International Journal of Software Engineering and Its Applications
Vol. 4, No. 4, October 2010

A Job-Shop Scheduling Model of Software Development Planning for Constraint-based Local Search

Irene Barba, Carmelo Del Valle
Dpt. Lenguajes y Sistemas Informáticos, Universidad de Sevilla
{irenebr,carmelo}@us.es

Abstract

Software development planning usually does not take any advantage of planning and scheduling techniques involving reasoning about time and resources, which are typical in the manufacturing area. This paper proposes a constraint-based model for the Job Shop Scheduling Problem to be solved using local search techniques. The model can be used to represent a multiple software process planning problem when the different (activities of) projects compete for limited staff. The main aspects of the model are: the use of integer variables which represent the relative order of the operations to be scheduled, and two global constraints, alldifferent (all the variable values are forced to be different from the others) and increasing (the variable values are forced to be greater than the previous ones in the sequence), for ensuring feasibility. The AllDifferent and Increasing constraints enforce the conjunction of one binary constraint, the not-equal constraint and the greater constraint respectively, for every pair of variables. By analyzing the set of all relations at the same time, both global constraints offer greater filtering power, enhancing the constraint model and the efficiency in the resolution of the problem. An interesting property of the model is that cycle detection in the schedules is implicit in the satisfaction of the constraints. In order to test the proposed model, a parameterized local search algorithm has been used, with a neighborhood similar to the Nowicki and Smutnicki one, which has been adapted in order to be suitable for the proposed model.

Keywords: *job shop scheduling, local search, constraint satisfaction problems, software development processes.*

1. Introduction

Software development has been modeled using a wide range of approaches. They vary according to the focus of the analysis and they address successfully the whole development process depending on how it is carried out. Many of the software management tools use temporal information and ignore in some ways the resources to be used, considering them unlimited, since they are based on PERT and CPM analysis. These may not be adequate in different situations, for example when smaller multiple projects are developed and projects compete for limited staff [13].

Business Process Management (BPM) [21] includes methods, techniques, and tools to support the design, enactment, management, and analysis of operational business processes. Nowadays, there exists an increasing interest on the part of the organization in managing its business processes since they need to adapt to the new commercial conditions, as well as to respond to competitive pressures, considering the business environment and the evaluation of their information systems. The software development can be modeled as a business process, so BPM tools seem to be a good approach to deal with all the aspects that entail the software development process, including the software development planning.

The area of Scheduling [2] includes problems in which it is necessary to determine an execution plan for a set of tasks related by precedence constraints. The execution of each task requires the use of one or more resources so that the tasks may compete for limited resources. In general, the objective is to find a feasible plan so that both precedence and resource constraints are satisfied, optimizing an objective function related to, in most cases, temporal measures, as the completion time of the last executed task (makespan) or the total weighted tardiness. In some projects, resource allocation patterns [17] are defined to express elaborate and fine-grained authorization control, such as separation of duty, history-based, case handling, and role-based resource allocation patterns.

The current work is focus on the Job Shop Scheduling Problem (JJSP)[10], which is a specific problem of Scheduling where the tasks are grouped by jobs that establish precedence relations among them. Another particularity of the JJSP is that the execution of each task only requires one resource. Besides the job shop approach, there are several scheduling models, such as: flow shop[6], where each job contains exactly one operation for every machine and all jobs go through all the machines in the same order; open shop [5], where there are not ordering constraints on tasks; the Resource-Constrained Project Scheduling Problem (RCPSP) [12], where there are a set of renewable resources, such as machines or personnel, which are available at any time in limited numbers of unit and each task, in general, needs several resources to be executed.

Other interesting issues deal with modeling tasks and resources. In the first case, there exists the multi-mode project scheduling approach, where each task can be executed in several operation modes, each one requiring, in general, different resources with different durations or costs. On the other hand, multi-capacity resources allow to model that tasks would require some quantity of resources without distinguishing between them.

Different scheduling approaches can be used to model and represent different kinds of software development process. Specifically, a job shop approach, traditional in manufacturing, may represent an important aid for the Software Development Planning since it can manage the interactions between projects and resources in a natural way and enables to consider minimizing different goals, as development time (makespan) and cost, while satisfying all the temporal and resource constraints. With this aim, a job shop scheduling model is presented in this paper, so that it can represent a multiple software project to be planned. The equivalence of terms used from both areas is in such a way that jobs correspond to single software projects, and resources can represent each person or software development team working in the projects.

In a wider perspective, in planning problems, related to scheduling ones, the tasks to be executed are not established a priori, so it is necessary to select and to order a set of tasks from a set of alternatives, in order to achieve an objective defined in advance. Some of the extensions to scheduling, such as alternative resources and process alternatives, lead to models that are closer to planning [18]. Also, the AI planning community has done several efforts to extend classical planning techniques to treat resources and time constraints. Currently, there is an increasing interest in integrating the application of Artificial Intelligence (AI) Planning and Scheduling (P&S) techniques since real-world problems involve both of them [1]. It includes the effective application of AI P&S techniques to the production and execution of models of organizations (business process management). There are several points where AI P&S tools can be effectively applied to the business process management.

Business Process Management (BPM) and AI Planning and Scheduling are two disciplines with many parallels, but which have largely been pursued by disjoint communities. Currently, there is a growing interest in the application of AI Planning and Scheduling techniques to automate tasks of real world problems, such as the production and execution of models of organization.

There are many views of the BPM Life Cycle, one of them [21] can be represented by Figure 1.

