RAIRO Operations Research
RAIRO Oper. Res. **41** (2007) 155–170
DOI: 10.1051/ro:2007015

A BRANCH-AND-BOUND METHOD FOR SOLVING MULTI-SKILL PROJECT SCHEDULING PROBLEM

Odile Bellenguez-Morineau[1] and Emmanuel Néron[1]

Abstract. This paper deals with a special case of Project Scheduling problem: there is a project to schedule, which is made up of activities linked by precedence relations. Each activity requires specific skills to be done. Moreover, resources are staff members who master fixed skill(s). Thus, each resource requirement of an activity corresponds to the number of persons doing the corresponding skill that must be assigned to the activity during its whole processing time. We search for an exact solution that minimizes the makespan, using a Branch-and-Bound method.

Keywords. Project Scheduling, Skills, Branch-and-Bound method.

Mathematics Subject Classification. 90C57, 90B30.

1. Problem definition

The problem we tackle is Multi-Skill Project Scheduling Problem. This model can be applied to different cases of project scheduling problem, where resources have more than one skill. This section defines the problem and presents an example. Then a link with classical project scheduling problem is presented.

In Multi-Skill Project Scheduling Problem, the project to schedule is made up of a set of activities A_i, $i \in \{0, ..., n\}$, linked by classical end-to-start precedence relations. So an activity-on-node graph $G = (A, E, d)$ is built, where A_0 and A_n are two dummy activities that correspond respectively to the beginning and the end of the project. An arc $(A_i, A_j) \in E$ if and only if A_i is a direct predecessor of A_j. Each arc (A_i, A_j) has a valuation $p_i \in d$, which is equal to the processing

Received September 30, 2005. Accepted January 25, 2006.

[1] Laboratoire d'Informatique de l'Université de Tours, 64 av. Jean Portalis, 37200 Tours, France; odile.morineau@univ-tours.fr

time of A_i. Each activity of the project has specific requirements of renewable resources to be processed.

The resources are staff members. Each member P_m, $m \in \{0, ..., M\}$, masters one or more specific skill(s) among all the skills S_k, $k \in \{0, ..., K\}$ existing in the project, so $MS_{m,k} = 1$ if and only if P_m masters S_k, 0 otherwise. Thus, each unit of skill required by an activity corresponds to a person that has to be assigned to do the required skill for this activity. For each activity A_i and each skill S_k, $b_{i,k}$ is the number of persons that we have to assign to A_i to do S_k during the whole processing time of A_i. A person can be assigned to a need only if he/she masters the required skill. Moreover, for those staff members we consider a finite number of unavailability periods. If a person P_m is available between $[t; t+1[$ then $Avail(P_m, t) = 1$, 0 otherwise. So, a person P_m can be chosen to do S_k for A_i iff $MS_{m,k} = 1$ and $Avail(P_m, t) = 1$, $\forall t \in [t_i; t_i + p_i[$, where t_i is the starting time of the activity. We are interested in the computation of a solution that minimize the makespan of the project. We recall main notations in Table 1.

The notion of skill have already been studied in workforce planning field [10, 21, 26], but to the best of our knowledge, contributions proposed for project scheduling problem are very few: there are two different ways to take into account skills of employees, either the problem is to find a solution where the assignments match the skills of employees [3, 4, 6, 19] and eventually their level of abilities [5], or the problem is to compute a solution at a minimum cost under different constraints [23]. In the latter case, every assignment of an employee has a cost that grows up if the employee is not well skilled for the activity to do, moreover the global project has a due date, and there is a penalty if the project is delayed after this due date.

The model of MSPSP was inspired from a problem appeared in the software development industry, where employees have several skills among programming, analysis, design, and so on. But it also arises to schedule training operators in a call center, when trainers master different types of training tasks [2]. More generally it can be used to model any problem where the resources have specific skills among those needed by the activities.

Here is a small example of MSPSP: the project is presented in Figure 1, and Tables 2 and 3 present respectively needs of activities and skills of staff members. In this example, there are 4 different skills, 6 activities, including the dummy source and sink that have no need, and 6 staff members. The goal is to minimize the makespan. Figure 2 presents one optimal schedule.

This kind of problem can be modelled as a Multi-Skill Project Scheduling Problem but it can also be seen as a particular case of Multi-Mode Resource-Constrained Project Scheduling Problem (MM-RCPSP) [24]. In this last one, activities have to be scheduled respecting resource and precedence constraints, but an activity has different ways of consuming resources: for every activity, there exist different *modes*. A mode is defined by a processing time and a given amount for each resource. So, once a mode is chosen for the execution of an activity, we exactly know which resource(s) will be required, in which quantity, and the duration of the activity.

TABLE 1. Input data and auxiliary notations.

