is done according to the same algorithm than the one for the \( P2|d_i| \sum w_i U_i \) problem given in figure 10.

To find an optimal solution for the \( P3||C_{\max} \) problem, SorSeach calculates, for each column \( j \) of table \( T_1 \), \( C_{\max}(s_1^j, T_2^P[t, t']) = P(s_1^j) + \frac{1}{3}(P(s_2^k) + T_2^P[t, t']) \) with \( t \) and \( t' \) the lowest indexes such that \( \delta_{1,3}(s_2^k) \geq -\delta_{1,3}(s_1^j) \) and \( \delta_{2,3}(s_2^k) \geq -\delta_{2,3}(s_1^j) \). The smallest \( C_{\max} \) value found among all columns of \( T_1 \) is the optimal \( C_{\max} \) value.

Theorem 7 SorSeach solves the \( P3||C_{\max} \) problem with a worst-case time and space complexity in \( O^*(\sqrt{3}^n) \approx O^*(2.0801^n) \).

Proof Similar to that of theorem 5. \( \square \)

5.5 The Four Machine Problem with Makespan Minimization

In this section we focus on a scheduling problem involving four identical parallel machines and referred to as \( P4||C_{\max} \). This problem, which is similar to the one tackled in section 5.4, can be defined as follows. Consider \( n \) jobs to be scheduled without pre-emption on four identical parallel machines available from time 0 onwards. Each job \( i \) is defined by a processing time \( p_i \) and completes at time \( C_i(s) \) on the machine \( j \)
which processes it in a given schedule \( s \). Without loss of generality, we assume that jobs are indexed such that \( p_1 \leq p_2 \leq \ldots \leq p_n \). The aim is to calculate a schedule \( s \) (an assignment of jobs on the four machines) which minimizes the makespan \( C_{\text{max}} \). This problem has been shown \( \mathcal{NP} \)-hard in the ordinary sense (Lenstra et al. [1977]).

Consider the algorithm \( \text{Enum} \) which solves the problem \( P_4||C_{\text{max}} \) by a brute-force search of all possible schedules. A schedule is defined by a partition of the set of jobs into 4 sets, one for each machine. Therefore, there are at most \( O(4^n) \) partitions and \( \text{Enum} \) requires \( O^*(4^n) \) time. This bound is worse to that of obtained for the more general \( P|\text{dec}|f_{\text{max}} \) problem and we show that it is possible to provide a reduced bound by application of a dedicated decomposition algorithm, referred to as \( \text{DecTS} \).

It is based on a dichotomic decomposition of the problem: let \( M_1 \) be the set of machines 1 and 2, and \( M_2 \) be the set of machines 3 and 4. The \( \text{DecTS} \) algorithm solves the two 2-machine problems by enumerating all possible assignments of the \( n \) jobs on these two sets of machines.

**Theorem 8** \( \text{DecTS} \) solves the \( P_4||C_{\text{max}} \) problem with a worst-case time and space complexity in \( O^*((1 + \sqrt{2})^n) \).

**Proof** \( \text{DecTS} \) makes an extensive use of the \( \text{SorSea} \) algorithm proposed in section 5.2 for the \( P_2||C_{\text{max}} \) problem which requires \( O^*(\sqrt{2}^n) \) time and space in the worst case. As there are \( \sum_{k=0}^{n} \binom{n}{k} \) assignments of jobs on sets \( M_1 \) and \( M_2 \), each requiring to run \( \text{SorSea} \) algorithm, the overall worst-case time complexity of \( \text{DecTS} \) algorithm is in:

\[
O^*(\sum_{k=0}^{n} \binom{n}{k}(\sqrt{2}^k + \sqrt{2}^{n-k})).
\]

By using the Newton’s binomial formula, the above complexity can be rewritten as \( O^*((1 + \sqrt{2})^n) \).

The dichotomic decomposition over the set of machines used in \( \text{DecTS} \) can be generalized to the \( P|\text{dec}|f_{\text{max}} \) problem in a recursive way. This leads to a worst-case time complexity in \( O^*((\sqrt{2} + \lceil \log_2(m) \rceil - 1)^n) \). Unfortunately, as far as \( m \geq 5 \), this bound is worse than the bound of \( O^*(3^n) \) obtained on the more general \( P|\text{dec}|f_{\text{max}} \) problem.