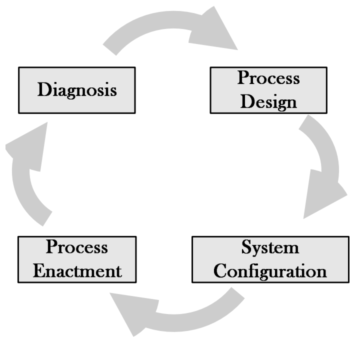


Figure 1. BPM life cycle

As follows, the different stages of BPM Life Cycle are presented, relating them to AI Planning and Scheduling applications:

- Process Design: This stage involves designing, modeling, simulating, etc. the organization processes. It is basically a human activity, though supported by computer-based tools to record and display the process model, run simulations, etc. In [15], the user can introduce the knowledge in a workflow modeling tool. After that, real models can be automatically generated using a planner that integrates planning and scheduling.
- System Configuration: In the process design phase, only a template of a BP is elaborated and, usually, this template does not contain many details that are necessary for the execution phase. In the system configuration phase, first, the customer must specify information about the product or service required. Secondly, a schedule must be generated, taking into account the information given by the customer, such as the target end date, dependencies between tasks or task durations. Then, the resources are assigned to tasks for the appropriate time slots (scheduling), considering the finite capacity and/or non-sharable resources and, generally, taking into account the optimization of one or more objective functions, such as process duration, robustness, cost, etc. The resulting plan may contain alternative branches that are pruned as information is gathered and decisions made during enactment.
- Process Enactment: In this phase, the previously obtained plan is carried out. The activities have to be coordinated to ensure correct sequencing and that compatible variants of the activities are performed. At the same time, resources will be involved in enacting multiple processes and instances of the same process. When the same resource is required by several tasks at the same time, generally, rules for prioritizing tasks must act on the conflict. Due to the finite capacity of the resources, different processes can interfere with each other.
- Diagnosis: As execution proceeds, the enactment information must be analyzed due to the possible appearance of unexpected incidents, such us unavailability of resources, real task durations different from expected ones, fails, etc. Minor incidents may require updating of the plan. More significant differences may require great

changes in the plan, even a re-planning, considering, in general, the optimization of one or more objective functions. On the other hand, the execution information of a business process can be very useful to improve many aspects, such as the detection of bottlenecks and potential fraudulent loopholes in the business process.

In this work, a model and a resolution method for the Job Shop Scheduling Problem are proposed, and they can be applied, between others, to the System Configuration phase of the business process management. In future works, the current approach is intended to be generalized to other scheduling and planning problems, so that they can be applied in the other cycle phases.

On the other hand, Constraint Programming (CP) has evolved in the last decade to a mature field due to, among others, the use of different generic and interchangeable procedures for inference and search, which can be used for solving different types of problems [3, 16]. Although a separation of models and algorithms is desirable for reusability issues, there is an influence between them that must be taken into account when a good behavior of the whole resulting method is pursued. Most models that have been used in CP have been tested using complete algorithms, and they are not equally suitable for other algorithmic approaches such as local search [23]. Constraint-based techniques have been used successfully to solve a wide scope of applications related to scheduling and planning problems.

This paper proposes a Constraint Satisfaction Problem (CSP) model for the Job Shop Scheduling Problem (JSSP) to be solved using local search techniques, that is, it defines the variables which determine a solution, the related constraints of the problem involving those variables, and some possible neighborhoods. The problem has been solved by different authors using local search [10, 14, 22], but the novelty consist of the proposed model, based on including the ordering of the operations directly in the variables and constraints of the CSP, so that further definitions and developments of the main components of local search algorithms would take advantage of this representation.

For such techniques, a very important issue is the defined neighborhood, that is, the set of candidates to which the walk may continue from the current solution. For JSSP, one of the best methods was proposed by Nowicki and Smutnicki [14], whose neighborhood was more constrained than other previous approaches. An adaptation and an extension of this neighborhood are proposed in this work, in order to be suitable for the defined CSP model.

The rest of the paper is organized as follows. First, Section 2 presents a formulation of the JSSP. After that, Section 3 includes the main ideas of constraint programming, including local search algorithms. Then, Section 4 describes the proposed model. Next, in Section 5 some experimental results are shown and analyzed. Finally, Section 6 presents some conclusions and future work.

2. Problem Definition

The Job Shop Scheduling Problem [2, 10] may be formulated as follows. There exists a set of n jobs $J_1,\ldots, J_n$ and a set of m machines $M_1,\ldots, M_m$. Each job J_i consists of a sequence of n_i operations $op_{i1},\ldots, op_{i,ni}$, which must be processed in this order. Each operation op_{ij} must be processed for p_{ij} time units, without preemption, on machine $\mu_{ij} \in \{M_1,...,M_m\}$. Each machine can only process one operation at a time. So, two types of constraints are defined, the precedence constraints among the operations of each job, and the resource constraints which force to select a permutation order of the operations that use each machine. These last constraints are the source of the NP-hard complexity of JSSP [6].

The typical objective, used in this work, is to find a feasible solution, minimizing the makespan, $C_{\max} = \max_{i=1..n}\{C_i\}$, where C_i is the completion time of job J_i, i.e. the completion time of $op_{i,ni}$.

Figure 2 shows the disjunctive graph representation for a simple example of the problem, with $n = 3$ and $n_i = 3, \forall i$. In a disjunctive graph $G = (V,C,D)$, there is a set V of nodes which correspond to the operations of the job-shop, a set C of directed arcs corresponding to the precedence constraints, and a set D of undirected arcs which connect the operations that use the same machine. A solution to the problem consists of fixing a direction for the undirected arcs, being feasible if there are no cycles.

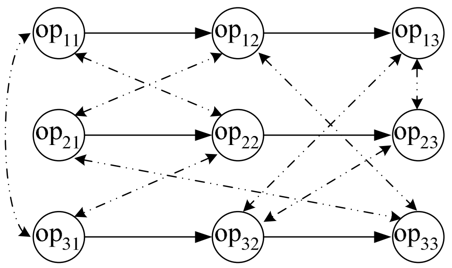


Figure 2. A disjunctive graph for a job shop problem

3. Constraint Programming

Constraint Programming (CP) has been evolved in the last decade to a mature field because, among others, of the use of different generic and interchangeable procedures for inference and search, which can be used for solving different types of problems [16]. On the other hand, constraint-based techniques have been used successfully to solve a wide scope of applications related to scheduling and planning problems, including the JJSP. A Constraint Satisfaction Problem (CSP) is defined by a set of variables, the set of domains of values for each variable and a set of constraints. Each constraint involves some variables and specifies the allowable combinations of values for them. A solution is defined by an assignment of values to all the variables, being feasible if it does not violate any constraint. Constraint Optimization Problems (COPs) require a solution that optimizes an objective function.