Notation	Definition
Activity data	
$n+1$	number of activities.
A_i, $i \in \{0, ..., n\}$	the set of activities of the project. A_0 is the dummy start node and A_n the dummy end of the project.
p_i	the processing time of the activity A_i.
$G = (A, E, d)$	the precedence graph.
$(A_i, A_j) \in E$	if there exists a precedence relation between A_i and A_j.
Resource data	
K	number of skills.
M	number of staff members.
T_{max}	Maximum time horizon considered for the project scheduling.
S_k, $k \in \{0, ..., K\}$	the set of skills. k is the number of the skill.
P_m, $m \in \{0, ..., M\}$	the staff members.
$MS_{m,k}$	equals to 1 if P_m is able to do S_k, 0 otherwise.
$Avail(P_m, t)$, $m \in \{0, ..., M\}$, $t \in \{0, ..., T_{max}\}$	equals to 1 if P_m is available at time t, 0 otherwise.
$b_{i,k}$, $i \in \{0, ..., n\}$, $k \in \{0, ..., K\}$	the number of employees, able to do S_k required to execute A_i.
Auxiliary notation	
r_i	the earliest starting time of the activity A_i according to precedence constraints.
t_i	the starting time of the activity A_i.
$Avail(P_m, t_1, t_2) = \sum_{t=t_1}^{t_2-1} Avail(P_m, t)$, $m \in \{0, ..., M\}$	the total time P_m is available between t_1 and t_2.

There is a link between these two different models because MM-RCPSP formulation can be used to describe a MSPSP instance. In fact, a mode corresponds to a given subset of staff members that matches the requirements of the activity. Every mode has the same processing time, and there exist as many different modes as feasible subsets of staff members satisfying needs of the activity. The main difference between MM-RCPSP and MSPSP lies in the number of modes usually proposed for each activity: in classical instances of Multi-Mode RCPSP [14, 15], there are at most 10 modes per activity, but if we want to enumerate the

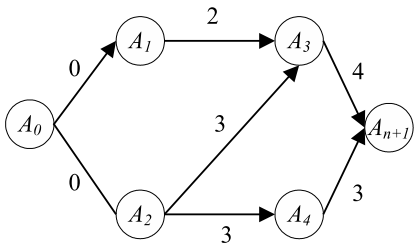

FIGURE 1. Example of precedence graph of the project.

TABLE 2. Example of needs of activities.

$b_{i,k}$	S_0	S_1	S_2	S_3
A_1	2	0	1	1
A_2	0	1	0	1
A_3	0	1	2	0
A_4	1	0	1	0

TABLE 3. Example of skills of employees.

$S_{m,k}$	S_0	S_1	S_2	S_3	Unav. periode
P_0	1	1	0	0	[2;3[
P_1	0	1	1	0	-
P_2	0	1	0	1	[2;6[
P_3	1	1	1	0	-
P_4	1	0	1	1	[6;8[
P_5	1	0	1	0	-

number of possible subsets of staff members in an instance of MSPSP, it will be very much larger. In some small instances with 3 skills and 10 persons, the number of modes per activity can exceed 1000. In the example presented there exist 13 different feasible subsets of staff members that can be assigned to the activity A_1. Moreover, most of the exact methods proposed for solving exactly MM-RCPSP [9, 11, 14, 20, 22] have a branching scheme based on an explicit enumeration of the modes for each activity. Thus, these methods cannot be used for solving exactly the MSPSP.

Next section introduces the branching scheme chosen for the branch-and-bound method, then Section 3 presents how a leaf node is treated. Section 4 describes upper and lower bounds used during the exploration of the search tree. Section 5 introduces the two different branching strategies tested and finally Section 6 introduces data sets and experimental results.

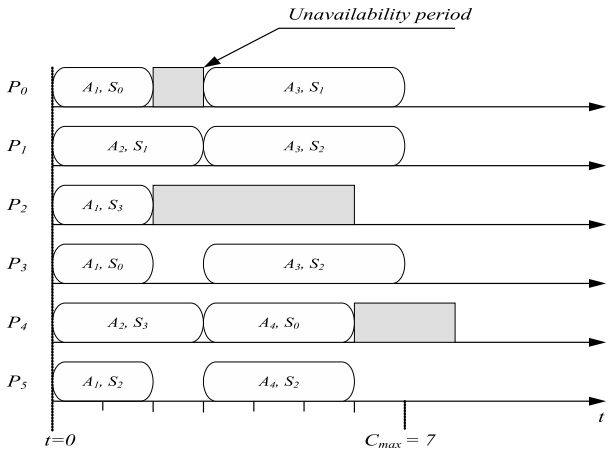

FIGURE 2. Example of one optimal solution.

2. Branching scheme

In order not to use a branching scheme based on the explicit enumeration of all the feasible subsets of resources for activities, we propose to use a branching scheme inspired from the one introduced by [7, 8], and based on the reduction of the slack of one activity at each node.

For every activity in the project, a release date r_i is computed according to the precedence graph. So, $r_i = \mathcal{L}(A_0, A_i)$, the length of a longest path from A_0 to A_i in the precedence graph. In the same way, once an upper bound is computed, it is used to compute a deadline for each activity according to the precedence relations: $\tilde{d}_i = UB - (\mathcal{L}(A_i, A_n) - p_i)$, where UB is the best known upper bound. Notice that $\tilde{d}_i$ has to be updated each time that a new best known solution is found. Thus, every activity has its own time-windows that allows to compute a slack m_i, equal to $\tilde{d}_i - r_i - p_i$.