6 A Flowshop Scheduling Problem

In this section we consider an intriguing particular 2-machine flowshop scheduling problem, referred to as \( F_2||C_{\text{max}}^k \) and defined as follows. Consider \( n \) jobs to be scheduled without preemption on two machines and all of them have first to be processed on machine 1 before being processed by machine 2. Each job \( i \) is defined by a processing time on machine \( j \), denoted by \( p_{i,j} \) and let \( 1 \leq k \leq n \) be a given value. The aim is to sequence jobs in order to minimize the makespan value of the \( k \)-th job in the schedule, referred to as \( C_{\text{max}}^k \). Clearly, if \( k = n \), the problem is polynomially solvable as it is exactly the \( F_2||C_{\text{max}} \) problem solved by the so-called Johnson’s algorithm (Johnson [1954]). However, for any arbitrary value \( k \), the \( F_2||C_{\text{max}}^k \) problem can be shown to be \( \mathcal{NP} \)-hard in the weak sense (T’kindt et al. [2007]). This problem can be nicely reformulated as a scheduling problem with common due date assignment and minimization...
of the number of late jobs, referred to as $F2|d_i = d, \ d\ unknown, \sum U_i = \epsilon |d$ with $\epsilon = n - k$. Then, all jobs are assumed to share a common due date which value has to be minimized under the condition that exactly $(n - k)$ jobs complete after this due date. This reformulation facilitates the presentation of exponential algorithms and will be used hereafter.

First, consider the Enum algorithm which solves the $F2|d_i = d, \ d\ unknown, \sum U_i = \epsilon |d$ problem by a brute-force search of all possible schedules. For each job we have either to decide whether it is early or late, thus leading to a set of $2^n$ solutions, each of these ones having a value of the common due date $d$ equal to the makespan of the early jobs (the late jobs are scheduled after the early jobs). Therefore, Enum has a worst-case time complexity in $O^*(2^n)$. We now provide two exponential-time algorithms with improved worst-case complexities. The first one, referred to as BraRed, is an application of the Branch & Reduce method, whilst the second one, referred to as SorSea, is an application of the Sort & Search method.

BraRed calculates an optimal solution by exploring a binary search tree: for each node, two child nodes are created by assigning a job $i$ early, and by assigning it late. Besides, each node such that the number of late jobs exceed the value of $\epsilon$ is pruned. Let us refer to $T(n, \epsilon)$ as the time complexity of BraRed to solve the problem with $n$ jobs among which $\epsilon$ are late. Due to the branching scheme, we have:

$$T(n, \epsilon) = T(n - 1, \epsilon) + T(n - 1, \epsilon - 1) = \left(\begin{array}{c} n \\
\epsilon \end{array}\right).$$

Due to the problem definition, we can assume that $\epsilon = \lambda n$ with $\lambda \in [0; 1]$ and we state the following result.

**Theorem 9** BraRed solves the problem with a worst-case time complexity in $O^*((\frac{1}{2})^n(\frac{1}{1-\lambda})^{1-\lambda}n)$, i.e. $O^*(c(\lambda)^n)$ with $c(\lambda) = (\frac{1}{\lambda})^n(\frac{1}{1-\lambda})^{1-\lambda}$, and polynomial space.

**Proof** This result can be shown by using the well-known Stirling’s formula which enables to approximate $k!$ by $(\frac{k}{e})^k \sqrt{2 \pi k}$. We have:

$$\frac{n!}{e^{(n-\epsilon)}} \sim \frac{\sqrt{2\pi n}}{(\frac{e}{2})^{\frac{n}{2}}} \left(\frac{\lambda}{1-\lambda}\right)^{\lambda n} \left(\frac{1}{1-\lambda}\right)^{1-\lambda n} \left(\frac{\lambda}{2}\right)^{\frac{1}{2}} \left(\frac{\lambda}{2}\right)^{\frac{1}{2}} \left(\frac{\lambda}{2}\right)^{\frac{1}{2}} \left(\frac{\lambda}{2}\right)^{\frac{1}{2}}.$$

Therefore, the worst-case time complexity is in $O^*(c(\lambda)^n)$ with $c(\lambda) = (\frac{1}{\lambda})^n(\frac{1}{1-\lambda})^{1-\lambda}$.

In table 2 we provide the worst-case bounds for different values of $\lambda$. The function $c(\lambda)$ is symmetric around $\lambda = 0.5$ which implies that the values of $c(\lambda)$, for $\lambda > 0.5$, can be deduced from that table.