There exists a wide scope of mechanisms used to solve CSPs and COPs, which can be classified as search or consistency algorithms [16]. Search algorithms are based on the exploration of the solution space to find a solution or to prove that there is no solution. It is possible to differentiate between systematic algorithms and local search algorithms: systematic algorithms generally explore a search tree which is based on the possible values for each of the variables of the CSP problem. On the other hand, local search algorithms, in general, perform an incomplete exploration of the search space by repairing infeasible complete assignments or trying to improve the objective value. On the other hand, Consistency algorithms consist on removing inconsistent values from the domain variables. One way to accomplish this is evolving from the initial problem towards equivalent problems whose solution space is smaller, so it is easier to solve. Once a problem is modeled by a CSP, a generic or specialized CSP solver can be used in order to obtain the required solution.

Most solving algorithms for CSPs proposed in the CP area are complete, and lastly local search is being considered as promising for solving large instances of complex problems [23], where complete algorithms fail. The constraints from the CSP model may be used for guarantying feasibility of the solutions explored, or even using their possible (degree of) violation as a guide for the search. Most of ideas associated to local search algorithms in other areas can be used for solving CSPs, or in our case, for COPs.

Local search algorithms move iteratively through the set of feasible solutions. For those movements, a neighborhood for the current solution is determined in each iteration as a set of the solutions that can be selected as the next solution, and that can be obtained from the current solution with small changes. Depending on the method of choosing the next solution from neighborhood and the criteria for stopping the iterative sequence of movements, different algorithms can be defined [8]. In order to test the proposed model, a basic tabu search algorithm [7] has been used, containing the main components that have been proved useful in local search, as described in Section 4.4.

4. Our Proposal

4.1. The CSP Model

As stated before, a CSP is defined by a set of variables *V*, a set of domains of values for each variable *D* and a set of constraints that involve the variables *C*. Typical CSP models for the JSSP state the start times st_{ij} of the operations op_{ij} as the variables of the CSP [4], and the constraints are divided in two groups, precedence constraints ($st_{ij} + p_{ij} \leq st_{i,j+1}$) and resource constraints ($st_{ij} + p_{ij} \leq st_{kl} \vee st_{kl} + p_{kl} \leq st_{ij}$, op_{ij} and op_{kl} using the same machine). Our proposed CSP model is based on using the CSP variables to establish the execution order of the operations of the JSSP, resulting in a simple model.

Let Π_J be a JSSP with a set *J* of *n* jobs, a set *M* of *m* machines, and a set *O* of *#ops* operations. The proposed model has the following components:

- Each operation op_{ij} is represented as an integer variable of the CSP v_{ij}, therefore the set of variables is $V = \left\{ v_{ij}, 1 \leq i \leq n, 1 \leq j \leq n_i \right\}$.
- The domain of each variable v_{ij} is $D(v_{ij}) = [1..\#ops]$, $\forall v_{ij} \in V$.
- The set *C* of constraints contains two types of items:
    1. **Precedence Constraints**: The value of each variable v_{ij} has to be less than the value of all the variables corresponding to the following operations in the same job: $v_{ij} < v_{ik}$, $\forall v_{ij}, v_{ik}$ such that $j < k$. In order to improve the efficiency and to obtain a clearer model, a new constraint (*increasing*) has been used between the operations of each job. It is defined on a sequence of variables $\{v_1, v_2, \ldots, v_n\}$ and it is equivalent to the satisfaction of the conditions $v_1 < v_2 < \ldots < v_n$.
    2. **Resource Constraints**: In order to satisfy that each machine can process only one operation at the same time, all the variable values are forced to be different from the others (*alldifferent* constraint is used), i.e., each solution is a permutation of the set *{1, 2, ..., #ops}*.

An interesting property of the model, using the *increasing* and *alldifferent* constraints, is that cycle detection in the disjunctive graph is implicit in the satisfaction of the constraints, so no solution of the CSP will contain cycles.

A solution for the constrained problem, in which a value for each CSP variable is given, is a permutation of *1..#ops* variables and can be represented by an ordered sequence of operations *S*. With this sequence, an "earliest start schedule" is associated by planning the operations in the order induced by the sequence, resulting in a JSSP solution. *S*(*m*) is denoted as the ordered sequence of operations that are executed on the machine *m* in the order fixed by the solution represented by *S*. Figure 3 shows a solution for the problem of Fig. 2. First, the value for each variable is shown and below is the corresponding solution *S*, where the position *a* in the sequence represents the value of the variable *S*[*a*] ($v_{ij} = a \equiv S[a] = op_{ij}$). Also, the ordered sequences corresponding to each machine are shown. Finally, Fig. 3 shows a JSSP solution where all the arcs in the graph are directed according to the fixed order in *S*. Notice that there can be several solutions of the CSP problem that lead to the same schedule, for example the solution $S = \{op_{21}, op_{31}, op_{32}, op_{11}, op_{12}, op_{13}, op_{22}, op_{33}, op_{23}\}$ for the problem of Fig. 2 leads to the same schedule that the solution shown in Fig. 3.

From now on, *PM*(*v*) and *SM*(*v*) are used to refer to the predecessor and successor variables of *v* on its machine, and similarly *PJ*(*v*) and *SJ*(*v*) on its job. *PM*(*PM*(*v*)) is denoted by $PM_2(v)$ (the same for *SM*(*v*)) and so on. Moreover, *m*(*v*) is denoted as the machine in which the operation corresponding to the variable *v* has to be executed.