At each node an activity is chosen, and the slack m_i of the activity A_i is updated. Each node has two children $N1$ and $N2$: in the first ($N1$) the deadline $\tilde{d}_i$ of the activity is decreased to $\tilde{d}_i - \lceil \frac{m_i}{2} \rceil$, and in the second ($N2$) the release date r_i is increased to $r_i + \lfloor \frac{m_i}{2} \rfloor$. By this way, the two children define disjoint sets of schedules. We reach a leaf node when every activity has a slack equal to 0. That means we exactly know that it has to start at its release date in order to finish at its deadline ($r_i + p_i = \tilde{d}_i$).

But the main difference with exact methods known for RCPSP or MM-RCPSP lies on the fact that it is not enough to define a solution. In fact, when every starting time is fixed, it possibly corresponds to several solutions or no solution, because resources are not yet affected. The corresponding assignment problem we have to solve is introduced in Section 3.

The rule chosen to select the activity on which the branching scheme is applied is really important, because each time a release date or a deadline is updated, this has to be propagated on its successors or predecessors, so it may increase the speed to reach a leaf node. Different strategies are presented in Section 5.

3. Leaf node treatment

To know if a leaf node corresponds or not to at least one valid solution, an assignment problem has to be solved. This problem can be seen as a Fixed job Scheduling Problem (FSP): for a set of activities, resources have to be assigned to match the requirements. For every activity, the starting time and the processing time are fixed. The resources can be split up into disjoint machine classes, and there exists a matrix which defines for each activity and each classe of resource if those resources are allowed to do the activity or not. This problem has been proved to be $\mathcal{NP} - Hard$ in the strong sense [13]. It is solved using an integer linear programming formulation. But before calling this method to solve a leaf node of the MSPSP, decomposition rules introduced below are applied.

3.1. Decomposition methods

Once a leaf node is reached, a Fixed job Scheduling Problem must be solved in order to know if there is a solution respecting fixed starting time of activities. But the corresponding FSP contains as many activities as the MSPSP instance, so it can be quite large. Therefore, we want to have a solution as quickly as possible. This is the reason why decomposition rules are called: it is easier to solve several small-size sub-problems instead of solving the global assignment problem.

The first decomposition rule is a temporal one. For a given instance of FSP, if we can find a cutting time-point t, that means a time t where no activity is in progress, the instance can be separated in two sub-parts. The first sub-part is made of all the activities that end before or at t, and the second one is made of all the activities that begin after or at t. And if at least one of these sub-parts has no solution, the original instance of FSP does not have any solution.

Property 1. *t is a cutting time-point if and only if: $\forall i \in \{0, .., n+1\}$, $\tilde{d}_i \leq t$ or $r_i \geq t$.*

Finding a cutting time-point leads to decomposition because a cutting time-point t is defined by the fact that no activity is in progress at t. So, whatever are the choice of assignment which are made before t, the corresponding persons are occupied until t in the worst case. It will have no consequence after t. By the same way, no assignment chosen after t have a consequence before t. So, the two sub-problem are independant.

The second decomposition rule is based on skill requirements. Any instance of Fixed job Scheduling Problem can be decomposed in sub-parts if there are some independant sets of skills. Two sets of skills are said to be independant if there does not exist a person who masters a skill of the first set and a skill of the second

one, and if there is no activity that requires a skill from the first set and a skill from the second one. A sub-part of the problem is built for each independant set of skills. Once again, if at least one sub-part does not have any solution, the global instance of FSP has no solution.

Property 2. *Two sets of skills* Ω_1 *and* Ω_2 *are said to be independant if and only if* $\forall k \in \Omega_1, \forall k' \in \Omega_2, (\nexists A_i | b_{i,k} \cdot b_{i,k'} \neq 0) \wedge (\nexists P_m | S_{m,k} \cdot S_{m,k'} = 1)$

A sub-part corresponding to an independant set of skill can be solved independantly from other sub-parts of the problem because it corresponds to a part of the global problem that takes into account skills not needed by activities in other sub-parts and not mastered by persons in other sub-parts. Moreover, activities of this sub-part do not need skills of an other independant set and persons of this sub-part do not master skill of an other independant set. So, choices of assignment made on activities and persons of this sub-part have no consequence on an other sub-part.

According to those decomposition rules, an instance of FSP get in a leaf node of the B&B method can be splitted into different small-size sub-parts, and if one of those sub-parts can be proved to be unfeasible, the leaf node can be pruned, or if all the sub-parts are feasible, we have a complete solution for the instance. In order to know if the sub-parts of an instance are feasible or not, the method used for FSP is applied.

3.2. Fixed Job Scheduling Problem treatment

In order to solve the FSP, we used an integer linear program, inspired from those existing for the Fixed Job Scheduling Problem [13, 16]. In the model below, $x_{i,m,k} = 1$ if P_m is assigned to do S_k for the activity A_i, 0 otherwise.