Notice that whatever the value of $\lambda$, BraRed has a lower worst-case case time complexity than Enum, and both require polynomial space to run. At last, BraRed has the particularity to use only a branching scheme but no reduction rules, as usual in a Branch & Reduce method. Despite our efforts, we have not been able to find reduction rules useful to decrease the worst-case time complexity; the available dominance conditions for the $F2|d_i = d, \ d\ unknown, \sum U_i = \epsilon |d$ problem (T’kindt et al [2007]) can always be made ineffective on pathological instances.

We now turn to the SorSea which we show to be more effective than BraRed algorithm for most of the values of $\epsilon$. We first focus on properties of the problem. It is
well-known that, given a set of jobs $E$, the optimal makespan is given in $O(|E| \log(|E|))$ time by the so-called Johnson’s algorithm (Johnson [1954]). Besides, it can be easily shown (e.g., T’kindt et al. [2007]) that, if $s_E$ denotes the schedule obtained by applying Johnson’s algorithm on set $E$, for any $E' \subseteq E$, $s_{E'}$ can be obtained by removing from $s_E$ the jobs in $E \setminus E'$. So, without loss of generality, we assume in the remainder that all jobs are numbered according to Johnson’s order, i.e. their position in the schedule given by the Johnson’s algorithm.

Let be $P_1(s) = \sum_{i \in s} p_i,1$ and $P_2(s) = \sum_{i \in s} p_i,2$. We have the following general result.

**Lemma 7** Let $s_1$ and $s_2$ be two partial sequences of jobs and $s = s_1/s_2$ is assumed to be sorted according to Johnson’s order. We have $C_{\text{max}}(s) = \max(P_1(s_1) + C_{\text{max}}(s_2)); C_{\text{max}}(s_1) + P_2(s_2))$.

**Proof** Let $n_1$ be the number of jobs in sequence $s_1$. Without loss of generality, we can renumber the jobs in $s_1$ from 1 to $n_1$ and jobs in $s_2$ from $n_1 + 1$ to $n$, in their order of apparition in the two sequences.

We have:

$$C_{\text{max}}(s) = \max_{1 \leq u \leq n} \left( \sum_{i=1}^{u} p_{i,1} + \sum_{i=u}^{n} p_{i,2} \right)$$

$$C_{\text{max}}(s) = \max \left( \max_{1 \leq u \leq n_1} \left( \sum_{i=1}^{u} p_{i,1} + \sum_{i=u}^{n} p_{i,2} \right); \right.$$  

$$\left. \max_{n_1 + 1 \leq u \leq n} \left( \sum_{i=1}^{u} p_{i,1} + \sum_{i=u}^{n} p_{i,2} \right) \right)$$

$$C_{\text{max}}(s) = \max \left( \max_{1 \leq u \leq n_1} \left( \sum_{i=1}^{u} p_{i,1} + \sum_{i=n_1+1}^{n} p_{i,2} + P_2(s_2); \right. \right.$$  

$$\left. P_1(s_1) + \max_{n_1 + 1 \leq u \leq n} \left( \sum_{i=n_1+1}^{u} p_{i,1} + \sum_{i=u}^{n} p_{i,2} \right) \right)$$

$$C_{\text{max}}(s) = \max \left( C_{\text{max}}(s_1) + P_2(s_2); P_1(s_1) + C_{\text{max}}(s_2) \right) \square$$

Let be $I_1 = \{1, \ldots, \lfloor n/2 \rfloor\}$ and $I_2 = \{ \lfloor n/2 \rfloor + 1, \ldots, n\}$ a partition into two jobs sets of the initial instance to solve. Starting from set $I_1$, SorSea builds a sequence $s_1^k$ of $(n - \epsilon_1)$ early jobs scheduled first, whilst starting from set $I_2$ it builds a sequence $s_2^k$ of $(n - \epsilon_2)$ early jobs scheduled right after the early jobs of $I_1$, with $\epsilon_1 + \epsilon_2 = \epsilon$ (figure 13). The sequence $s = s_1^k/s_2^k$ of early jobs necessarily follows Johnson’s order and, thus, the value of the common due date can be set to $d = C_{\text{max}}(s)$.