$v_{11} = 2$	$v_{12} = 6$	$v_{13} = 8$
$v_{21} = 4$	$v_{22} = 5$	$v_{23} = 9$
$v_{31} = 1$	$v_{32} = 3$	$v_{33} = 7$

S	op_{31}	op_{11}	op_{32}	op_{21}	op_{22}	op_{12}	op_{33}	op_{13}	op_{23}
	1	2	3	4	5	6	7	8	9

$S(m_1)$	op_{31}	op_{11}	op_{22}
$S(m_2)$	op_{21}	op_{12}	op_{33}
$S(m_3)$	op_{32}	op_{13}	op_{23}

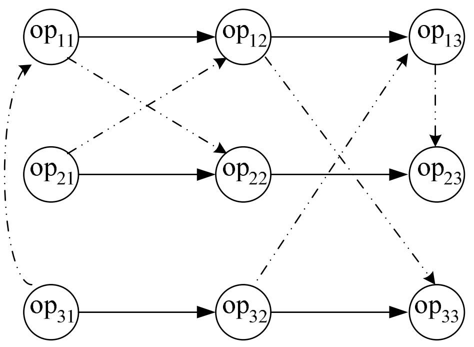


Figure 3. Example of a feasible solution

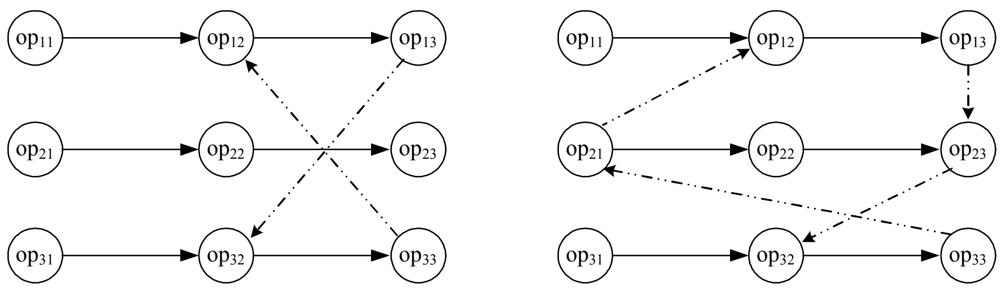
4.2. Cycle Detection

A solution for the problem consists of establishing directions for the undirected arcs in the disjunctive graph (Section 2), being feasible provided that there is no cycles. A cycle for a solution in the disjunctive graph is a closed directed (simple) path, with no repeated vertices other than the starting and ending vertices.

Figure 4 shows two cycles on the disjunctive graph presented in section 2. In Fig. 4.a it can be seen a cycle that contains four operations and involves two jobs, meanwhile Fig. 4.b contains six operations of three jobs.

A cycle can be seen as a sequence of operations that contains two types of edges:

- Precedence edges: are fixed by the problem.
- Resource edges: are given by the decisions made to solve the problem.



(a) Cycle containing 4 operations **(b) Cycle containing 6 operations**

Figure 4. Examples of cycles

All the possible cycles that can be formed in the graph involve, at least, two machines and four operations, two belonging to one job, and two belonging to another job, such as it is shown in the figure 5. In this figure it is possible to see a cycle formed by four operations, two belonging to J_1 (op_{1i} and op_{1j}) and two belonging to J_2 (op_{2k} and op_{2l}). In the sequence of operations appears, at least, two precedence edges, that connect operations using different machines. All the operations that appear in the figure can be executed on different machines, so the machines involved in this cycle can be between 2 and 4. It is important to clarify that op_{1j} and op_{2k} do not have to be executed in the same machine (the same for op_{1i} and op_{2l}).

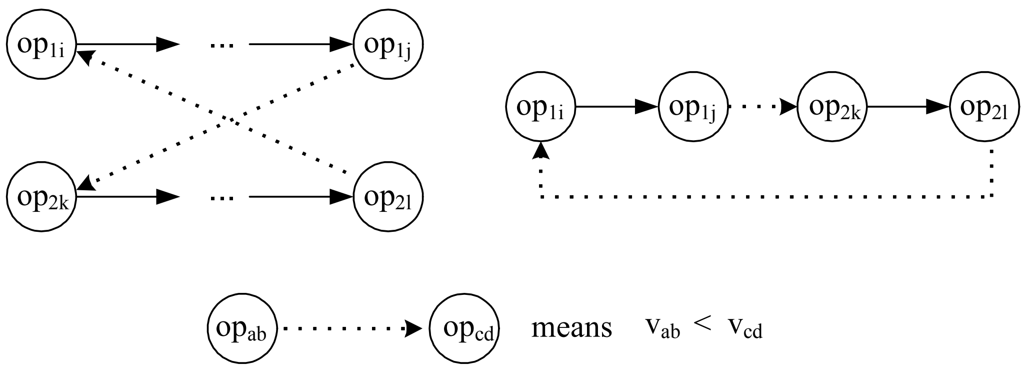


Figure 5. A cycle in a disjunctive graph

Theorem 1. *Any solution of the CSP, with the proposed model, will contain no cycles.*
Proof. Let op_{1i}, op_{1j} , op_{2k} and op_{2l} be four operations with the followings characteristics:

- op_{1i} and op_{1j} belong to J_1, $i < j$, so $v_{1i} < v_{1j}$ (Increasing constraint on J_1).
- op_{2k} and op_{2l} belong to J_2, $k < l$, so $v_{2k} < v_{2l}$ (Increasing constraint on J_2).

These relations are established by the problem. Regarding to the relation between the operations op_{2k} and op_{1j} of different jobs, established by a solution for the problem, there can be two possibilities:

1. $v_{2k} < v_{1j}$: The relation between op_{1i} and op_{2l} can be:
    a. $v_{1i} < v_{2l}$ (Figure 6.1a). In this case, there are four possible ordered sequences and none of them contain cycles:
        i. $v_{1i} < v_{2k} < v_{2l} < v_{1j}$: There is no cycle, because of the increasing constraint in J_1.

ii. $v_{2k} < v_{1i} < v_{1j} < v_{2l}$: There is no cycle, because of the increasing constraint in J_2.

iii. $v_{2k} < v_{1i} < v_{2l} < v_{1j}$: There is no cycle, because of 1.

b. $v_{2l} < v_{1i}$ (Figure 6.1b). There is only one possible ordered sequence: $v_{2k} < v_{2l} < v_{1i} < v_{1j}$, without cycle because of 1.

2. $v_{1j} < v_{2k}$: The relation between op_{1i} and op_{2l} can be:

   a. $v_{1i} < v_{2l}$ (Figure 6.2a). There is only one possible ordered sequence: $v_{1i} < v_{1j} < v_{2k} < v_{2l}$, without cycle because of 2a.

   b. $v_{2l} < v_{1i}$ (Figure 6.2b). This case is not allowed, because the sequence would be $v_{1i} < v_{1j} < v_{2k} < v_{2l} < v_{1i}$ (there would be a cycle) and this is not allowed because of the increasing constraint in both jobs.

The symmetric proof is valid for the operations op_{2l} and op_{1i}.