$$\forall i \in \{0..n\}, \forall m \in \{0..M-1\}, \qquad \sum_{k=0}^{K-1} x_{i,m,k} \leq 1 \tag{1}$$

$$\forall (i,j) \in \{0..n\}^2 \text{ s.t } t_i \leq t_j < t_i + p_i,$$

$$\forall m \in \{0..M-1\}, \quad \sum_{k=0}^{K-1} x_{i,m,k} + \sum_{k=0}^{K-1} x_{j,m,k} \leq 1 \tag{2}$$

$$\forall i \in \{0..n\}, \forall k \in \{0..K-1\}, \qquad \sum_{m=1}^{M} x_{i,m,k} = b_{i,k} \tag{3}$$

$$\forall i \in \{0..n\}, \forall k \in \{0..K-1\}, \forall m \in \{0..M-1\}, \qquad x_{i,m,k} \leq S_{m,k} \tag{4}$$

$$\forall i \in \{0..n\}, \forall k \in \{0..K-1\}, \forall m \in \{0..M-1\},$$

$$\exists\, t \in [t_i, t_i + p_i[\text{ s.t. } A(P_m, t) = 0, \qquad x_{i,m,k} = 0 \tag{5}$$

If there exists a solution to this model then there is a solution for the corresponding Fixed Job Scheduling Problem. Constraint (1) ensures that a person does at most one skill for a given activity; (2) forces one person not to be assigned to

two activities that overlap; (3) implies that all skill requirements of activities are satisfied; (4) ensures that a person can be assigned to a need if and only if he/she masters the corresponding skill. Finally, (5) ensures that a person can be assigned to an activity if and only if he/she is available all along the processing time of the activity. This model is solved using Cplex 8.0.

4. Lower and upper bounds

The number of nodes that has to be treated may be very large, and even when a leaf node is reached, the assignment problem to solve may be time-consuming. Thus, efficient bounds have to be used. This part is devoted to the description of a heuristic indicator needed by the upper bound, the upper bound and the two lower bounds used to prune the search tree.

4.1. Criticity indicator

Due to the fact that for a given instance the number of different feasible subsets of persons for one activity can be very large, we need to evaluate the ones that should be better than others, in order to build as quickly as possible a solution as good as possible. For this reason, we compute a heuristic indicator: the *criticity*. This indicator is based on the way skills are needed by the project, in order to distinguish which ones will be critical for the project. So, each time we try to schedule an activity A_i at time t_i, the criticity is computed by the following way:
For a skill S_k:
$CS_k^{t_i} = \frac{Total\ time\ S_k\ will\ be\ required\ by\ activities\ during\ [t_i;t_i+p_i[}{Total\ time\ persons\ can\ do\ S_k\ during\ [t_i;t_i+p_i[}$, *i.e.*
$CS_k^{t_i} = \frac{\sum_{j\in\mathcal{C}(S_k,t_i,t_i+p_i)} p_j}{\sum_{m=0}^{M} Avail(P_m,t_i,t_i+p_i)\cdot MS_{m,k}}$, where $\mathcal{C}(S_k,t_i,t_i+p_i)$ is the set of activities that require S_k between t_i and t_i+p_i.
Then, for a person P_m:
$CP_m^{t_i}$ = sum of the *criticity* of all the skill(s) he/she masters, *i.e.*
$CP_m^{t_i} = \sum_{k=0}^{K}(S_{m,k}\cdot CS_k^{t_i})$
This indicator is computed a priori (see [3] for further details). It will be used in the computation of the upper bound in order to discriminate different assignments on an activity.

4.2. Upper bound

The upper bound used to evaluate the makespan of the project is based on a heuristic method introduced in [3]. This greedy method inspired from the Serial Schedule Scheme for RCPSP [12] is based on a priority rule to sort activities and a dispatching rule to assign activities to person. Then activities are scheduled at their minimum feasible starting time on the set of available staff members that is 'less useful' for other activities. That means we choose the set of persons with the

minimum sum of staff members criticity indicators. At the beginning the upper bound is the best value of the makespan we get with eight different usual priority rules to order activities (Minimum Slack Time, Latest Starting Time, ...). We call this procedure at the root node, and each α nodes in order to decrease the gap between lower bound and upper bound if possible, with α a parameter between 10 and 500. The upper bound is also updated every time a better solution is found at a leaf node.

4.3. Two lower bounds

The two lower bounds introduced here have been presented in [19]. Both are destructive, that means once a value D is fixed, either we are able to detect an unfeasibility, so D cannot be respected as a deadline for the project and D+1 is a valid lower bound, or we cannot detect unfeasibility and have no conclusion on D.

In practice, at the root node, we use binary search between the first value of D, given by the critical path lower bound and a valid upper bound given by the greedy algorithm (see Sect. 4.2). For other nodes, we only have to test the current value of the upper bound UB_{best} minus 1: if it cannot be respected, i.e. if at least one of the two lower bounds detects an unfeasibility, the node is pruned because we know that for this node UB_{best} is a valid lower bound so this node cannot lead to a solution better than the best one already known.

4.3.1. *Lower bound based on blocks*

The first lower bound used is inspired from one existing for the RCPSP [18]. It is based on the notion of block. A block, or anti-chain, is a feasible subset of activities that can be processed simultaneously, *i.e.*, without violating neither the resource constraints nor the precedence constraints. So, for each couple of activities (A_i, A_j), tests are made in order to know if it is possible for them to be in progress at the same time. Lack of precedence relations is first checked, then overlapping of their time-windows, and finally the resource constraint is checked by solving the corresponding assignment problem using a max-flow formulation.