SorSea builds a table $T_1$ in which each column $j$ is associated with a partial schedule of early jobs $s_1^j$ and a partial schedule of $\epsilon_1$ late jobs $s_1^j$. There are at most $2^k$ columns since each job in $I_1$ can be set either early or late. To each column $j$ are associated

<table>
<thead>
<tr>
<th>$\frac{1}{2}$</th>
<th>$\lambda$</th>
<th>$c(\lambda)$</th>
<th>Worst-case bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.50</td>
<td>2</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>6</td>
<td>0.53</td>
<td>1.8898</td>
<td>$O(1.3899^n)$</td>
</tr>
<tr>
<td>4</td>
<td>0.25</td>
<td>1.7547</td>
<td>$O(1.7547^n)$</td>
</tr>
<tr>
<td>5</td>
<td>0.20</td>
<td>1.6493</td>
<td>$O(1.6493^n)$</td>
</tr>
<tr>
<td>6</td>
<td>0.16</td>
<td>1.5691</td>
<td>$O(1.5691^n)$</td>
</tr>
<tr>
<td>8</td>
<td>0.14</td>
<td>1.5069</td>
<td>$O(1.5069^n)$</td>
</tr>
<tr>
<td>10</td>
<td>0.12</td>
<td>1.4575</td>
<td>$O(1.4575^n)$</td>
</tr>
<tr>
<td>11</td>
<td>0.11</td>
<td>1.4174</td>
<td>$O(1.4174^n)$</td>
</tr>
<tr>
<td>10</td>
<td>0.10</td>
<td>1.3841</td>
<td>$O(1.3841^n)$</td>
</tr>
</tbody>
</table>

**Table 2** Worst-case bounds of BraRed algorithm for different values of $\lambda$
the values of $P_1(s_1^k),P_2(s_1^k)$ and $C_{\text{max}}(s_1^k)$. Next, SorSea builds a table $T_2$ in which each column $k$ is associated with a partial schedule of early jobs $s_1^k$ and a partial schedule of $\epsilon_2$ late jobs. As for table $T_1$, there are at most $2^\frac{n}{2}$ columns, which are in table $T_2$ sorted by non decreasing value of $(C_{\text{max}}(s_1^k) - P_2(s_1^k))$. To each column $k$ is associated the values of $P_1(s_1^k),P_2(s_1^k),C_{\text{max}}(s_1^k)$, $C_{\text{max}}(s_1^k) = \min_{i \geq k} C_{\text{max}}(s_1^i)$ and $P_2^{\text{min}}(s_1^k) = \min_{k \leq i} P_2(s_1^i)$.

For a given column $j$ of $T_1$, i.e. partial schedules $s_1^j$ and $\bar{s}_1^j$, SorSea searches in $T_2$ the indexes $k$ and $\ell$:

$$k = \arg\min(u \in T_2 \mid C_{\text{max}}(s_1^u) - P_2(s_1^u) \geq C_{\text{max}}(s_1^j) - P_1(s_1^j)),$$

$$\ell = \arg\max(u \in T_2 \mid C_{\text{max}}(s_1^u) - P_2(s_1^u) \leq C_{\text{max}}(s_1^j) - P_1(s_1^j)).$$

Notice that $\ell$ is either equal to $k$ or $(k - 1)$. Then, SorSea deduces the smallest value of the common due date $d(s_1^j)$ in a schedule of $\epsilon_2$ late jobs starting by $s_1^j$ as follows:

$$d(s_1^j) = \min(P_1(s_1^j) + C_{\text{max}}^{\text{min}}(s_1^k), C_{\text{max}}(s_1^j) + P_2^{\text{min}}(s_1^k)).$$

The optimal value of the common due date $d$ is obtained by applying the above search into table $T_2$ for each column $j$ from table $T_1$ and by keeping the smallest value $d(s_1^j)$ found.

**Theorem 10** SorSea solves the F2|d_1 = d, d unknown, \(\sum U_i = \epsilon|d\) problem with a worst-case time and space complexity in \(O^*(\sqrt{n})\).

**Proof** First, SorSea builds table $T_1$, thus requiring $O^*(\sqrt{n})$ time and space. Next, it builds table $T_2$ also in $O^*(\sqrt{n})$ time and space since the sorting of the columns is done in $O^*(2^\frac{n}{2} \times \log(2^\frac{n}{2})) = O^*(\sqrt{n})$ time. The main part of SorSea consists in searching for each column $j$ of $T_1$ the columns $k$ and $\ell$ such that $k = \arg\min(u \in T_2/C_{\text{max}}(s_1^u) - P_2(s_1^u) \geq C_{\text{max}}(s_1^j) - P_1(s_1^j))$ and $\ell = \arg\max(u \in T_2/C_{\text{max}}(s_1^u) - P_2(s_1^u) \leq C_{\text{max}}(s_1^j) - P_1(s_1^j))$. By a binary search, whenever $j$ is given, the values of $k$ and $\ell$ can be computed in $O^*(\log(2^\frac{n}{2})) = O(n)$ time. As there are $2^\frac{n}{2}$ columns in table $T_1$, the search for the optimal solution in tables $T_1$ and $T_2$ can be achieved in $O^*(\sqrt{n})$ time and space. \(\square\)

Now, we can establish which algorithm has a lower worst-case time bound among SorSea and BraRed. It is clear that in terms of space requirement, BraRed outperforms SorSea since it requires polynomial space in the worst-case.