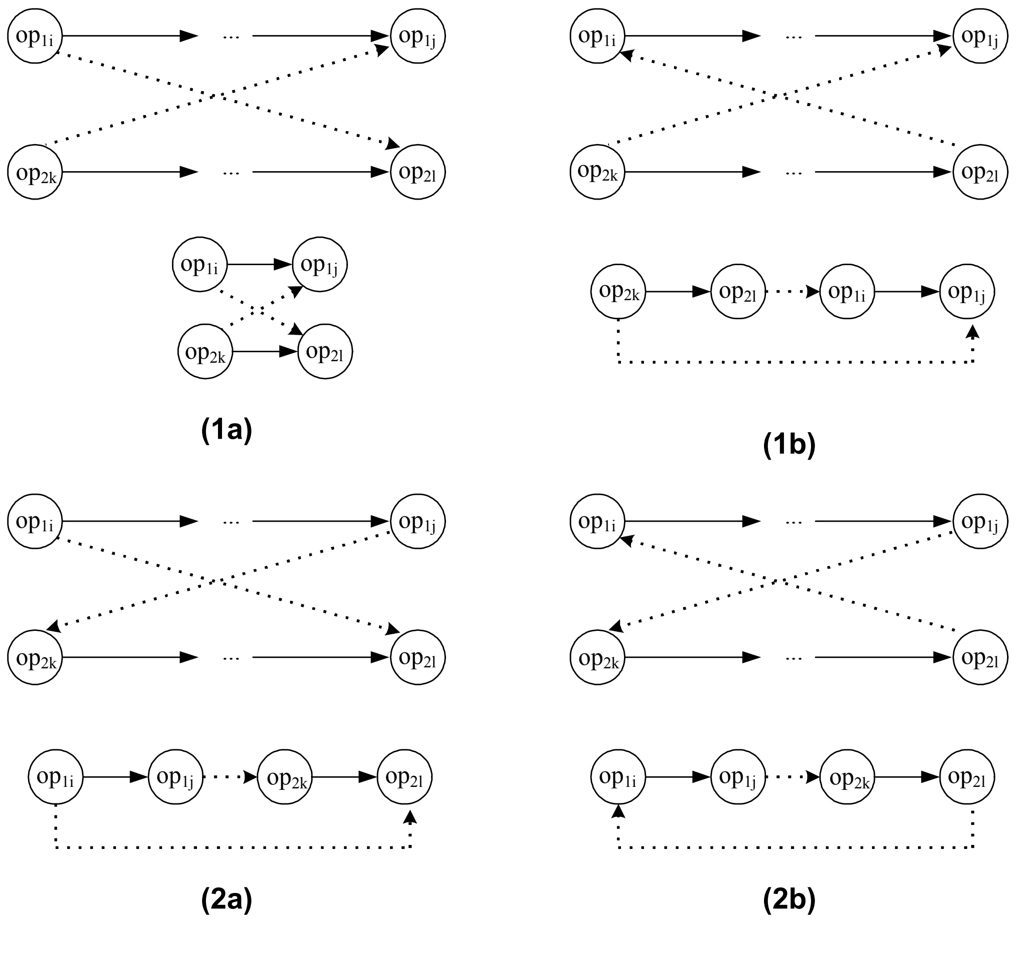


Figure 6. Cases for the proof of theorem 1

4.3. Neighborhoods

For JSSP, most of the successful approaches use neighborhood based on reversing critical operations (increasing their durations imply a larger makespan) that must be processed on the same machine. One of the best methods was proposed by Nowicki and Smutnicki [14], whose neighborhood was more constrained than other previous approaches. The movements allowed were to reverse two adjacent critical operations belonging to the same critical block (a sequence of critical operations on the same machine) so that one of them is not an internal operation in the block, excluding the swap between the first two operations of the first block when the second one is internal, and the swap between the last two operations of the last block when the first is internal.

A family of neighborhoods for the proposed model is defined, in which the basic idea is to make a swap between the values of two variables corresponding to operations of the same machine, i.e., between the relative order of those operations in a solution, trying to change the order of operations belonging to a critical path of a solution *S* (*CP(S)* from now), based on the Nowicki and Smutnicki (NS from now) neighborhood.

For a variable *v*, *σ(v)* is defined as the set of the variables *w* satisfying the following condition: the swap between *v* and *w* in *S* (denoted as *swap(v,w,S)*) causes a swap between *v* and *PM(v)* on *m(v)* and this is the only swap caused on *m(v)*. The variables *w* that meet this condition are those between $PM_2(v)$ (not included) and *PM(v)* (included) in *S*. It can be seen that the swaps between *v* and variables that appear before *PJ(v)* in *S* lead to unsatisfiable solutions. Then, *σ(v)*, when *v* is not the first in its job and has, at least, two predecessors on its machine, is defined as:

$$\sigma(v) = \{ w \in V \mid \max(PJ(v), PM_2(v)) < w \le PM(v) \}$$

If *PJ(v)* and $PM_2(v)$ do not exist, the outer lower bound is 0. On the other hand, if only one of them exists, the outer lower bound is established by it. Lastly, all the variables which have the smallest value on their machine (i.e., which are executed first) do not have any possible swaps ($\sigma = \emptyset$).

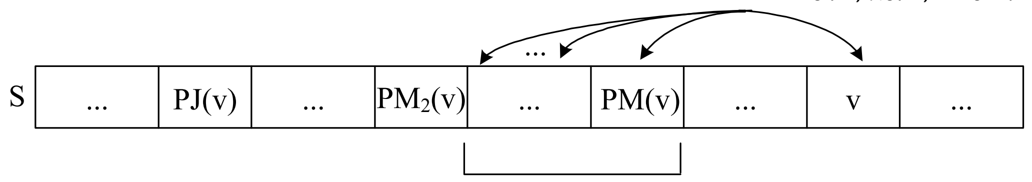
In Fig. 7 different cases of possible swaps are shown. In Fig. 7.a, *PJ(v)* appears before $PM_2(v)$, then the outer lower bound of the range of possibilities is established by $PM_2(v)$. In Fig. 7.b, $PM_2(v)$ is before *PJ(v)*, then it is given by *PJ(v)*. In Fig. 7.c, *PJ(v)* is after *PM(v)*, so no swap for *v* can be realized.

