Insertion techniques for static and dynamic resource constrained project scheduling

Christian Artigues, Philippe Michelon, Stéphane Reusser

Laboratoire d'Informatique d'Avignon,
339, chemin des meinajariés, Agroparc, BP 1228,
84911 Avignon Cedex 9, France
email: christian.artigues@lia.univ-avignon.fr

LIA report 163, revised version, October 2001

Abstract

A flow network model is presented for the static resource-constrained project scheduling problem. Static and dynamic scheduling methods, based on a new polynomial insertion algorithm taking advantage on the flow structure, are proposed. The performed computational experiments on some state-of-the-art problem instances show the potential of this approach.

1 Introduction

This paper considers the resource constrained project scheduling problem (RCPSP) [6] both in static and dynamic environments. Static scheduling lies in determining a solution to a scheduling problem instance, i.e. characterizing the starting times of a known set of activities subject to deterministic resource and precedence constraints. Dynamic scheduling lies in reacting to some changes of the initial problem characteristics that generally occur in real time. This can be done either by performing a global rescheduling each time such a change occurs, or by udapting an initial static schedule to each disruption so as to perform only local modifications on the current schedule.

The disruption considered in this study is the occurence of an unexpected activity.

A polynomial algorithm aiming at inserting a new activity inside an existing solution represented by an Activity-On-Node / flow network has been defined in [1]. This algorithm extends to the RCPSP the concept of dominant insertion positions, previously used for variants of the job-shop problem [15] [5] [2]. In this previous work, $\mathcal{O}(n^3)$ dominant insertion positions were generated, where n is the number of activities in the problem. In this paper, we show that the number of dominant insertion positions can be reduced to $\mathcal{O}(n^2)$ and we propose a new version of the algorithm, which has an $\mathcal{O}(n^2m)$ time complexity. We provide a new rescheduling algorithm for the insertion of an unexpected activity in a dynamic context. We evaluate its robustness in a simulated dynamic environment compared to a priority rule heuristic. For the static case, we propose two heuristics embedding the insertion algorithm. The first one is a simple n-pass constructive method, where n is the number of activities. The second one is a tabu search method which reinserts at each iteration one or several activities. In both cases, we give computational results on state-of-the-art instances which demonstrate the efficiency of this approach compared to the best metaheuristics described in [10] and [14]. Although some directions for solving the static RCPSP with the AON-flow model are given in [8], this paper presents for the first time promising experimental results.

The AON-flow formulation of the static RCPSP is given in Section 2. Section 3, provides a method to generate an AON-flow network solution to this problem. The considered insertion problem and the proposed polynomial algorithm which solves it to optimality are presented in Section 4. We provide a computational analysis of the use of the insertion algorithm for rescheduling in the context of an occurence of an unexpected activity in Section 5. The heuristics derived from the insertion algorithm for the static RCPSP are presented and evaluated in Section 6.

2 The AON-flow network formulation of the static RCPSP

It is assumed that a project made of a set of activities $V = \{1, ..., n\}$ has to be scheduled on a set of renewable resources $\mathcal{R} = \{1, ..., m\}$. Each resource

$k \in \mathcal{R}$ has a finite capacity R_k. Precedence constraints of activities within the project are modeled by a set of project arcs E such that $(i,j) \in E$ means that activity j has to start after the completion of activity i. Each activity $i \in V$ requires a non-negative amount r_{ik} of each resource $k \in \mathcal{R}$ and has a duration p_i. The scheduling problem lies in characterizing a n-uple $S = (S_1, ..., S_n)$ where S_i is the starting time of activity i, while minimizing the total project duration (makespan) denoted by $C_{\max}$. This problem, known as the Resource Constrained Project Scheduling Problem (RCPSP) and denoted $PS|prec|C_{\max}$ [6], will be defined in the remaining by the triple $(V, E, \mathcal{R})$. Its difficulty comes from the resource limitation constraints, preventing some activities requiring the same resource from being scheduled simultaneously. This can be enounced by defining each resource k as the union of R_k resource units (or parallel machines), such that a given resource unit cannot be allocated at the same time to more than one activity. Hence, in any feasible solution, a resource unit allocated to an activity i has to be directly transferred after the completion of i to a unique activity j. However since all the units of the same resource are equivalent, one has only to know the number of units directly transferred from one activity to another. On this basis, a flow network model can be defined for the RCPSP. In this model, variable f_{ijk} denotes the number of units of resource k directly transferred from an activity i to an activity j and binary variable x_{ij} is equal to 1 if activity j is constrained to start after the completion of activity i; x_{ij} is equal to 0 otherwise.

Furthermore $s = 0$ and $t = n + 1$ are two dummy activities such that $r_{sk} = r_{tk} = R_k, \forall k \in \mathcal{R}$ and $p_s = p_t = 0$. M and N are two arbitrary large integers. Under these assumptions, a formulation for the RCPSP which extends the classical mathematical model of the job shop scheduling problem can be written:

$$\min C_{\max} \tag{1}$$

subject to:

$$x_{ij} = 1 \qquad \forall (i,j) \in E \tag{2}$$

$$S_j - S_i - M x_{ij} \geq p_i - M \qquad \forall i \in V \cup \{s\}, \forall j \in V \cup \{t\} \tag{3}$$

$$f_{ijk} - N x_{ij} \leq 0 \qquad \forall i \in V \cup \{s\}, \forall j \in V \cup \{t\}, \forall k \in \mathcal{R} \tag{4}$$

$$\sum_{j \in V \cup \{t\}} f_{ijk} = r_{ik} \qquad \forall i \in V \cup \{s\}, \forall k \in \mathcal{R} \tag{5}$$

$$\sum_{i \in V \cup \{s\}} f_{ijk} = r_{jk} \qquad \forall j \in V \cup \{t\}, \forall k \in \mathcal{R} \tag{6}$$

$$C_{\max} \geq S_i + p_i \qquad \forall i \in V \tag{7}$$

$$f_{ijk} \in \mathrm{IN} \qquad \forall i \in V \cup \{s\}, \forall j \in V \cup \{t\}, \forall k \in \mathcal{R} \tag{8}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in V \cup \{s\}, \forall j \in V \cup \{t\} \tag{9}$$

Equation (1) gives the objective of the scheduling problems. Constraints (2) give the precedence relations within the project. Constraints (3) are disjunctive constraints that prevent two activities linked through a resource unit flow from being scheduled simultaneously. Constraints (4) describe the link between variables f_{ijk} and variables x_{ij}, that is the implication $x_{ij} = 0 \Rightarrow (f_{ijk} = 0, \forall k \in \mathcal{R})$. Constraints (5) and (6) express that the input and output flow of an activity on a resource k must be equal to its required capacity on that resource, i.e. the flow conservation property. Constraints (7) give the expression of the makespan. Constraints (8) and (9) give the domain of variables f_{ijk} and x_{ij}.

A set of feasible solutions of such a problem can be nicely represented by an activity-on-node(AON)/flow network $\mathcal{G}$ (see also [1] and [8] among others). The nodes of this graph represent activities from the set $V \cup \{s,t\}$. Two classes of arcs are defined. "Project" arcs are taken from set E. In addition, a "resource" arc is generated between each two activities such that $\exists k \in \mathcal{R}, f_{ijk} > 0$. Hence, a resource predecessor (successor) of an activity j is an activity i such that there exists a resource arc (i,j) (resp (j,i)). Each (project or resource) arc (i,j) is weighted by p_i and each resource arc (i,j) is associated with a vector capacity $(f_{ij1}, ..., f_{ijm})$. Such a graph, which is an extension of the one proposed in [7] for multiresource jobshops, can be used as an activity-on-node network to compute feasible earliest and latest starting time of all activities. Hence, for each activity $i \in V \cup \{t\}$ a feasible earliest starting time ES_i can be set to the length of the longest path between s and i denoted $l(s,i)$. Such earliest starting times correspond to a feasible left shifted schedule. Similarly a feasible latest starting time LS_i can be set to $l(0,t) - l(i,t)$. Such latest starting times correspond to a feasible right shifted schedule. Note that EF_i corresponds to the earliest finishing time $ES_i + p_i$ and LF_i corresponds to the latest finishing time $LS_i + p_i$.

Furthermore, variable x_{ij} can be set to 1 if there is a path in $\mathcal{G}$ from i to j, and to 0 otherwise. Hence, a family of feasible schedules (between the left shifted one and the right shifted one) are perfectly defined by the flow $\mathcal{F} = (f_{ijk} | i \in V \cup \{s\}, j \in V \cup \{t\}, k \in \mathcal{R})$. The longest path from s to t,

(the critical path), has a length $ES_t = l(0,t)$ equal to the makespan of (all solutions defined by) the flow $\mathcal{F}$. Hence, a given flow $\mathcal{F}$ is feasible if and only if it verifies constraints (5),(6) and if the graph $\mathcal{G}$ it induces does not contain any cycle (which would lie in the non-existence of the above defined longest paths).