**Lemma 8** SorSea has a lower worst-case time complexity than BraRed for any value $\frac{n}{2} \in [0.110027; 0.889973]$.

**Proof** The worst-case time bound of SorSea algorithm is equal to $\sqrt{n}$ whilst that of BraRed is equal to $c(\lambda)n$ with $c(\lambda) = (\frac{1}{2})^\lambda(\frac{1}{1-\lambda})^{1-\lambda}$ (theorem 9). The values of $\lambda = \frac{n}{2}$ such that $c(\lambda) < \sqrt{n}$ can be computed by means of a mathematical software like SCILAB (SCILAB [2011]), thus leading to the given result. \(\square\)
Figure 14 presents a summary of the worst-case time bounds for \textit{Enum}, \textit{SorSea} and \textit{BraRed}: the hardest problems for which \textit{BraRed} reaches the complexity of \textit{Enum} are those with $\epsilon = \frac{n}{2}$. It is interesting to notice that the branch-and-bound algorithm proposed by T’kindt et al [2007] for solving the $F2|d_i = d, d$ unknown, $\sum U_i = \epsilon|d$ problem relies on the same branching scheme than \textit{BraRed} algorithm. Therefore, this branch-and-bound algorithm has the same worst-case time bound than \textit{BraRed} (theorem 9).

7 Conclusions and Future Research Lines

In this paper we have investigated the worst-case time and space complexities of some scheduling problems for which we have proposed exact exponential-time algorithms. The study of such algorithms for \textit{NP}-hard optimisation problems has been the matter of recently growing scientific interest, excluding scheduling problems for which almost no exponential-time algorithms were known.

Exact exponential-time algorithms are exact algorithms designed to have an upper bound on their time (and maybe, space) complexity in the worst-case better than a brute-force search. By the way, we establish the property that the related \textit{NP}-hard problems can be solved within at most a known bounded number of steps. This is an important result since we get some information on the difficulty of these problems.

To the best of our knowledge few result were known in scheduling theory. In this paper, we have presented worst-case time complexities for 15 scheduling problems (table 1) including the $1|\text{prec}|f_{\text{max}}, 1|\text{prec}|\sum f_i, P|\text{prec}|f_{\text{max}}$ and $P|\text{prec}|\sum f_i$ problems which cover a large set of basic scheduling problems. For 8 of them the presented complexities are new. The first conclusion that can be derived from this paper, relies on the method used to build exponential-time algorithms. One which applied well is the \textit{Sort & Search} method, leading often to worst-case time and space complexities in $O^*(\sqrt{2^n})$. Surprisingly, the \textit{Branch & Reduce} method which resembles a branch-and-bound algorithm did not enable, for most of the tackled problems, to derive an exponential-time algorithm with a worst-case time complexity better than that of the brute-force search algorithm. This is related to the \textit{reduction rules} used in the \textit{Branch & Reduce} method which are really hard to establish for scheduling problems. Dynamic programming has been also successfully applied to derive complexities. Beyond these, more or less, classic methods we have also derived exponential-time algorithms by proposing dedicated decomposition algorithms, as for the $P|\text{dec}|f_{\text{max}}$ and $P|\text{dec}|\sum f_i$ problems.

The second contribution of this paper relies on the fact that all the proposed exponential-time algorithms, whatever the method applied, are based on specific decomposition schemes of schedules that enable to break down the complexity. The question, now open, is whether it is possible or not to use these decomposition schemes...
in exact algorithms which would be more efficient in practice than known exact algorithms. Notice that the latter do not necessarily have a better worst-case time bound than that of the brute-force search of solutions.

Acknowledgements

The researches presented in this paper have been partially supported by the working group on Operations Research (GDR RO) of the CNRS, which the authors would like to thank.

References

Jackson J (1955) Scheduling a production line to minimize maximum tardiness. Management Science Research Project, University of California (USA), research report 43
Johnson SM (1954) Optimal two and three stage production schedules with set-up time included. Naval Research Logistics Quarterly 1:61–68