In order to reduce and set a maximum number of neighbors for a solution, a parameter *δ* is defined, as the maximum number of possible swaps for a variable *v*, from *PM(v)* toward variables appearing before it in *S*. It must be noticed that *δ* has to be greater than 1 so that the algorithm can reach any possible solution, taking into account the proposed model. According to this parameter, the set of considered swaps for a variable is defined as:

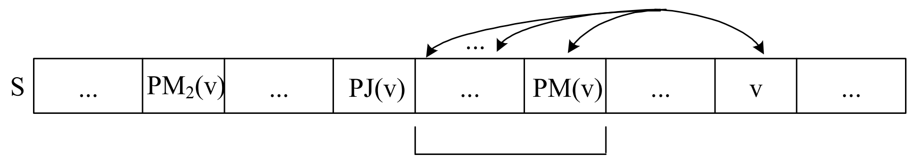
$$\sigma_\delta(v) = \{ w \in V \mid \max(PJ(v) - \delta, PJ(v), PM_2(v)) < w \le PM(v) \}$$

A family of neighborhoods, $N_{1\lambda}^{\delta}$, depending on the possible variables to swap, has been defined. For *λ= 0*, the idea is to swap variables that are at the beginning or at the end of a critical block (CB from now, *CB(v)* for the CB of a variable *v*), except the beginning of the first CB or the end of the last CB, similar to NS neighborhood. These variables are given by the set $V_0(S)$:

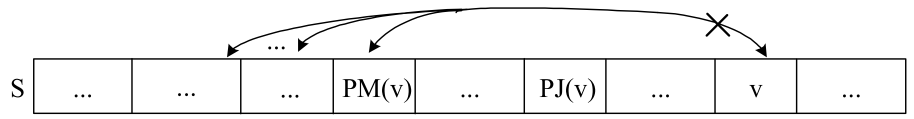
$$V_0(S) = \{ v \in CP(S) \mid v = SM(\mathit{first}(CB(v))) \vee v = \mathit{last}(CB(v)) \}$$



(a) Possible swap(v) when PJ(v)<PM$_2$(v)



(b) Possible swap(v) when PM$_2$(v)<PJ(v)



(c) Any possible swap(v) when PM(v)<PJ(v)

Figure 7. Possible swaps for a variable v

where *first(CB(v))* and *last(CB(v))* are the first and the last operations of *CB(v)*, respectively. *V$_0$(S)* contains the possible variables to be swapped in N_{10}^{δ} ($N_{10}^{\delta} = \{swap(v,w,S) \mid v \in V_0(S) \wedge w \in \sigma_{\delta}(v)\}$).

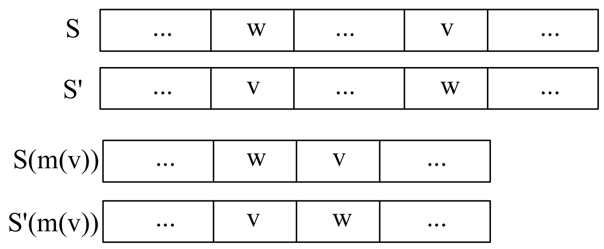
Due to the proposed model and the tabu search, it is possible to reach a solution which an empty neighborhood. In order to overcome this problem and get more diversification during the search, other more general neighborhoods $N_{1\lambda}^{\delta}$,, different from NS proposal, have been defined depending on a parameter *λ*. For *λ> 0*, it is allowed to swap internal variables of CBs, more internal as *λ* is increasing. The set of possible variables to swap, is now given by:

$$V_{\lambda}(S) = \{v \in CP(S) \mid (v = SM_{\lambda+1}(first(CB(v))) \vee v = PM_{\lambda}(last(CB(v)))) \wedge \lambda \leq \#CB(v)/2\}$$

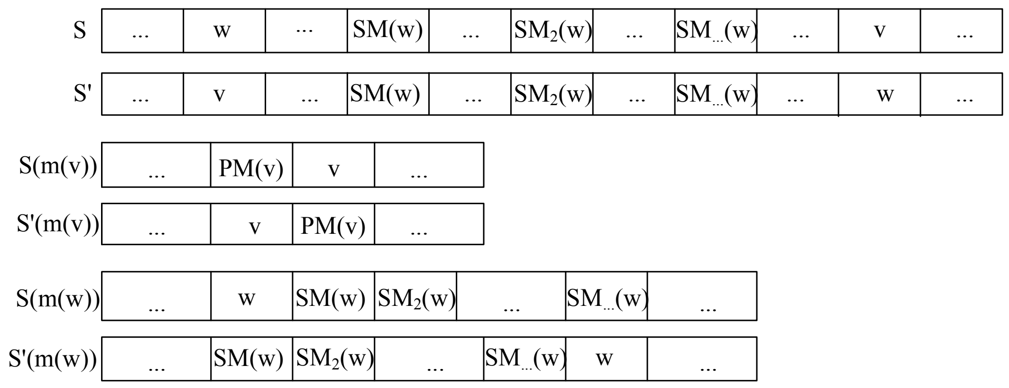
Then, the neighborhood $N_{1\lambda}^{\delta}$, is defined as $N_{1\lambda}^{\delta} = \{swap(v,w,S) \mid v \in V_{\lambda}(S) \wedge w \in \sigma_{\delta}(v)\}$. In order to allow swaps between all the non-critical operations (belonging or not to CP), another neighborhood has been defined: $N_{2}^{\delta} = \{swap(v,w,S) \mid v \in V \wedge w \in \sigma_{\delta}(v)\}$.

The swap between v and w has the following consequences:

1. Swap between the execution order of *v* and *PM(v)*, executed on the same machine, **which does not depend on *w*** (change in *m(v)*).
2. If *w ≠ PM(v)*, i.e. *m(w) ≠ m(v)*, other changes will be given. If *SM(w) < v*, then the execution orders of all the operations *w'* satisfying *m(w') = m(w)* and *w < w' < v* will be changed. Specifically, the relative order of all these operations are moved forward on their machine (change in *m(w)*).