As an illustrative example, let us consider an instance of the static RCPSP proposed in [11] characterized by a single resource of capacity 4 and 10 non-dummy activities. This instance is represented by the AON graph displayed in part (a) of Figure 1, where each node represents an activity and each arc represents a project precedence constraint. The first value displayed under each node is the duration of the activity while the second value is its request on the single resource. Part (b) of Figure 1 represents a feasible AON-flow network for this instance. Bolded arcs represent non-null flow units and the value displayed near each bolded arc (i,j) is the amount of transferred units f_{ijk}. Furthermore, the values displayed here under each node i are the precedence and resource feasible earliest and latest starting times $ES_i..LS_i$ induced by the flow. Thus, besides the more precise characterization of resource usage, the flow model gives a representation of a family of feasible schedules between the left shifted one (represented in part (c) of Figure 1) and the right shifted one.

3 Extension of the parallel schedule generation scheme to flow generation

In this Section, we describe a simple method to obtain a feasible flow for the static RCPSP by extending the parallel schedule generation scheme (SGS). A SGS lies in building a feasible schedule by stepwise extension of a partial schedule, starting from scratch. We refer to [14] for a detailed description of the algorithm. We cite [10] for a brief description:

> The parallel SGS does time incrementation. At each iteration g, there is a schedule time t_g and a set of eligible activities. An activity is eligible if it can be precedence-and resource-feasibly started at the schedule time. Activities are chosen from the eligible set and started at the schedule time until there are no more eligible activities left. Afterwards, the scheduling scheme steps to

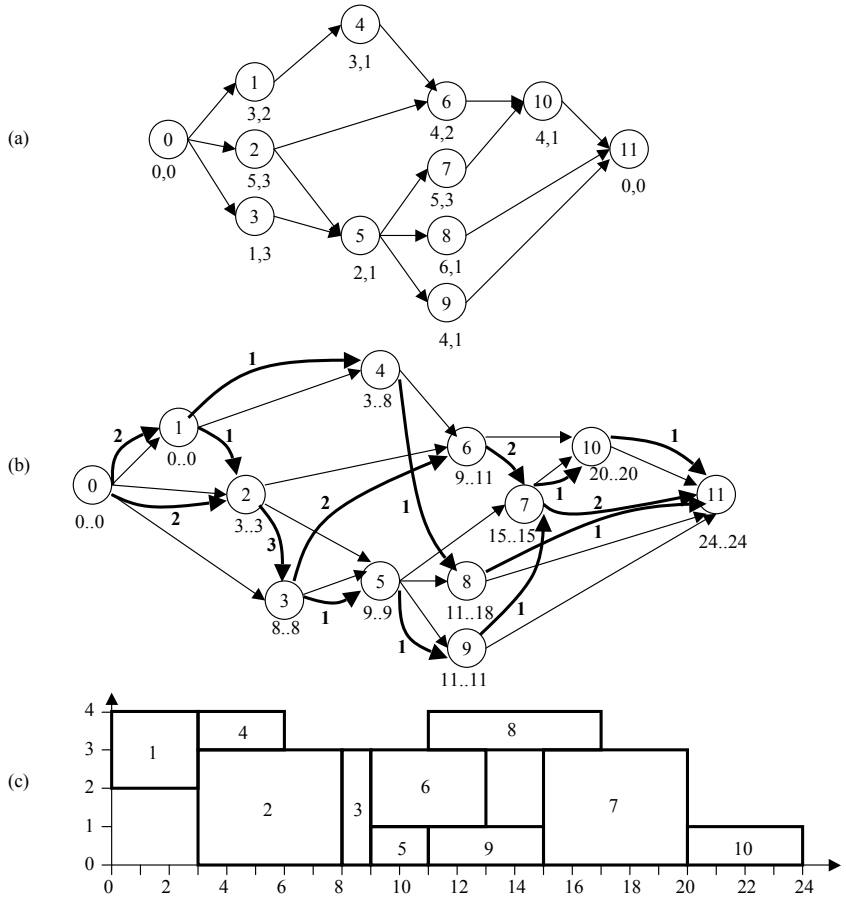

Figure 1: Example of an AON network for a static RCPSP

the next schedule time which is given by the earliest finish time of the activities in progress.

The generation of a feasible flow $\mathcal{F}$ is integrated to the parallel SGS as follows. $\mathcal{F}$ is initialized so that the source activity s sends directely all its resource units to the dummy sink activity t. Then, each time an activity j is selected for being scheduled by the parallel SGS, the flow is updated by taking the units that j requires from the input flow of activity t and by sending back all those units from j to t. The initialization and updating algorithm are detailed below. InitFlow has an $\mathcal{O}(n^2m)$ time complexity whereas UpdateFlow has an $\mathcal{O}(nm)$ time complexity.

Algorithm InitFlow($\mathcal{F}$)
1. $\forall k \in \mathcal{R}, f_{stk} := R_k$
2. $\forall k \in \mathcal{R}, \forall i,j \in V \cup \{s,t\}, (i,j) \neq (s,t), f_{ijk} := 0$
end.

Algorithm UpdateFlow($j, \mathcal{F}$)
$req(k)$ is the number of units still required by j
1. $\forall k \in \mathcal{R}$, $req(k) := r_{jk}$
2. For $i \in V \cup \{s\}$ do
3. For $k \in \mathcal{R}$ do
4. $q := \min(req[k], f_{itk})$
5. $req(k) := req(k) - q$
6. $f_{itk} := f_{itk} - q$
7. $f_{ijk} := f_{ijk} + q$
8. end For
9. End For
10. $\forall k \in \mathcal{R}$, $f_{jtk} = r_{jk}$
end.

The overall algorithm, denoted GenFlow generates a feasible flow since the parallel SGS inserts the selected activity "at the end" of the partial schedule, i.e. at a time point where all scheduled activities are all already started. In this simple version, scheduled activities candidate for sending flow to the selected activity j are listed in the lexicographic order. Note that integrating this flow computation does not increase the time complexity of the parallel SGS which remains $\mathcal{O}(n^2m)$ [14].

Note that the AON-flow network of Figure 1 has been generated by our extension of the parallel SGS with the MINLFT priority rule (i.e. the Minimum Latest Finishing Time with respect to project precedence constraints only). For the considered instance, a makespan value of 24 is obtained, which is not optimal.

4 An $\mathcal{O}(n^2m)$ insertion algorithm

Both static and dynamic scheduling methods proposed here make use of a polynomial insertion algorithm denoted INS. INS takes as input a feasible solution of an initial problem represented by the AON-flow network and an activity to insert in the schedule. To find an optimal insertion position INS uses both arc capacities and nodes earliest and latest starting times. It outputs a new feasible network minimizing makespan increase while including the new generated node, with the additional constraint to keep unchanged the previously defined resource precedence constraints. This solution is obtained by exploration of a set of dominant cuts of maximal capacity in the flow-network. In our previous work [1] we had exhibited $\mathcal{O}(n^3)$ dominant insertion positions, where n is the number of activities in the problem. In this section, we show that the number of dominant insertion positions can be reduced to $\mathcal{O}(n^2)$ and we propose a new version of the algorithm, which has an $\mathcal{O}(n^2m)$ time complexity.

4.1 Insertion Problem formulation

We consider the problem of inserting a new activity x defined by a duration p_x, a vector request r_x, a set of project predecessors V_x^- and a set of project successors V_x^+ inside an existing flow $\mathcal{F}$ for a RCPSP $(V, E, \mathcal{R})$ while minimizing the makespan $C'_{\max}$ of the resulting flow. Such an insertion has to be performed only by making some of the resource units initially transferred from an activity i to an activity j to be transferred from i to x and then from x to j. Such constraints, which keep all previously defined resource precedence constraints, reduce considerably the set of possible solutions. Hence an insertion position is unambiguously defined by the values of subtracted resource units $\mathcal{Q} = (q_{ijk} | i \in V \cup \{s\}; j \in V \cup \{t\}; k \in \mathcal{R})$. Such an insertion position $\mathcal{Q}$ is feasible if and only if the flow $\mathcal{F}'$ defined by

$$\begin{cases} f'_{ixk} = \sum\limits_{j \in V \cup \{t\}} q_{ijk} & \forall i \in V \cup \{s\}, \forall k \in \mathcal{R} \\ f'_{xik} = \sum\limits_{j \in V \cup \{s\}} q_{jik} & \forall i \in V \cup \{t\}, \forall k \in \mathcal{R} \\ f'_{ijk} = f_{ijk} - q_{ijk} & \forall i \in V \cup \{s\}, \forall j \in V \cup \{t\}, \forall k \in \mathcal{R} \end{cases} \quad (10)$$

is feasible for the resource constrained scheduling problem $(V', E', \mathcal{R})$ where $V' = V \cup \{x\}$, $E' = E \cup \{(i,x) | i \in V_x^-\} \cup \{(x,i) | i \in V_x^+\}$. In the remaining, data or variables of the initial problem or solution will be denoted without a $(')$, such as $C_{\max}$, whereas data and variables of the new problem or solution will be denoted with a $(')$, such as $C'_{\max}$.