The graph $G_1 = (X_1, F, c)$, $X_1 = \{A_i, A_j\} \bigcup \{S_k | \forall k \in \{0..K-1\}\} \bigcup \{P_m | \forall m \in \{0..M-1\}\}$ in which we search for a maximum flow is presented in Figure 3. The first layer is made up of the activities A_i and A_j, the second one is made up of the skills needed by at least one of the two activities, and the third is made up of staff members available during the time interval considered. There exists an edge $e_a \in F$ between the source and an activity A_i, with a maximum capacity $\varphi_a \in c$ which is equal to the sum of the needs of the activity considered. There is also an edge $e_a \in F$ between an activity and a skill, its maximum capacity $\varphi_a \in c$ is equal to the number of persons needed by the activity for the corresponding skill. An edge $e_a \in F$ exists between a skill and a person only if the person masters this skill and its maximum capacity $\varphi_a \in c$ is then equal to 1. Finally, there exists an edge $e_a \in F$ between a staff member and the sink, with a maximum capacity $\varphi_a \in c$ equal to 1.

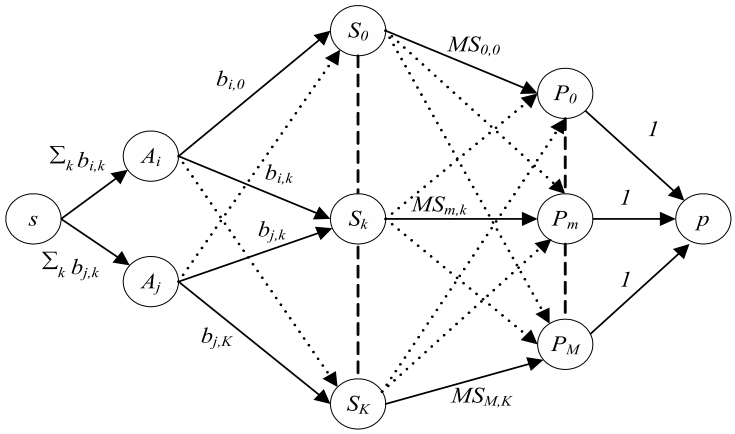

FIGURE 3. Graph G_1 used to check resource constraint on two activities A_i and A_j.

Edges have maximum capacity: from the source to an activity it is equal to the sum of the needs of the activity we consider, from an activity to a skill it is equal to the number of persons needed by the activity for the corresponding skill and from a staff member to the sink it is equal to 1, because a person cannot do more than one skill for one activity at a time. An edge between a skill and a person exists only if the person masters this skill and its maximum capacity is then equal to 1.

The maximum flow is computed on this graph and it is compared to the sum of the needs of the two activities we are testing. If they are equal we conclude that the two activities can be in progress at the same time, but if the maximum flow found is strictly lower than the sum of the needs, we know that there is a resource conflict between the two activities.

Once we know exactly which couples of activities can be in progress at the same time, the graph of "compatibility" is built. In this graph $G' = (A, E')$, there exists an arc between two activities $A_i \in A$ and $A_j \in A$ only if there is no unfeasibility for them to be in progress at the same time. Each node has an associated weight p_i that is equal to the processing time of the activity. In order to determine the longest set of activities that cannot be in progress at the same time, which is a lower bound of the project duration, we search for a stable set with the maximum weight in this graph. This resolution is based on a MIP, solved by Cplex 8.0:

$$\begin{cases} \max \sum_i u_i \cdot p_i \\ u_i \in \{0,1\} \\ u_i = 1 \text{ if } A_i \text{ is in the stable set, 0 otherwise} \\ \text{s.t. } (A_i, A_j) \in G' \Rightarrow u_i + u_j \leq 1 \end{cases}$$

4.3.2. *Lower bound based on energetic reasoning*

The second lower bound used is based on energetic reasoning used to get satisfiability tests for the classical RCPSP in [1] and [17]. This is based on the fact

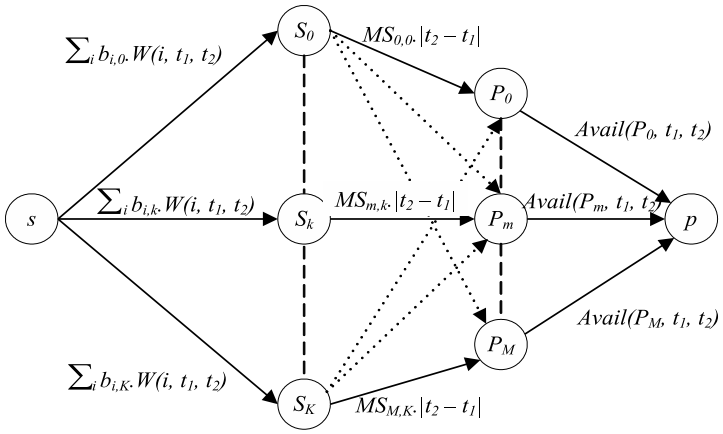

FIGURE 4. Solving the assignment problem for energetic lower bound.

that on a given time-interval $[t_1, t_2]$, where we assume $t_1 < t_2$ without any loss of generality, we are able to detect if all the mandatory parts of the activities that have to be processed in this time-interval can be done or not. Time points t_1 and t_2 are taken in $T = \{r_i, r_i + p_i, \tilde{d}_i - p_i, \tilde{d}_i, \forall i \in \{1..n\}\}$ and the mandatory part $w(i, t_1, t_2)$ of an activity A_i that has to be scheduled between t_1 and t_2 is computed either by left-shifting or right-shifting the activity in its time-window $[r_i, \tilde{d}_i]$. So the left-work of A_i is equal to $max(0, r_i + p_i - t_1)$ and its right-work is equal to $max(0, t_2 - (\tilde{d}_i - p_i))$. The mandatory part of A_i is equal to min(left-work,right-work, $p_i, t_2 - t_1$).