(a) Swap between variables corresponding to operations executed in the same machine


S ... w ... SM(w) ... SM2(w) ... SM...(w) ... v ...
S' ... v ... SM(w) ... SM2(w) ... SM...(w) ... w ...
S(m(v)) ... PM(v) v ...
S'(m(v)) ... v PM(v) ...
S(m(w)) ... w SM(w) SM2(w) ... SM...(w) ...
S'(m(w)) ... SM(w) SM2(w) ... SM...(w) w ...

(b) Swap between variables corresponding to operations executed in different machines

Figure 8. Swap between variables

According to this, two types of movements can be given. First, the swap between variables corresponding to operations executed on the same machine, only one swap in *S(m(v))* is given (Fig. 8.a). Secondly, the swap between variables corresponding to operations executed on different machines, that leads to a swap in *S(m(v))* and several swaps in *S(m(w))*, one for each direct or indirect successor on the machine of *w* that is between *w* and *v* in *S* (Fig. 8.b). In Fig. 8 the neighbor for *S* is referred as *S'*.

4.4. The Parameterized Algorithm

Considering the defined neighborhoods, a local search algorithm has been developed (Algorithm 1). Although any initial solution can be used, the choice of better initial solutions usually allows to obtain better results, as it is found for the NS method [11]. In this way, for the experiments of the next section, the INSA algorithm [14] has been used. As indicated in Subsection 4.1, a schedule can be represented by different solutions of the model. Thereby, for selecting the actual initial solution a random procedure is used from the schedule obtained by the INSA algorithm.

According to the evolution of the search, different neighborhoods are used in order to select the next movement, which will correspond to a feasible solution. In each iteration, a movement to the best neighbor of N_{10}^{δ} is attempted (*λ= 0*), but, if the neighborhood is empty or all their members are in the tabu list, a more extended neighborhood is searched, by increasing *λ*. If ¸ reaches the allowed maximum value without finding a suitable next solution to visit, the more general neighborhood N_2^δ is used, and now the neighbor is selected randomly. N_2^δ is also used when the algorithm has not found a better solution for a number *K* of iterations. In this case, the algorithm returns to the best solution found so far.

Besides that, most of the computational cost of local search algorithms are due to the evaluation of neighbors. In order to reduce its amount, several approaches have been proposed, such as that of Taillard [20], which evaluates the neighbors using a lower bound estimation of the makespan in constant time, instead of calculating it exactly. In the proposed algorithm, the selection of candidates is made in two steps. First, the best swap between two critical operations is selected using the Taillard estimation of the expected makespan. After that, a variable is selected from the δ possibilities, choosing the one with the greatest improvement in its slack because of the change.

Algorithm 1: The parameterized local search algorithm

```
begin
    determine an initial solution $S$ randomly
    $best := S$;
    for $it := 1$ to $NIterations$ do
        if solution has not improved in last $K$ iterations then
            choose a neighbor $S'$ of $best$ in $N_2^\delta$ randomly;
        else
            $\lambda := 0$;
            repeat
                determine set $N_{1\lambda}^\delta$ of non-tabu neighbors of $S$;
                if $N_{1\lambda}^\delta$ is not empty then
                    choose a best solution $S'$ in $N_{1\lambda}^\delta$;
                else
                    $\lambda := \lambda + 1$;
            until $S'$ has been selected or $\lambda > maxBlockSize(S)/2$ ;
            if $S'$ has not been selected (all of $N_{1\lambda}^\delta$ are empty) then
                choose a neighbor $S'$ of $S$ in $N_2^\delta$ randomly;
        $S := S'$;
        if $S$ improves $best$ then
            $best := S$;
end
```

5. Experimental Results

ILOG JSolver [9] has been used for implementing the Algorithm 1, and for managing the constraints of the problem. As stated before, the algorithm has several parameters, δ, *K* (maximum number of iterations without improving the solution), and the tabu list size (*TLS*), that may affect its behavior, and its tuning represents a non-trivial problem. Since the main interest of this work is not the competitiveness of the algorithm proposed, but the CSP model which is defined, a scenario for some comparative results was chosen, in which the algorithm would be executed for a fixed number of 10000 iterations, which were selected randomly from the results of the INSA algorithm. For such situation, the value of *K* was chosen to be 1000. For selecting δ and *TLS*, the algorithm was run on a reduced set of instances for δ from

2 to 5 and from *TLS* from 5 to 10. The best results on the minimal and average makespan of the best solution after 10000 iterations were found for $\delta = 2$ and $TLS = 6$. The best results for $\delta = 2$ can be explained by the fact that for higher values of δ, there is more probability for finding a variable w such that the swap between v and w will be feasible, which would enforce the diversification strategy too much.

Table 1. Results on a set of JSS instances

				Proposed Model				NS			
Instance	*n*	*m*	*UB*	*BRE%*	*MRE%*	*SDRE%*	*RT*	*BRE%*	*RT*	$BRE\%_{104}$	RT_{104}
FT10	10	10	930	2.25	3.95	0.96	8.72	0	0.68	0	0.25
ABZ7	20	15	656	9.90	13.98	2.16	64.40	2.28	4.62	3.20	0.84
LA02	10	5	655	0.45	3.80	1.83	3.02	0	0.10	0	0.11
LA19	10	10	842	2.49	6.25	1.82	10.52	0.11	0.83	0.11	0.35
LA21	15	10	1046	5.16	8.23	1.05	18.76	0.86	0.86	0.86	0.42
LA24	15	10	935	3.85	6.56	1.13	18.72	1.39	1.33	1.50	0.45
LA25	15	10	977	7.26	11.24	1.84	20.65	1.12	1.39	2.04	0.45
LA27	20	10	1235	6.96	12.07	2.03	30.73	1.94	1.27	1.94	0.51
LA29	20	10	1152	8.42	12.57	2.22	33.24	3.13	3.40	4.51	0.48
LA36	15	15	1268	7.09	11.42	1.25	38.61	0.79	3.66	2.76	0.62
LA37	15	15	1397	9.09	14.59	2.30	41.80	1.50	2.74	3.29	0.78
LA38	15	15	1196	5.85	8.09	0.99	41.72	1.84	2.75	2.59	0.65
LA39	15	15	1233	7.94	10.44	0.90	41.04	0.89	3.50	1.62	0.79
LA40	15	15	1222	7.03	10.28	1.02	36.52	1.64	2.40	2.13	0.62
TA02	15	15	1244	6.35	10.49	1.64	36.47	2.73	2.83	2.73	0.70
TA18	20	15	1396*	12.60	15.25	1.32	57.82	3.65	4.64	5.73	0.97
TA26	20	20	1645*	9.36	13.14	1.34	93.66	3.10	10.64	3.28	1.58
TA32	30	15	1795*	14.20	17.84	1.30	116.93	3.12	18.36	6.85	1.44