4.2 Evaluation of an insertion position

Given a feasible insertion position $\mathcal{Q}$, the makespan increase it induces can be computed as follows. $\mathcal{Q}$ defines an activity set $\mathcal{P}(\mathcal{Q})$ corresponding to the resource predecessors of x after insertion and an activity set $\mathcal{S}(\mathcal{Q})$ corresponding to the resource successors of x after insertion:

$$\mathcal{P}(\mathcal{Q}) = \{i \in V \cup \{s\} | \exists j \in V \cup \{t\}, \exists k \in \mathcal{R}, q_{ijk} > 0\} \quad (11)$$

$$\mathcal{S}(\mathcal{Q}) = \{i \in V \cup \{t\} | \exists j \in V \cup \{s\}, \exists k \in \mathcal{R}, q_{jik} > 0\} \quad (12)$$

Given a set of activities $\mathcal{P}$, let $ES_x(\mathcal{P})$ denote the new earliest starting time of x when inserted after all activities in $V_x^- \cup \mathcal{P}$. Symetrically, given a set of activities $\mathcal{S}$, let $LF_x(\mathcal{S})$ denote the latest finishing time of x when inserted before all activities in $V_x^+ \cup \mathcal{S}$ (with respect to the current makespan):

$$ES_x(\mathcal{P}) = \max_{i \in V_x^- \cup \mathcal{P}} (EF_i) \quad (13)$$

$$LF_x(\mathcal{S}) = \min_{i \in V_x^+ \cup \mathcal{S}} (LS_i) \quad (14)$$

Then, the makespan increase $\delta C_{\max}(\mathcal{Q})$ induced by an insertion characterized by a feasible insertion position $\mathcal{Q}$ is equal to:

$$\delta C_{\max}(\mathcal{Q}) = C'_{\max} - C_{\max} = \max\{0, ES_x(\mathcal{P}(\mathcal{Q})) + p_x - LF_x(\mathcal{S}(\mathcal{Q}))\} \quad (15)$$

It is noticeable that the increase of makespan is determined only by resource predecessors and successors whatever may be the number of units transferred from a given resource successor to x, or from x to a given resource successor.

4.3 Conditions for insertion position feasibility

An insertion position $\mathcal{Q}$ is feasible if and only if the flow $\mathcal{F}'$ defined by equations 10 is a feasible solution of the RCPSP $(V', E', \mathcal{R})$. Necessary and sufficent feasibility conditions can be stated as follows.

Proposition 1 *An insertion position $\mathcal{Q}$ is feasible if and only if:*

1. $\forall k \in \mathcal{R}, \sum_{i \in V \cup \{s\}; j \in V \cup \{t\}} q_{ijk} = r_{xk}$
2. *there is no path in graph $\mathcal{G}$ from $V_x^+ \cup \mathcal{S}$ to $V_x^- \cup \mathcal{P}$.*

Proof. This proposition is trivial according to definition of $\mathcal{F}'$ (equations 10) and considering the feasibility condition of a flow (see Section 2). Indeed, condition 1 ensures the conservation property (5),(6) of $\mathcal{F}'$ while condition 2 ensures the acyclic property of the induced graph $\mathcal{G}'$. ■

Since the makespan increase $\delta C_{\max}(\mathcal{Q})$ does not depend on the number of transferred units but rather on the resource predecessors and successors, we introduce in the next Subsection the notion of sufficient insertion position which ensures feasibility without computing *a priori* amounts (q_{ijk}).

4.4 Sufficient insertion positions and insertion cuts

Let $\mathcal{P}$ and $\mathcal{S}$ be two disjoint activity sets. Given $\mathcal{P}$, $\mathcal{S}$, the initial flow $\mathcal{F}$ and the characteristics of the activity to be inserted x, the following algorithm inserts the new activity x at an insertion position $\mathcal{Q}$ deduced from $(\mathcal{P}, \mathcal{S})$ and generates the new flow $\mathcal{F}'$. For each activity $i \in \mathcal{P}$, for each activity $j \in \mathcal{S}$ and for each resource $k \in \mathcal{R}$, q_{ijk} is set to the minimum between the existing flow f_{ijk} and the remaining request $req(k)$ of the activity, which is updated at each iteration. Algorithm INSERT detailed below has an $\mathcal{O}(n^2 m)$ complexity.

Algorithm INSERT$(\mathcal{P}, \mathcal{S}, x, \mathcal{F}')$
1. $\forall i \in V \cup \{s, x\}$, $\forall j \in V \cup \{x, t\}$, $\forall k \in \mathcal{R}$, $q_{ijk} := 0$; $f'_{ijk} := f_{ijk}$
2. $\forall k \in \mathcal{R}$, $req(k) := r_{xk}$
2. For each $i \in \mathcal{P}$
3. For each $j \in \mathcal{S}$
4. $\forall k \in \mathcal{R}$, $q_{ijk} = \min(req(k), f_{ijk})$
5. $req(k) := req(k) - q_{ijk}$

6. $f'_{ijk} := f'_{ijk} - q_{ijk}$; $f'_{ixk} := f'_{ixk} + q_{ijk}$; $f'_{xjk} := f'_{xjk} + q_{ijk}$
6. End For
7. End For
End INSERT

The following proposition gives a sufficient condition for the feasibility of the flow generated by INSERT.

Proposition 2 *Given a pair ($\mathcal{P}$,$\mathcal{S}$) of activity sets, the insertion position $\mathcal{Q}$ computed by algorithm* INSERT *is feasible if :*

1. $\forall k \in \mathcal{R}$, $\sum_{i \in \mathcal{P}, j \in \mathcal{S}} f_{ijk} \geq r_{xk}$

2. *there is no path in graph $\mathcal{G}$ from $V_x^+ \cup \mathcal{S}$ to $V_x^- \cup \mathcal{P}$.*

Proof. This proposition is directly deduced from definition of $\mathcal{Q}$ inside algorithm INSERT and proposition 1. ■

A pair of activity sets $(\mathcal{P}, \mathcal{S})$ verifying conditions 1 and 2 of proposition 2 is called a *sufficient insertion position*. Its *capacity* is the vector $(\sum_{i \in \mathcal{P}, j \in \mathcal{S}} f_{ijk} | k \in \mathcal{R})$. The interest is to characterize a feasible insertion position only by a pair of activity sets (provided that the pair verifies conditions 1 and 2 of proposition 2) without explicitely computing $\mathcal{Q}$. Furthermore, any sufficient insertion position $(\mathcal{P}, \mathcal{S})$ provides an upper bound of makespan increase : let $\delta C_{\max}(\mathcal{P}, \mathcal{S})$ be the expression obtained by replacing $\mathcal{P}(\mathcal{Q})$ by $\mathcal{P}$ and $\mathcal{S}(\mathcal{Q})$ by $\mathcal{S}$ in equation 15. Obviously, since $\mathcal{P}(\mathcal{Q}) \subseteq \mathcal{P}$ and $\mathcal{S}(\mathcal{Q}) \subseteq \mathcal{S}$ the feasible insertion position $\mathcal{Q}$ derived from $(\mathcal{P}, \mathcal{S})$ by algorithm INSERT verifies $\delta C_{\max}(\mathcal{Q}) \leq \delta C_{\max}(\mathcal{P}, \mathcal{S})$.

Two kinds of sufficient insertion position are of particular interest : the *maximal* insertion positions (or insertion cuts) on one hand, and the *minimal* insertion positions on the other hand. A maximal insertion position or insertion cut $(\mathcal{P}_{\max}, \mathcal{S}_{\max})$ is a sufficient insertion position verifying :

$$\forall k \in \mathcal{R}, \sum_{i \in \mathcal{P}_{\max}, j \in \mathcal{S}_{\max}} f_{ijk} = R_k \tag{16}$$

Such a cut covers all the resource units and is maximal according to the inclusion: there is no other sufficient insertion position $(\mathcal{P}, \mathcal{S}) \neq (\mathcal{P}_{\max}, \mathcal{S}_{\max})$

such that $\mathcal{P}_{\max} \subseteq \mathcal{P}$ and $\mathcal{S}_{\max} \subseteq \mathcal{S}$. In particular, condition 16 is verified if $\mathcal{P}_{\max} \cup \mathcal{S}_{\max} = V \cup \{s,t\}$ which justifies the word "cut".

On the opposite, a minimal insertion position $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$ is a sufficient insertion position verifying :

$$\begin{cases} \forall i \in \mathcal{P}_{\min}, \exists k \in \mathcal{R}, \sum\limits_{u \in \mathcal{P}_{\min} \setminus \{i\}, j \in \mathcal{S}_{\min}} f_{ujk} < r_{xk} \\ \forall j \in \mathcal{S}_{\min}, \exists k \in \mathcal{R}, \sum\limits_{i \in \mathcal{P}_{\min}, v \in \mathcal{S}_{\min} \setminus \{j\}} f_{ivk} < r_{xk} \end{cases} \tag{17}$$

Such an insertion position is minimal according to the inclusion: there is no other sufficient insertion position $(\mathcal{P}, \mathcal{S}) \neq (\mathcal{P}_{\min}, \mathcal{S}_{\min})$ such that $\mathcal{P} \subseteq \mathcal{P}_{\min}$ and $\mathcal{S} \subseteq \mathcal{S}_{\min}$. It has the remarkable property that if $\mathcal{Q}_{\min}$ denotes the insertion position computed by INSERT from $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$ then $\delta C_{\max}(\mathcal{P}(\mathcal{Q}_{\min}), \mathcal{S}(\mathcal{Q}_{\min})) = \delta C_{\max}(\mathcal{P}_{\min}, \mathcal{S}_{\min})$.