Once all the mandatory parts are computed, we check if there is enough available resources on this interval to execute at least all of them. As previously, this problem can be modelled as an assignment problem that can be solved using a max-flow formulation presented in Figure 4. This graph G_2, where we search for the maximum flow in order to verify if there are enough resources is made of a first layer of nodes that represent each skill S_k and of a second layer of nodes that correspond to the staff members. Each edge from s to a level of a skill S_k has a maximum capacity equal to the mandatory parts times the number of persons needed for this skill ($\sum_{i=0}^{n} \sum_{k=0}^{K} (w(i, t_1, t_2) \cdot b_{i,k})$). An edge between a skill S_k and a staff member P_m exists if the staff member masters the skill, i.e. $S_{m,k} = 1$, and has a maximum capacity equal to the length of the time-interval $[t_1, t_2]$, and finally the edge between a person P_m and p has a maximum capacity equal to the total time this person is available between t_1 and t_2. Then we are sure that if there exists at least one time-interval where those mandatory parts cannot be satisfied, the node can be pruned. The complexity of this lower bound is $(K + M)^3$ but in practice, it is well-solved. As mentionned in [19], this lower bound is complementary with the first one so both are used in the search tree, in order to evaluate nodes.

4.4. Time-bound adjustment

Another interesting way to improve the search speed is based on time-bound adjustments: if time-windows of activities can be decreased, either we can prune a node earlier if an unfeasibility is detected, or it possibly reduces the number of nodes needed to reach a leaf node.

The first way to do it is close to the second lower bound, based on energetic reasoning. (It is an extension of time-bound adjustments proposed for RCPSP [1].) On a time-interval $[t_1, t_2]$, where t_1 and t_2 are generated by the same way as in Section 4.3.2, we use all the mandatory parts $w(i, t_1, t_2)$ of activities A_i except the one of an activity A_j. Then, a binary search is applied with max-flow tests in order to determine the maximum part of activity A_j that can be done on this time-interval, respecting ressource constraint. If this maximum part W_{max} is smaller than the left-work of A_j, r_j is updated to $t_2 - W_{max}$, or if it is smaller than the right-work of A_j, $\tilde{d_j}$ is updated to $t_1 + W_{max}$. Once this is done for every activity, the new time-windows are propagated according to the precedence graph, and time-bound adjustment are applied again, until we cannot find any new adjustments. This had been proved to be efficient for RCPSP, but it is solved in a pseudo-polynomial time so it can be time-consuming. This is the reason why these adjustments will be done only at the root node, until there is no time-window to update.

In order to adjust time-windows of activities if possible all along the search tree, some obvious tests are done. At each node, for every activity A_j, on every time interval $[t_1, t_2]$, we have to compute:

- $MaxPart = (\sum_m A(P_m, t_1, t_2) - \sum_k \sum_{i \neq j} w(i, t_1, t_2) \cdot b_{i,k}) / (\sum_k b_{j,k})$, which is the maximum part of A_j that can be done according to the global available staff members, and the sum of the needs of other activities. It is equivalent to relax the skill matching constraint.
- $\forall k$, $MaxPart_k = (\sum_{m \in E_k} A(P_m, t_1, t_2) - \sum_{i \neq j} w(i, t_1, t_2) \cdot b_{i,k}) / (b_{j,k}))$, with E_k the set of staff members that master S_k. This is the maximum part of A_j we can schedule on this time-interval according to the available staff members that master the skill S_k and the global need of this skill. This is equivalent not to take into account that a person cannot do more than one skill at a time.

Then the minimum of all those values is computed, it is called $CapaMax$: $CapaMax = min\ (MaxPart,\ min_k\ MaxPart_k)$. If the left-work of A_j is bigger than $CapaMax$ then r_j is updated to $t_2 - CapaMax$, or if the right-work is bigger than $CapaMax$, $\tilde{d_j}$ is updated to $t_1 + CapaMax$.

The last way that can be used in order to limit the search is to well-choose the way we explore this tree, according to the branching strategy, trying to decrease the number of level that are necessary to reach a leaf node.

5. Branching strategy

Due to the size of the search tree that may be important, we choose a depth first branching strategy. Add to this, the way we choose to build nodes is really important to decrease the total number of nodes explored. First, a chosen activity of which slack is decreased can make decrease several other activities slacks by propagation on the precedence graph, and so increase the speed we reach a leaf node. Then the way the search tree is explored can give quickly a good solution, and decrease the number of node by updating the global upper bound. This is the reason why we test different search strategies in order to compare them and keep the best one.

The first one is based on the size of every slack. The activity chosen at each level to be branched is the one with the maximum slack. This strategy makes the biggest slack reduction possible, and the propagation may really decrease some other slacks.