Table 1 shows the results of the algorithm for a larger set of JSSP benchmarks, taken from the OR-library, and some harder instances from Taillard [19]. For each JSSP instance, the table shows some statistics about the algorithm used in this work: the relative error of the best solution from the 100 restarts (*BRE%*) with respect to the best known solution (*UB*, which is not proved optimal for the values indicated by *), the mean relative error (*MRE%*) and the standard deviation of relative error (*SDRE%*). Also, the mean computational time for running the algorithm is given (*RT*). As reference, the results obtained by the *NS* algorithm is shown in two situations: in the original form, that takes into account several factors, and after 10000 iterations. As expected, the algorithm is not fully competitive (as well as it has been developed in Java, many of its components are not optimized) with that of Nowicki and Smutnicki, considered as one of the best methods for solving the JSSP. Instead, the results shown must be taken as a reference for further improvements of the algorithm or for different approaches that can use the model, which is the main contribution of this paper.

6. Conclusions and Future Work

This paper proposes a CSP model for the Job Shop Scheduling Problem to be solved using local search techniques. The model can be used to represent a multiple software process planning problem when the different (activities of) projects compete for limited staff. The main aspects of the model are the use of integer variables which represent the relative order of the operations to be scheduled and two types of global constraints for ensuring feasibility. Also, a neighborhood for this model has been defined based on an adaptation of Nowicki and Smutnicki one. The main focus is not on the competitiveness of the algorithm which is proposed, but on the definition of the CSP model. This can be used directly by different algorithms based on local search or evolutionary strategies, so that they can be applied for solving real planning and re-planning problems.

As future work, the algorithm and neighborhood should be improved for solving more efficiently the JSSP. Also, the proposed model is expected to be adapted for other similar sequencing problems in a direct way.

On the other hand, it is expected to extend the software development process to other (more generic or specific) models and to adapt the corresponding (planning and/or) scheduling models and solving algorithms. Some interesting issues to be tackled are multi-mode tasks, multi-capacity resources, multi-objective optimization or uncertainty. Furthermore, it is intended to analyze widely the points where AI P&S tools can be effectively applied to the production and execution of models of organizations (business process management).

Acknowledgments

This work has been partially funded by the Consejería de Innovación, Ciencia y Empresa of Junta de Andalucía (P08-TIC-04095) and by the Spanish Ministerio de Ciencia e Innovación (TIN2009-13714) and the European Regional Development Fund (ERDF/FEDER).

References

[1] M. Boddy, A. Cesta, and S. Smith, ICAPS-04 Workshop on Integrating Planning into Scheduling, AAAI Press, 2004.

[2] P. Brucker and S. Knust, Complex Scheduling, Springer, 2006.

[3] R. Dechter, Constraint Processing, Morgan Kaufmann Publishers, 2003.

[4] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks", Artificial Intelligence, 1991, 49, pp. 61-95.

[5] U. Dorndorf, E. Pesch, T. Phan-Huy, "Solving the open shop scheduling problem", Journal of Scheduling 4 (3), pp. 157-174, 2001

[6] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling", Math. Oper. Res., 1976, 1(2), pp. 117-129.

[7] F. Glover and M. Laguna, Tabu Search, Blackwell Scientific Publishing, Oxford, England, 1993.

[8] H. H. Hoos and T. Stutzle, Stochastic Local Search. Foundations and Applications, Morgan Kaufmann, 2005.

[9] ILOG, "Ilog JSolver", 2003.

[10] A. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present, and future", European Journal of Operational Research, 1999, 113 (2), pp. 390-434.

[11] A. Jain, B. Rangaswamy, and S. Meeran, "New and Stronger Job-Shop Neighborhoods: A Focus on the Method of Nowicki and Smutnicki (1996)", Journal of Heuristics, 2000, 6, pp. 457-480.

[12] R. Kolisch and A. Drexl, "Local for multi-mode resource-constrained project". IIE Transactions (Institute of Industrial Engineers), 29(11), pp. 987-999, 1997.

[13] C. A. Leonhard, and J. S. Davis, "Job-Shop Development Model: A Case Study", IEEE Software, 1995, 12 (2), pp. 86-92.

[14] E. Nowicki and C. Smutnicki, "A fast taboo search algorithm for the job-shop problem", Management Science, 1996, 42(6), pp. 797-813.

[15] M. D. R-Moreno, D. Borrajo, A. Cesta and A. Oddi, “Integrating planning and scheduling in workflow domains”, Expert Systems with Applications 33, 2007, pp. 389-406.

[16] F. Rossi, P. van Beek, and T. Walsh, Handbook of Constraint Programming, Elsevier, 2006.

[17] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, D. Edmond, “Workflow Resource Patterns: Identification, Representation and Tool Support”. In: Pastor, Ó., Falcao e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, Springer, Heidelberg, 2005

[18] D. Smith, J. Frank, and A. Jónsson, “Bridging the gap between planning and scheduling”. Knowledge Engineering Review, 15(1), pp. 47–83, 2000.

[19] E. Taillard, “Benchmarks for basic scheduling problems”, European Journal of Operational Research, 1993, 64, pp. 278-285.

[20] E. Taillard, “Parallel Taboo Search Techniques for the Job-Shop Scheduling Problem”, ORSA Journal on Computing, 1994, 16(2), pp. 108-117.

[21] W. M. P. van der Aalst, A.H.M. Hofstede and M. Weske, “Business Process Management: A survey”, Business Process Management: International Conference, BPM 2003, 2003: Proceedings.

[22] R. Vaessens, E. Aarts, and J. Lenstra, “Job-shop scheduling by local search”, INFORMS Journal on Computing, 1994, 8, pp. 302-317.

[23] P. Van Hentenryck and L. Michel, Constraint-Based Local Search, The MIT Press, 2005.