Proposition 3 *For any feasible insertion position $\mathcal{Q}$ there exists a sufficient insertion position $(\mathcal{P}, \mathcal{S})$ such that $\delta C_{\max}(\mathcal{Q}) = \delta C_{\max}(\mathcal{P}, \mathcal{S})$*

Proof. From a given feasible insertion position $\mathcal{Q}$, a sufficient insertion position $(\mathcal{P}, \mathcal{S})$ can be obviously be derived by setting $\mathcal{P} := \mathcal{P}(\mathcal{Q})$ and $\mathcal{S} := \mathcal{S}(\mathcal{Q})$. ■

From propositions 2 and 3, it follows that the search of a feasible insertion position $\mathcal{Q}^*$ minimizing makespan increase is equivalent to the search of a sufficient insertion position $(\mathcal{P}^*, \mathcal{S}^*)$ of minimal makespan increase.

4.5 Dominance rule

A sufficient insertion position $(\mathcal{P}, \mathcal{S})$ is said to be dominant compared to a minimal insertion position $(\mathcal{P}_{\min}, \mathcal{S}_{min})$ if $\delta C_{\max}(\mathcal{P}, \mathcal{S}) \leq \delta C_{\max}(\mathcal{P}_{\min}, \mathcal{S}_{\min})$. The time complexity of the proposed algorithm is obtained thanks to the following dominance rule :

Proposition 4 *Let i and j denote two activities of the initial project. A sufficient insertion position $(\mathcal{P}, \mathcal{S})$ such that*

$$ES_x(\mathcal{P}) \leq \max_{u \in V_x^- \cup \{i\}} (EF_u) \tag{18}$$

$$LF_x(\mathcal{S}) \geq \min_{u \in V_x^+ \cup \{j\}} (LS_v) \tag{19}$$

is dominant compared to any other minimal insertion position $(\mathcal{P}_{\min}, \mathcal{S}_{min})$ such that $i \in \mathcal{P}_{min}$ and $j \in \mathcal{S}_{min}$.

Proof. By inserting equations 18 and 19 into the expression of makespan increase 15, it comes: $\delta C_{\max}(\mathcal{P},\mathcal{S}) \leq \delta C_{\max}(\mathcal{P}_{\min},\mathcal{S}_{\min})$ ■

To illustrate the concept of dominance, let us consider that an activity of request 4 has to be inserted in the solution displayed in Figure 1. $\{(0,1),(2,3,4,5,6,7,8,9,10,11)\}$ is a maximal and minimal insertion position. The reader can check that it dominates any other insertion position $(\mathcal{P},\mathcal{S})$ such that $1 \in \mathcal{P}$ and $2 \in \mathcal{S}$ because $\{(0,1),(2,3,4,5,6,7,8,9,10,11)\}$ verifies (18) and (19) for $i=1$ and $j=2$.

4.6 Generation of dominant insertion positions

4.6.1 Overview of the algorithm and proof of optimality

The proposed algorithm aims at generating a set of dominant sufficient insertion positions, i.e. including at least an optimal one. First, let $(\mathcal{P}^0,\mathcal{S}^0),...,(\mathcal{P}^\Delta,\ \mathcal{S}^\Delta)$ denote a series of cuts in graph $\mathcal{G}$ where $\mathcal{P}^\alpha = (V \cup \{s,t\}) \setminus \mathcal{S}^\alpha$ and $\mathcal{S}^\alpha$ is defined by the following recursion :

$$\begin{cases} \mathcal{S}^0 = \{t\} \cup \{i \in V | EF_i > ES_x^-\} \\ \mathcal{S}^\alpha = \mathcal{S}^{\alpha-1} \setminus \{j^{\alpha-1}\} \qquad \alpha = 1,...,\Delta \end{cases} \tag{20}$$

where

- $j^\alpha \in \mathcal{S}^\alpha$ is an activity verifying: $LS_{j^\alpha} = \min\{LS_j | j \in \mathcal{S}^\alpha\}$.
- $\Delta \geq 0$ is the index of the first encountered cut verifying $LS_{j^\Delta} \geq LF_x^+$.
- ES_x^- is the earliest starting time of activity x considering only its project predecessors:
$$ES_x^- = \max_{i \in V_x^-}(EF_i) \tag{21}$$
- LF_x^+ is the latest starting time of activity x with respect to the current makespan, considering only its project successors:
$$LF_x^+ = \min_{j \in V_x^+}(LS_j) \tag{22}$$

Then, for each considered cut $(\mathcal{P}^\alpha,\mathcal{S}^\alpha)$, let us define the finite series of subcuts $(\mathcal{P}^{\alpha,0},\mathcal{S}^\alpha),...,(\ \mathcal{P}^{\alpha,\Phi_\alpha},\mathcal{S}^\alpha)$ by the following recursion:

$$\begin{cases} \mathcal{P}^{\alpha,0} = \mathcal{P}^\alpha \\ \mathcal{P}^{\alpha,\beta} = \mathcal{P}^{\alpha,\beta-1} \setminus \{i^{\alpha,\beta-1}\} \quad \beta = 1,...,\Phi_\alpha \end{cases} \tag{23}$$

where

- $i^{\alpha,\beta} \in \mathcal{P}^{\alpha,\beta}$ is an activity verifying $EF_{i^{\alpha,\beta}} = \max\{EF_i | i \in \mathcal{P}^{\alpha,\beta}\}$
- $\Phi_\alpha \geq 0$ is the index of the first encountered subcut verifying either

$$EF_{i^{\alpha,\Phi_\alpha}} \leq ES_x^- \tag{24}$$

or

$$\exists k \in \mathcal{R}, \sum_{i \in \mathcal{P}^{\alpha,\Phi_\alpha} \setminus \{i^{\alpha,\Phi_\alpha}\}, j \in \mathcal{S}^\alpha} f_{ijk} < r_{xk}. \tag{25}$$

Proposition 5 *Each element of the set* $\{(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^\alpha) | \alpha = 0, ..., \Delta; \beta = \Phi_0, ..., \Phi_\Delta\}$ *is a sufficient insertion position.*

Proof. First, let us demonstrate that $(\mathcal{P}^\alpha, \mathcal{S}^\alpha)$, $\alpha = 0, ..\Delta$ is a series of insertion cuts, i.e. a series of maximal insertion positions. $(\mathcal{P}^0, \mathcal{S}^0)$ has a maximal capacity since in any path made of resource arcs in $\mathcal{G}$, one activity (at least s) belongs to $\mathcal{P}^0$ and one activity (at least t) belongs to $\mathcal{S}^0$. Furthermore, the time properties 20 defining $\mathcal{P}^0$ and $\mathcal{S}^0$ are sufficient conditions for the validity of condition 2 of proposition 2 (i.e. no cycling conditions). Hence $(\mathcal{P}^0, \mathcal{S}^0)$ is an insertion cut. By a recursion argument, let us assume that $(\mathcal{P}^\alpha, \mathcal{S}^\alpha)$ is an insertion cut. Then, $\mathcal{P}^{\alpha+1} = \mathcal{P}^\alpha \cup \{j^\alpha\}$ and $\mathcal{S}^{\alpha+1} = \mathcal{S}^\alpha \setminus \{j^\alpha\}$ implies that $(\mathcal{P}^{\alpha+1}, \mathcal{S}^{\alpha+1})$ has the same capacity as $(\mathcal{P}^\alpha, \mathcal{S}^\alpha)$ since the flow conservation holds. Furthermore, since $LS_{j^\alpha} < LF_x^+$ and j^α has the smallest latest starting time among activities of $\mathcal{P}^\alpha$, the no cycling condition of proposition 2 is still verified and $(\mathcal{P}^{\alpha+1}, \mathcal{S}^{\alpha+1})$ is an insertion cut.

Second, each subcut $(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^\alpha)$, $\beta = 0, ..., \Phi_\alpha$ has a sufficient capacity as stated by condition 25. $(\mathcal{P}^\alpha, \mathcal{S}^\alpha)$ being an insertion cut, $(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^\alpha)$ verifies the no cycling condition of proposition 2 since $\mathcal{P}^{\alpha,\beta} \subset \mathcal{P}^\alpha$. Hence, $(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^\alpha)$ is a sufficient insertion position. ■

The following proposition allows us to discard any activity of the set $\mathcal{P}^0$ as a possible successor and any activity of the set $\mathcal{S}^\Delta$ as a possible predecessor.