The second one is based on the first lower bound (*cf.* Sect. 4.3.1). In this lower bound we have to build a maximum-weighted stable set, this set at least contains a critical path, so updating those activities' slack should disturb the makespan. So at each level, once the stable set is computed, the activities that it contains are stored, and we branch on one of those that do not have a slack equal to 0.

And last search strategy we use is based on the second lower bound (*cf.* Sect. 4.3.2), where we test all time intervals $[t_1, t_2]$ and verify if there is a resource conflict or not. If this bound does not detect an unfeasibility we know that for each interval the maximum flow found is equal to the sum of the needs of the mandatory parts, but during all those intervals the amount of resources potentially consumed is not the same. So, the activity we decide to branch on is from an interval as loaded as possible, that means where requirements are maximum. So,we take first an activity from the interval where $\frac{\sum_i w(i,t_1,t_2)\cdot\sum_k b_{i,k}}{\sum_m Avail(P_m,t_1,t_2)}$ is as near to 1 as possible.

This two last ways of branching may increase the speed we detect an unfeasibility, so the number of nodes created may be decreased too.

6. Experimental results

In order to test this method, lots of instances have been generated. MSPSP is really close to other project scheduling problems like RCPSP or Multi-Mode RCPSP [14, 15, 23, 25], so we decide to keep precedence graph from existing data sets of classical RCPSP and Multi-Mode RCPSP instances and add skill requirements in order to get MSPSP instances.

We take a set of instances from Multi-Mode RCPSP and Single-Mode RCPSP with a number of activities equal to 12, 14, 16, 18, 20, 22, 25, or 32. We compute by this way 1070 various instances. To generate resources and needs of activities we used pseudo-randomly generation in order to have between 3 and 5 skills and between 4 and 10 persons assigned to the project. All those persons do not have

TABLE 4. Comparison between the deviation and execution time.

#Act.	branching strat.	Av. time (s)	#Unreas.	Av. dev. on unreas.(%)
	Max. slack	0.21	4	7.88
≤ 14	Stable set	0.17	4	7.88
(346 instances)	Load. time-int.	0.09	4	7.88
	Max. slack	9.68	6	8.21
$16 \leq \#act. \leq 18$	Stable set	10.84	6	8.21
(344 instances)	Load. time-int.	9.74	6	8.21
	Max. slack	141.96	50	6.68
$20 \leq \#act. \leq 32$	Stable set	146.18	54	6.93
(380 instances)	Load. time-int.	142.53	51	6.64

the same number of skills mastered in order to have some persons that master very few skills and others that are polyvalent. Add to this, in some instances each skill has the same probability to be mastered, and in some others instances there exist some skills very less often mastered and other mastered by a large part of the staff. So, this global set of instances represents a wide range of instances. Some of them are strongly constrained: there are very few sets of activities that can be scheduled in parallel and other have lots of activities that can be in progress at the same time.

For each generated instance, we exactly know the network complexity (which is between 1.5 and 2.1), but it is not enough to caracterize the difficulty of an instance. After lots of experimentations, it appears that neither the precedence disjunction (percentage of couples of activities that cannot be in progress at the same time due to their precedence relation) nor the resource disjunction (percentage of couples of activities that cannot be in progress at the same time due to the resource constraint) [1] is directly linked to the difficulty to solve an instance.

In order to evaluate the efficency of the Branch-and-Bound method on this problem, different versions of this method have been tested on all the instances. The maximum execution time is limited to 20 min, so there are some unresolved instances. Add to this, as the instances cannot be characterized before execution, some of them are easy to solve, so the B&B method reach the optimal solution at the root node.

The three different versions shown in Table 4 are the three branching strategies explained in Section 5: the ones based on maximum slack activities, stable set activities, and loaded time-interval activities. The deviation is computed as follow: $dev = (bestUB - bestLB)/bestUB$, where $bestLB$ is the best value find by the lower bounds at the root node. This table also shows the average time used to solved an instance. According to this table, we can conclude that the three different branching strategies are almost equivalent in terms of deviation and execution time, but the number of nodes developped in each method is not exactly the same, as it is shown in Table 5. The branching strategy based on the stable set activities

TABLE 5. Comparison between the number of nodes needed.

Branching strat.	Min. #nodes	Av. #nodes	Max. #nodes
Max. slack	0	9358.34	362114
Stable set	0	6909.64	261104
Load. time-int.	0	9227.98	359384

clearly needs less nodes than other ones, but the treatment of each node is much time-consumming than for other methods, because the choice of the activity to branch on is not easily determined.

To conclude, the branching strategy do not have a significative impact on the results: the number of unreasolved instances is comparable, and for those **unreasolved** instances the average deviation does not exceed 7%. So, we can say that this Branch-and-Bound method reaches good results for small size instances up to 32 activities. Some other tests have been made on 120 bigger instances (With 60 or 90 activities) and the number of instances not optimally solved grows up to 118, that means more than 98%. The average deviation got on those unreasolved instances in 20 min of execution grows up to 9.30%.

7. Conclusion

In this paper, we propose a Branch-and-Bound method to solve the Multi-Skill Project Scheduling Problem. We introduce different branching strategies and define the way to treat the assignment problem in each leaf node. Results show that it works on small and average size instances. Most of the big size instances are not optimally solved in 20 minutes but the deviation is acceptable.