Proposition 6 *Any minimal insertion position* $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$ *such that* $(\mathcal{S}_{\min} \cap \mathcal{P}^0) \neq \emptyset$ *(resp.* $(\mathcal{P}_{\min} \cap \mathcal{S}^\Delta) \neq \emptyset$*) is dominated by a sufficient insertion position* $(\mathcal{P}, \mathcal{S})$ *such that* $(\mathcal{S}_{\min} \cap \mathcal{P}^0) = \emptyset$*(resp.* $(\mathcal{P}_{\min} \cap \mathcal{S}^\Delta) = \emptyset$*).*

Proof. Only the demontration allowing to discard activities of $\mathcal{P}^0$ as possible successors is provided, the other demonstration being symetric. Suppose that set $\mathcal{S}_{\min}$ is the union of two disjoint sets $\mathcal{S}_{\min,1}$ and $\mathcal{S}_{\min,2}$ such that $\mathcal{S}_{\min,1} \subseteq \mathcal{P}^0$ and $\mathcal{S}_{\min,2} \subseteq \mathcal{S}^0$. Let us construct a predecessor set $\mathcal{P}$ and a successor $\mathcal{S}$ as follows. $\mathcal{P}$ is initially set to $\mathcal{P}_{\min}$. Let us now define a successor subset $\mathcal{S}_s$ initially set to $\mathcal{S}_{\min,1}$ and let us modify $\mathcal{P}$ and $\mathcal{S}_s$ as follows. In $\mathcal{S}_s$, let i denote the activity with the smallest EF. Let us remove i from $\mathcal{S}_s$ and let us add in $\mathcal{S}_s$ any activity j which receives at least one unit of flow from i. In addition let us add i to $\mathcal{P}$. Let $\mathcal{S} = \mathcal{S}_{\min,1} \cup \mathcal{S}_s$. From flow conservation property on one hand and from the time properties of $\mathcal{P}^0$ on the other hand, it can easily be demonstrated that $(\mathcal{P}, \mathcal{S})$ is a sufficient insertion position verifying $\delta C_{\max}(\mathcal{P}, \mathcal{S}) \leq \delta C_{\max}(\mathcal{P}_{\min}, \mathcal{S}_{\min})$. We can iterate the modification process of $\mathcal{P}$ and $\mathcal{S}_s$ until $\mathcal{S}_s \subseteq \mathcal{S}^0$, which ends the demonstration. ■

The following proposition claims the generated insertion position are dominant with respect to the considered constraints.

Proposition 7 *The set of insertion positions* $\{(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^{\alpha}) | \alpha = 0, ..., \Delta; \beta = \Phi_0, ..., \Phi_\Delta\}$ *contains an optimal insertion position.*

Proof. Let us demonstrate with a recursion argument that at an iteration $\gamma \in \{0, ..., \Delta\}$, one of the generated sufficient insertion positions $\{(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^{\alpha}) | \alpha = 0, ..., \gamma; \beta = 0, ..., \Phi_\alpha\}$ dominates any minimal insertion position $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$ such that $LF_x(\mathcal{S}_{\min}) \leq LF_x(\mathcal{S}^{\alpha})$. First, if $\gamma = 0$, then we have $ES_x(\mathcal{P}^0) = ES_x^-$ and $\delta C_{\max}(\mathcal{P}^0, \mathcal{S}^0) = \max(0, ES_x^- + p_x - LS_{j^0})$ which implies that no lower makespan increase can be obtained with $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$ verifying $LF_x(\mathcal{S}_{\min}) \leq LF_x(\mathcal{S}^0) = LS_{j^0}$. Hence the proposition is verified at step 0 of the recursion.

Suppose now that the proposition holds at step $\gamma - 1$, for $\gamma \geq 1$. Can we find a sufficient insertion position such that $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$ such that $LF_x(\mathcal{S}_{\min}) \leq LF_x(\mathcal{S}^{\gamma})$ not dominated by a sufficient insertion position of the set $\{(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^{\alpha}) | \alpha = 0, ..., \gamma; \beta = 0, ..., \Phi_\alpha\}$? For such a sufficient insertion position $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$, we have either $\mathcal{S}_{\min} \subseteq \mathcal{S}^{\gamma}$ or $\mathcal{S}_{\min} \not\subseteq \mathcal{S}^{\gamma}$. Symetrically we have either $\mathcal{P}_{\min} \subseteq \mathcal{P}^{\gamma}$ or $\mathcal{P}_{\min} \not\subseteq \mathcal{P}^{\gamma}$.

If $\mathcal{S}_{\min} \not\subseteq \mathcal{S}^{\gamma}$ then $\mathcal{S}_{\min}$ contains some activities from a set $\mathcal{S}^{\alpha}$ with $\alpha < \gamma$ (since activities from $\mathcal{P}^0$ have been discarded as possible sucessors in proposition 6). In that case $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$ is dominated since $LF_x(\mathcal{S}_{\min}) <= LF_x(\mathcal{S}^{\alpha})$. Hence one can assume that $\mathcal{S}_{\min} \subseteq \mathcal{S}^{\gamma}$. If $\mathcal{P}_{\min} \not\subseteq \mathcal{P}^{\gamma}$ then $\mathcal{P}_{\min}$ contains activities from $\mathcal{S}^{\gamma}$. With a similar argument as in proposition 6, it

can be shown that a subcut of $(\mathcal{P}^\gamma, \mathcal{S}^\gamma)$ dominates $(\mathcal{P}_{\min}, \mathcal{S}_{\min})$. Hence one can assume that $\mathcal{P}_{\min} \subseteq \mathcal{P}^\gamma$ and $\mathcal{S}_{\min} \subseteq \mathcal{S}^\gamma$. In that case, the only way to decrease $\delta C_{\max}(\mathcal{P}^\gamma, \mathcal{S}^\gamma)$ is to decrease $EF_x(\mathcal{P}^\gamma)$. This can only be done by generating subcuts $(\mathcal{P}^{\gamma,0}, \mathcal{S}^\gamma), ..., (\mathcal{P}^{\gamma,\Phi_\gamma}, \mathcal{S}^\gamma)$. ■

As one can notice, there are $\mathcal{O}(n^2)$ dominant sufficient insertion positions. In the next Subsection we give some details of the algorithm which allows to generate and evaluate them all in $\mathcal{O}(n^2 m)$ time. Furthermore, an example of generation of dominant insertion positions is given in Section 6.1.

4.6.2 Practical implementation of the algorithm

To explore efficiently the set of possible predecessors and successors, we represent those sets implicitely by a matrix *cap* and 2 vectors n_+ and n_-. $cap(i,k)$ is the number of resource k units $(\forall k \in \mathcal{R})$ that an activity $i \in V \cup \{s,t\}$ gives to the current sufficient insertion position $(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^\alpha)$. $n_+(i)$ is the number of resource successors of an activity $i \in V \cup \{s\}$ that belong to $\mathcal{S}^\alpha$. $n_-(i)$ is the number of resource predecessors of an activity $i \in V \cup \{t\}$ that belong to $\mathcal{P}^{\alpha,\beta}$.

Depending on activity i and current insertion position $(\mathcal{P}^{\alpha,\beta}, \mathcal{S}^\alpha)$, the following 3 cases have to be considered.

- if $n_+(i) > 0$ (which implies that $n_-(i) = 0$) then $i \in \mathcal{P}^{\alpha,\beta}$, $n_+(i) = |\{j \in \mathcal{S}^\alpha | \exists k \in \mathcal{R}, f_{ijk} > 0\}|$ and $cap(i,k) = \sum_{j \in \mathcal{S}^\alpha} f_{ijk}$ is the number of k units directly issued from i that are candidate to be allocated to the inserted activity, i.e. an upper bound for f'_{ixk} if the current cut is selected;

- if $n_-(i) > 0$ (which implies that $n_+(i) = 0$) then $i \in \mathcal{S}^\alpha$, $n_-(i) = |\{j \in \mathcal{P}^{\alpha,\beta} | \exists k \in \mathcal{R}, f_{jik} > 0\}|$ and $cap(i,k)$ is the number of k units directly received by i that are candidate to be allocated to the inserted activity, i.e. an upper bound for f'_{xik} if the current cut is selected;

- if $n_+(i) = 0$ and $n_-(i) = 0$ then either
  - $i \notin \mathcal{S}^\alpha$ and $i \notin \mathcal{P}^{\alpha,\beta}$,
  - or $i \in \mathcal{P}^{\alpha,\beta}$ and i does not send any flow to an activity of $\mathcal{S}^\alpha$,
  - or $i \in \mathcal{S}^\alpha$ and i does not receive any flow to an activity of $\mathcal{P}^{\alpha,\beta}$.

We assume that resource arcs of graph $\mathcal{G}$ is are coded by a matrix $\mathcal{Y} = (y_{ij})$ i.e., $y_{ij} = 1$ if and only if $\exists k \in \mathcal{R}, f_{ijk} > 0$ and $y_{ij} = 0$ otherwise.

By using this representation, one can easily check the participation of a given activity to the current insertion position. Furthermore, the transfer of an activity i from $\mathcal{S}^\alpha$ to $\mathcal{P}^\alpha$ in a current insertion cut $(\mathcal{P}^\alpha, \mathcal{S}^\alpha)$ can be done in $\mathcal{O}(nm)$ by algorithm TRANSFERFROM$\mathcal{S}$TO$\mathcal{P}$ described below. When an activity i is transferred from $\mathcal{S}$ to $\mathcal{P}$ its participation to the capacity of the cut does not change but its number of predecessors and succesors are updated (step 1). The participation of its resource predecessors must be decreased (step 2) and the participation of its resource succesors must be increased (step 3) accordingly.

Algorithm TRANSFERFROM$\mathcal{S}$TO$\mathcal{P}(i, cap, n_+, n_-)$
1. $n_-(i) = 0$; $n_+(i) = \sum_{j \in V \cup \{t\}} y_{ij}$
2. $\forall j \in V \cup \{s\}$, $n_+(j) := n_+(j) - y_{ji}$; $\forall k \in \mathcal{R}$, $cap(j,k) := cap(j,k) - f_{jik}$
3. $\forall j \in V \cup \{t\}$, $n_-(j) := n_-(j) + y_{ij}$; $\forall k \in \mathcal{R}$, $cap(j,k) := cap(j,k) + f_{ijk}$
end.

The initial dominant cut $(\mathcal{P}^0, \mathcal{S}^0)$ is generated by algorithm GENERATEINITIALDOMINANTCUT described below. Steps 1,2 and 3 initialise $\mathcal{P}^0$ to $\{s\}$ and $\mathcal{S}'$ to $V \cup \{t\}$. The next steps transfer the activity of $\mathcal{S}^0$ having the smallest EF from $\mathcal{S}^0$ to $\mathcal{P}^0$ until the stop condition holds, making the current cut correspond to the initialisation of recursion 20.GENERATEINITIALDOMINANTCUT has an $\mathcal{O}(n^2 m)$ time complexity.

Algorithm GENERATEINITIALDOMINANTCUT(cap, n^+, n^-)
1. $\forall k \in \mathcal{R}$, $cap(s,k) := R_k$; $\forall i \in V \cup \{t\}$, $cap(i,k) := f_{sik}$
2. $n^-(s) = 0$; $\forall i \in V \cup \{t\}$, $n^-(i) := y_{si}$
3. $n^+(s) = \sum_{i \in V \cup \{t\}} y_{si}$; $\forall i \in V \cup \{t\}$, $n^+(i) = 0$
4. Sort activities of $V \cup \{t\}$ in non decreasing order of EF_i's in a list $\mathcal{L}$.
5. $i :=$ next element of $\mathcal{L}$.
6. while $i \neq t$ and $EF_i \leq EF_x^-$ do
7. TRANSFERFROM$\mathcal{S}$TO$\mathcal{P}(i, cap, n^+, n^-)$
8. $i :=$ next element of $\mathcal{L}$.
9. End While.
End. end.

The dominant insertion cuts and their dominant subcuts are generated by algorithm GENERATEDOMINANTINSERTIONPOSITIONS described below, which outputs the optimal makespan increase $\delta C^*_{\max}$ and the insertion

cut which includes the optimal subcut described by (cap^*, n^*_-, n^*_+). Activities are sorted in two lists. List $\mathcal{L}_1$ is sorted by non decreasing order of LS_j's and is explored through index j for the generation of insertion cuts $(\mathcal{P}^0,\ \mathcal{S}^0), ..., (\mathcal{P}^\Delta, \mathcal{S}^\Delta)$ inside loop 4...13. At step 12, the next cut is obtained by transferring the activity j from the set of successors to the set of predecessors, unless the stop condition indentifying the last dominant cut holds. List $\mathcal{L}_2$ is sorted by non increasing order of EF_i's and explored through index i for the generation of subcuts $(\mathcal{P}^{\alpha,0}, \mathcal{S}^\alpha), ..., (\mathcal{P}^{\alpha,\Phi_\alpha},\ \mathcal{S}^\alpha)$ (loop 6...10). An initial remaining capacity vector $remcap$ is initialized to the resource capacity at step 5. The participation of i to the current sufficient insertion position is substracted from $remcap$ at step 9, simulating the removal of i from the set of predecessors. The makespan increase is reevaluated at step 8, unless the stopping condition identifying the last subcut holds. Note that since the removal of i is only simulated by using vector $remcap$, this removal will have to be done before the actual insertion by INSERT. The interest of the simulation is the low complexity it induces: it can be done in $\mathcal{O}(m)$ for each activity i to be removed. Hence GENERATEDOMINANTINSERTIONPOSITIONS has an $\mathcal{O}(n^2m)$ time complexity.

Algorithm GENERATEDOMINANTINSERTIONPOSITIONS($\delta C^*_{\max}, cap^*, n^*_-, n^*_+$)
1. Sort activities by non decreasing order of LS_j's in a list $\mathcal{L}_1$
2. Sort activities by non increasing order of EF_i's in a list $\mathcal{L}_2$
4. do
3. $\quad j :=$ next element of $\mathcal{L}_1$.
5. $\quad remcap(k) := R_k, \forall k \in \mathcal{R}$.
6. $\quad$ do
7. $\quad\quad i :=$ next element of $\mathcal{L}_2$.
8. $\quad\quad \delta C_{\max} = \max(0, \max(ES^-_x, EF_i) + p_x - \min(LF^+_x, LS_j))$
9. $\quad\quad remcap(k) := remcap(k) - cap_{ik}, \forall k \in \mathcal{R}$
10. $\quad$ while $remcap(k) \geq r_{xk}, \forall k \in \mathcal{R}$
11. $\quad$ If $\delta C_{\max} \leq \delta C^*_{\max}$ then $(\delta C^*_{\max}, cap^*, n^*_+, n^*_-) := (\delta C_{\max}, cap, n^+, n^-)$.
12. $\quad$ If $LS_j < LF^+_x$ then TRANSFERFROM$\mathcal{S}$TO$\mathcal{P}(j, cap, n^+, n^-)$.
13. while $LS_j < LF^+_x$.
End.

5 Experimental evaluation of the insertion algorithm for rescheduling in a dynamic RCPSP

The computational experiments performed in [1] tested the previous version of the insertion algorithm against priority rule based heuristics as rescheduling algorithm in a dynamic environment. The results show that, even if the insertion algorithm does not always outperform the priority rule based heuristic, its behaviour is much more stable: the average standard deviation of the makespan value is about twice lower for the insertion algorithm than for the rescheduling algorithm. In this section, we provide a new rescheduling algorithm based on the insertion algorithm and we test it against the parallel SGS/MINLFT priority rule for the insertion of an unexpected activity.

The sketch of the experiments, performed on the 600 KSD instances with 120 activities [12] is as follows. For each instance, an initial schedule and an initial flow are computed with GenFlow, the extension of the parallel SGS with the MINLFT priority rule presented in Section 3. Let $C_{\max}$ denote the makespan of this initial solution. A new activity, say x, is randomly generated to simulate the disruption. We then compare the makespan obtained for the insertion of this activity by the 3 following rescheduling algorithms, each having an $\mathcal{O}(n^2m)$ time complexity.

1. ParLft consists in generating a new problem taking account of x and then in computing a solution with the parallel SGS and the MINLFT priority rule.

2. ReschIns consists in inserting the new activity inside the initial flow by the proposed insertion algorithm.

3. ReschInsActive similar to ReschIns, except that a serial SGS is applied after the insertion, with the current start time of the activities as priority rule (lexicographic order breaks ties). The purpose of this second pass is to generate an active schedule strictly better than the one generated by ReschIns.

For each of the 600 instances, we perform 27 random generation, each corresponding to a variant in the characterisics of activity x. The characteristics are coded by a 3 digit number pwr giving the level of 3 characteristics compared to already scheduled activities. $p \in \{0, 1, 2\}$ describes the duration of the activity, i.e. small, medium and large, respectively. $w \in \{0, 1, 2\}$

describes the time window of the activity, i.e. the tightness of its precedence constraints. $r \in \{0, 1, 2\}$ describes the level of the request of the activity on the resources, i.e low, medium and high, respectively.

In figure 2, the results of the experiment are given in terms of the average and the maximum increase of the makespan, for each of the 27 experiments. It appears that the level of resource request is discriminant. For high resource requests, SCHEDINS and SCHEDINSACTIVE always outperform PARLFT. For low and medium resource requests, PARLFT performs better in average than RESCHINS, as already mentionned in [1]. However RESCHINSACTIVE always improves RESCHINS and make the difference with PARLFT unsignificant. For the maximal deviation the performance of PARLFT is drammatic and RESHINSACTIVE always slightly improves RESCHINS. Hence, RESCHINSACTIVE appears to be a robust new rescheduling method.

6 Insertion heuristics for the static RCPSP

6.1 A simple polynomial heuristic

For static scheduling, we propose a new simple heuristic, called INSHEUR which mixes the insertion algorithm and the parallel SGS with the objective to compensate the unstability of the priority rule based heuristic.

From a given static RCPSP instance with n activities, we generate n different RCPSP instances, each one corresponding to the initial problem except that one activity has been given a null duration and a null request on each resource. For each of these instances, we apply successively the extension of the parallel SGS and the insertion of the modified activity after restoring its duration and requests, as detailed below.

Algorithm INSHEUR
1. For $i \in V$ do
2. set $p_i := 0$; $r_{ik} = 0, \forall k \in \mathcal{R}$ and save actual values.
3. Generate a feasible flow $\mathcal{F}$ with the extension of the parallel SGS.
4. Compute feasible ES, EF, LS and LF, for all tasks
5. Restore the actual duration and requests of i
6. Insert i in $\mathcal{F}$ with the proposed insertion algorithm and compute $\mathcal{F}'$.
7. Store $\mathcal{F}'$ if it has the smallest makespan value obtained so far.
8. End For.
end.