Nevertheless the size is not the only element to take into account to evaluate the difficulty of an instance. So, it is one of our future research directions to find out relevant elements to characterize instances, in order to know why for the same size and the same network complexity, some of them are solved at the root node and some other ones are not optimally solved in 20 minutes.

References

[1] Ph. Baptiste, C. Le Pape and W. Nuitjen, Satisfiability tests and time-bound adjustments for cumulative scheduling problems, in *Ann. Oper. Res.* **92** (1999) 305–333.

[2] O. Bellenguez, C. Canon and E. Néron, Ordonnancement des formations des télé-opérateurs dans un centre de contacts clients, in *Proc. ROADEF 2005*, Tours, France (2005).

[3] O. Bellenguez and E. Néron, Méthodes approchées pour le problème de gestion de projet multi-compétence, in *École d'Automne de Recherche Opérationnelle*, TOURS, France (2003) 39–42.

[4] O. Bellenguez and E. Néron, An exact method for solving the multi-skill project scheduling problem, in *Proc. Operations Research (OR2004)*, Tilburg, The Netherlands (2004) 97–98.

[5] O. Bellenguez and E. Néron, Lower bounds for the multi-skill project scheduling problem with hierarchical levels of skills, in *Proc. Practice and Theory of Automated Timetabling (PATAT2004)*, Pittsburgh, PA, USA (2004) 429–432.
[6] O. Bellenguez and E. Néron, Methods for solving the multi-skill project scheduling problem, in *Proc. 9th International Workshop on Project Management and Scheduling (PMS2004)*, Nancy, France (2004) 66–69.
[7] J. Carlier, Scheduling jobs with release dates and tails on identical parallel machines to minimize the makespan. *Eur. J. Oper. Res.* **29** (1987) 298–306.
[8] J. Carlier and B. Latapie, Une méthode arborescente pour résoudre les problèmes cumulatifs. *RAIRO-RO* **25** (1991) 23–38.
[9] S. Hartmann and A. Drexl. Project scheduling with multiple modes: A comparison of exact algorithms. *Networks* **32** (1998) 283–297.
[10] W. Hopp and M. Van Oyen, Agile workforce evaluation: a framework for cross-training and coordination. *IIE Transactions* **36** (2004).
[11] J. Josefowska, M. Mika, R. Rozycki, G. Waligora and J. Weglarz, Simulated annealing for multi-mode resource-constrained project scheduling problem. *Ann. Oper. Res.* **102** (2001) 137–155.
[12] R. Klein, *Scheduling of resource-constrained projects*. Kluwer Academic Publishers (2000).
[13] A.W.J. Kolen and L.G. Kroon, On the computational complexity of (maximum) class scheduling. *Eur. J. Oper. Res.* **54** (1991) 23–28.
[14] R. Kolisch and A. Sprecher, Psplib - a project scheduling problem library. *Eur. J. Oper. Res.* **96** (1997) 205–216.
[15] R. Kolisch and A. Sprecher, Psplib - a project scheduling problem library. pages `ftp://ftp.bwl.uni--kiel.de/pub/operations--research/psplib/html/indes.html`, (2000).
[16] L.G. Kroon, M. Salomon and L.N. Van Wassenhove, Exact and approximation algorithms for the operational fixed interval scheduling problem. *Eur. J. Oper. Res.* **82** (1995) 190–205.
[17] P. Lopez, J. Erschler and P. Esquirol, Ordonnancement de tâches sous contraintes: une approche énergétique. *RAIRO-APII* **26** (1992) 453–481.
[18] A. Mingozzi, V. Maniezzo, S. Riciardelli and L. Bianco, An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science* **44** (1998) 714–729.
[19] E. Neron, Lower bounds for the multi-skill project scheduling problem, in *Proc. 8th International Workshop on Project Management and Scheduling (PMS2002)*, Valencia, Spain (2002) 274–277.
[20] L. Ozdamar, A genetic algorithm approach to a general category project scheduling problem. *IEEE Transactions on Systems, Man, and Cybernetics* **29** to be published.
[21] G. Pepiot, N. Cheikhrouhou and R. Glardon, Modèle de compétence: vers un formalisme, in *Proc. Conférence francophone de MOdélisation et de SIMulation (MOSIM2004)*, Nantes, France (2004) 503–510.
[22] B. De Reyck and W. Herroelen, The multi-mode resource-constrained project scheduling problem with generalized precedence relations. *Eur. J. Oper. Res.* **119** (1999) 538–556.
[23] E. Rolland, R.A. Patterson, K. Ward and B. Dodin, Scheduling differentially-skilled staff to multiple projects with severe deadlines. *Eur. J. Oper. Res.* (2005). Submitted to the special issue of PMS (2004).
[24] A. Sprecher and A. Drexl, Multi-mode resource constrained project scheduling problem by a simple, general and powerful sequencing algorithm. *Eur. J. Oper. Res.* **107** (1998) 431–450.
[25] A. Sprecher, R. Kolisch and A. Drexl, Semi-active, active and non-delay schedules for the resource constrained project scheduling problem. *Eur. J. Oper. Res.* **80** (1995) 94–102.
[26] I. Toroslu, Personnel assignment problem with hierarchical ordering constraints. in *Proc. Personnel assignment Problem with hierarchical ordering constraints* (2003) 493–510.