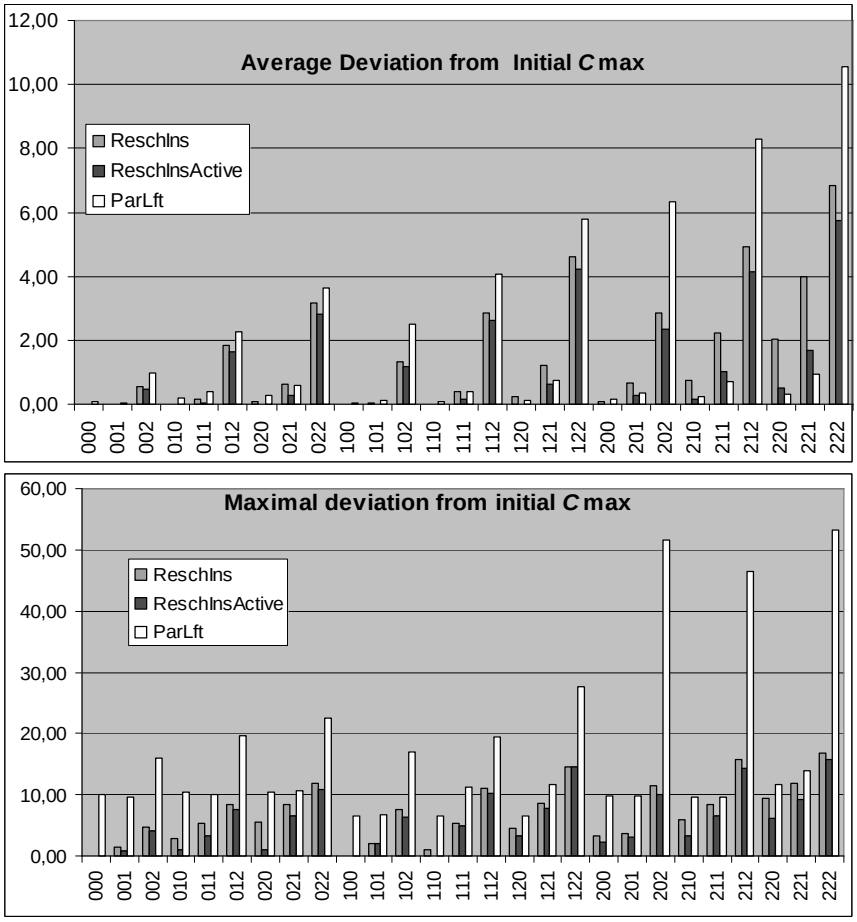

Figure 2: Comparison of three rescheduling algorithms

Hence InsHeur has an overall time complexity of $\mathcal{O}(n^3 m)$.

We illustrate an iteration of InsHeur in Figure 3. Suppose that for the RCPSP instance of Figure 1, both request and duration of activity 7 have been set to 0 (step 2 with $i = 7$). Part (a) of Figure 3 shows the AON-flow solution network obtained from this new instance by the extension of the parallel SGS with the MINLFT rule (step 3 and 4). A makespan value of 17 is obtained for this instance. Under each node, we have displayed values $ES_i..LS_i$ and $EF_i..LF_i$.

Suppose that we want now to insert activity 7 after restoring its duration and request (steps 5 and 6). To illustrate the behaviour of the insertion algorithm let us enumerate the sufficient insertion positions that are generated for this insertion.

Initial dominant cut: $(\mathcal{P}^0, \mathcal{S}^0) = (\{0,1,2,3,4,5\}, \{6,7,8,9,10,11\})$.
no predecessor can be removed from $\mathcal{P}^0$.
$\delta C_{\max}(\mathcal{P}^0, \mathcal{S}^0) = 7$; $\delta C^*_{\max} = 7$; $j^0 = 6$.
Cut $(\mathcal{P}^1, \mathcal{S}^1) = (\{0,1,2,3,4,5,6\}, \{7,8,9,10,11\})$.
no predecessor can be removed from $\mathcal{P}^1$.
$\delta C_{\max}(\mathcal{P}^1, \mathcal{S}^1) = 7$; $j^1 = 8$.
Cut $(\mathcal{P}^2, \mathcal{S}^2) = (\{0,1,2,3,4,5,6,8\}, \{7,9,10,11\})$.
predecessor 8 can be removed from $\mathcal{P}^1$.
$(\mathcal{P}^{2,1}, \mathcal{S}^2) = (\{0,1,2,3,4,5,6\}, \{7,9,10,11\})$.
no predecessor can be removed from $\mathcal{P}^{2,1}$.
$\delta C_{\max}(\mathcal{P}^{2,1}, \mathcal{S}^2) = 5$; $\delta C^*_{\max} = 5$; $j^2 = 9$
$(\mathcal{P}^2, \mathcal{S}^2)$ verifies the stop condition.

Hence the optimal insertion position is defined by ($\mathcal{P}^{2,1}, \mathcal{S}^2$) with a makespan increase of 5. Part (b) displays the result (the new AON-flow network) of the insertion of activity 7 after restoring its request and duration by algorithm Insert in $(\mathcal{P}^{2,1}, \mathcal{S}^2)$. The corresponding left-shifted schedule is displayed in part (c) and a makespan value of 22 is obtained, which strictly improve the result of the parallel SGS with the MINLFT rule on the initial instance.

The computational experiments have been performed on the Kolisch, Sprecher and Drexl (KSD) instances with 30, 60 and 120 activities [12] and on the Baptiste and Lepape (BL) instances with 20 and 25 activities [3]. We have tested InsHeur with priority rules MINLFT (Minimum Latest Finishing Time) and WCS (Worst Case Slack), a priority rule designed by Kolisch [13]. These two rules have been selected since they have been shown to perform best as single pass heuristics in [10] for the KSD instances. In Figure 4,

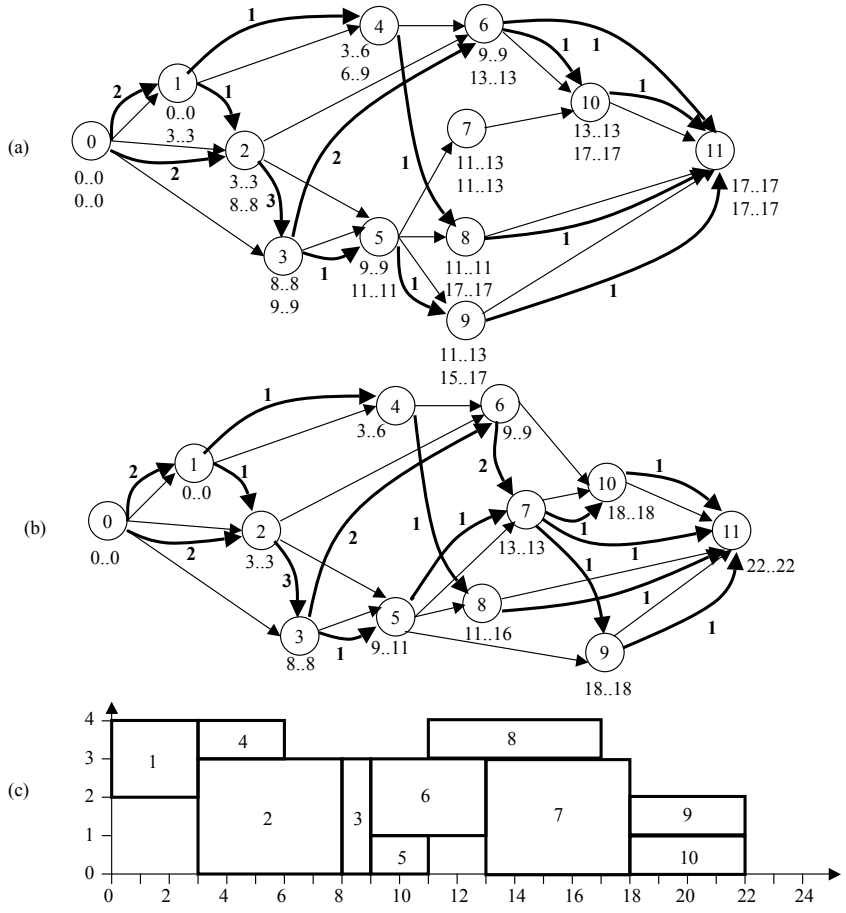

Figure 3: An iteration of InsHeur: insertion of activity 7

we have reported the average deviation from the optimal makespan for KSD instances with 30 activities and for BL instances obtained by InsHeur with the two rules. For the KSD instances with 60 and 120 activities we have reported the average deviation from the critical path lower bound, since not all of these instances have been solved to optimality. The experiments have been performed on a SUN sparc Ultra-4 workstation. On average for the largest problems (KSD 120 instances) an iteration of InsHeur lasts 0.07 seconds for the MINLFT rule and 0.15 secondes for the WCS rule, which is a more complex rule (see [13]).

To evaluate our results, we have also reported the performance of the best algorithms tested in [10]. For some of them which are multi-pass or iterative methods, the number of iterations is indicated, each iteration corresponding approximatively to a single call to the SGS.

Let us first compare InsHeur and the best single pass heuristics. On all tested instances, InsHeur clearly outperforms the single pass heuristics, especially for the BL instances (which are known to be hard highly cumulative instances) where the average deviation from optimum is almost twice lower for our algorithm. We obtain the best results with the WCS priority rule which is consistent with the results of [10]. This shows that our insertion algorithm can greatly improve the performance of myopic priority rule based heuristics, especially for highly cumulative cases.

Let us now compare the results of InsHeur with those of the iterative algorithms reported in [10], especially the sampling methods whose purpose is to improve the performance of single pass priority rule bases heuristics by randomly biasing the selection of the priority rule. A single call of InsHeur on an instance with n activities corresponds approximatively to n calls to the parallel SGS. Hence to evaluate fairly the performance of the proposed heuristic, we should compare its results to the ones of the other iterative algorithms presented in [10] limiting those algorithms to n iterations. Since the smaller number of iterations for which results are reported in [10] is much greater than n (1000 iterations), the displayed results in Figure 4 are obviously disadvantaging for our heuristic. However, it can be noticed that for the KSD instances with 30 and 60 activities, InsHeur performs only slightly worse than the sampling methods (1.74 vs 1.40 for KSD 30, 14.20 vs 13.66 for KSD 60) and less that 1.4% for KSD 30 (1.55% for KSD 60) above the performance of the best iterative algorithm limited to 1000 iterations (GA denotes the Genetic Algorithm of Hartmann [9]). The more encourageing results is for the KSD instances with 120 activities where our heuristic per-

forms better than GA limited to 1000 iterations, with smaller computational requirements. This seems to indicate that the performance of the proposed heuristic remains correct as the problem size increases.

		KSD			BL
		$n = 30$	$n = 60$	$n = 120$	$n = 20, 25$
Algorithm	Iterations	(%opt)	(%CpmLb)	(%CpmLb)	(%opt)
InsHeur/LFT	n	1.93	14.38	39.56	5.34
InsHeur/WCS	n	1.74	14.20	39.34	4.81
results from [10]:					
Par. LFT	1	4.39	17.46	43.86	10.35*
Par. LFT/sampling	1000	1.40	13.59	39.60	-
Par. WCS	1	3.88	16.87	43.57	8.95*
Par. WCS/sampling	1000	1.40	13.66	39.65	-
Ser. GA act. list	1000	0.38	12.68	39.37	-

* these results have been obtained by our implementation of the parallel SGS.

Figure 4: Average deviation from optimum or critical path lower bound

6.2 A tabu search method

We end the description of the possible uses of the insertion algorithm by its inclusion in a tabu search method, with multiple neighborhoods defined by the reinsertion of one or several activities. We use GenFlow (see Section 3) with the MINLFT priority rule to generate an initial solution.

To reinsert an activity, it is first removed from the flow network by a simple algorithm which redirects the incoming flow of the activity to its resource successors in a lexicographic order. Then, the earliest and latest start times are updated by longest path computations. Last, the activity is inserted by the insertion algorithm. Note that such a move can be performed in $\mathcal{O}(n^2m)$. We define as V_1 the set of all schedules (neighbours) that can be obtained by applying such a reinsertion scheme to an activity. For neighborhood V_1, we restrict to the reinsertion of all critical activites and keep the best non tabu solution.

Another neighborhood V_2 is used for the diversification of the search. It uses the decomposition of the AON solution network in levels. Suppose that

the network has l levels. Then, V_2 defines l neighbours, each one being obtained by removing successively all activities of the same level, and then by reinserting them successively. We remove the activities in the lexicographic order and we reinsert them in the reverse order. We do not test multiple removal and reinsertion orders, which highly reduce the search space. Similarly as for V_1, activities can only be inserted in non tabu positions.

An element of the tabu list is made of the activity, the set of resource successors and the set of resource predecessors. A move is tabu if it reinserts the same task after the same predecessors or before the same successors (see [5] and [7] for a similar definition for the generalized job-shop). The tabu list size varies dynamically from $TL_{\min}$ to $TL_{\max}$.

We integrate neighborhoods V_1 and V_2 as follows: if no improvement on the best solution found occurs, we perform at most NI_1 iterations of neighborhood V_1, then NI_2 iterations of V_2. Each time the best solution is improved, we restart this sequence from the beginning.

We have tested the tabu search method on the instances of [12] with 60 and 120 activities. For the 480 instances with 60 activities, we have set $TL_{\min} = 5$, $TL_{\max} = 30$, $NI_1 = 3000$, $NI_2 = 2000$, setting the maximum number of iterations without improvement to 5000. We obtain an average deviation above the CPM lower bound of 12.05% with an average CPU time of 3.2s. In [10], an experimental evaluation of state-of-the-art metaheuristics limited to 5000 iterations is reported. The best one is the genetic algorithm of Hartmann [9] with an average deviation above the CPM lower bound of 11.89%. The tabu search method of Baar *et al* [4] obtains 13.49%.

For the 600 instances with 120 activities, we have set $TL_{\min} = 10$, $TL_{\max} = 60$, $NI_1 = 8000$, $NI_2 = 3000$, setting the maximum number of iterations without improvement to 11000. We obtain an average deviation above the CPM lower bound of 36.16% with a average CPU time of 67s. For a comparison, the Genetic Algorihm of Hartmann [9] obtains an average deviation of 36.74%, being limited however to 5000 iterations.

7 Concluding remarks

In this paper, we have proposed an $\mathcal{O}(n^2m)$ version of a polynomial insertion algorithm for the resource-constrained project scheduling problem. We have shown that such an algorithm is of great interest for robust rescheduling in a dynamic environment. The use of this algorithm embedded inside a simple

$\mathcal{O}(n^3m)$ constructive heuristic for the static RCPSP has been shown to be efficient, especially for highly cumulative instances compared to single pass priority-rule based heuristics and for the largest instances of the literature compared to recent sampling methods. We have embedded the insertion algorithm inside a tabu search method competitive with the state-of-the-art approaches reported in [10].

We will focus now on improving the performance of the insertion algorithm by testing new methods for generating initial flows. At each iteration of the SGS, some priority rules (other than the lexicographic order used in this paper) could be defined to choose the completed activities candidate for sending flow to the activity selected for being scheduled. The serial scheduling scheme has also to be extended to flow generation.

We believe that the tabu search method could be highly improved by analyzing the connectivity of its neighborhood and by adapting to the RCPSP the concept of blocks of critical activities that have been shown useful for the generalized job-shop problem [5], [15].

References

[1] Artigues, C., Roubellat, F., 2000. A polynomial insertion algorithm in a multi-resource schedule with cumulative constraints and multiple modes. European Journal of Operational Research. 127, 297-316.

[2] Artigues, C., Roubellat, F., An efficient algorithm for operation insertion in practical jobshop schedules. to appear in Production Planning and Control.

[3] Baptiste P., Le Pape C., 2000. Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems, Constraints 5, 119-139.

[4] Baar T., Brucker P., Knust S., 1998. Tabu-search algorithms and lower bounds for the resource-constrained project scheduling problem, in: S.Voss, S.Martello, I.Osman, C.Roucairol (eds.): Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization, Kluwer, 1-18.

[5] Brucker, P., Neyer, J., 1998. Tabu-search for the multi-mode job-shop problem. OR Spekrum 20, 21-28.

[6] Brucker, P., Drexl, A., Möring, R., Neumann, K., Pesch, E., 1999. Resource-constrained project scheduling: notation, classification, models, and methods, European Journal of Operational Research 112, 3-41.

[7] Dauzère Pérès, S., Roux, W., Lasserre J.B., 1998. Multi-resource shop scheduling with resource flexibility, European Journal of Operational Research 107, 289-305.

[8] Fortemps P., Hapke M., 1997. On the Disjunctive Graph for Project Scheduling. Foundations of Computing and Decision Sciences, vol. 22 (1997), pp. 195-210.

[9] Hartmann, S., 1998, A competitive genetic algorithm for resource-constrained project scheduling. Naval Research Logistics, Vol. 45, p. 733-750.

[10] Hartmann, S., Kolisch, R., 2000 . Experimental Evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem, European Journal of Operational Research. 127, 297-316.

[11] Klein, R., Scholl, A., 1999. Computing lower bound by destructive improvement: An application to resource-constrained project scheduling, European Journal of Operational Research 112, 322-346.

[12] Kolisch, R., Sprecher A., Drexl A., 1995, Characterization and generation of a general class of resource-constrained project scheduling problem, Management Science, 41(10), 1693–1703.

[13] Kolisch, R., 1996, Efficient priority rules for the resource-constrained project scheduling problem. Journal of Operations Management, 14(3), p. 179-192, 1996.

[14] Kolisch, R., Hartmann, S., 1999. Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis, in: Project Scheduling: Recent Models, Algorithms and Applications, chapter 7, J. Weglarz, Kluwer, Amsterdam, 147-178.

[15] Vaessens, R.J.M., Generalize job shop scheduling: complexity and local search. Ph.D. thesis, Eindhoven University of Technology, Rotterdam.