KATHOLIEKE
UNIVERSITEIT
LEUVEN

DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

RESEARCH REPORT 0128

MODELS FOR ROBUST RESOURCE ALLOCATION IN PROJECT SCHEDULING

by
R. LEUS
W. HERROELEN

D/2001/2376/0128

Models for Robust Resource Allocation in Project Scheduling

Roel Leus and Willy Herroelen

August 2001

Operations Management Group
Department of Applied Economics
Katholieke Universiteit Leuven
Naamsestraat 69, B-3000 Leuven (Belgium)
Phones +32-16-32 69 67 and +32-16-32 69 70
Fax +32-16-32 67 32
e-mail: <first name>.<name>@econ.kuleuven.ac.be

Models for Robust Resource Allocation in Project Scheduling

Roel Leus[§] and Willy Herroelen

ABSTRACT

The vast majority of resource-constrained project scheduling efforts assumes complete information about the scheduling problem to be solved and a static deterministic environment within which the pre-computed baseline schedule will be executed. In reality, however, project activities are subject to considerable uncertainty which generally leads to numerous schedule disruptions. In this paper, we present a resource allocation model that protects the makespan of a given baseline schedule against activity duration variability. A branch-and-bound algorithm is developed that solves the proposed robust resource allocation problem in exact and approximate formulations. The procedure relies on constraint propagation during its search. We report on computational results obtained on a set of benchmark problems.

Keywords: *Project management; Project planning; Scheduling; Constraint Satisfaction.*

[§]Research Assistant of the Fund for Scientific Research – Flanders (Belgium) (F.W.O.)

1. Introduction

The research on the resource constrained project scheduling problem (RCPSP) has widely expanded over the last few decades (for reviews see Brucker et al., 1999; Herroelen et al., 1998; Kolisch and Padman, 2001). The vast majority of these research efforts focus on exact and suboptimal procedures for constructing a workable schedule assuming complete information and a static deterministic problem environment. Project activities are scheduled subject to both precedence and resource constraints, mostly under the objective of minimizing the project duration. The resulting schedule, subsequently referred to as the *pre-schedule*, serves as the baseline for executing the project.

During project execution, however, project activities are subject to considerable uncertainty which may lead to numerous schedule disruptions. This uncertainty stems from a number of possible sources: activities may take more or less time than originally estimated, resources may become unavailable, material may arrive behind schedule, new activities may have to be incorporated or activities may have to be dropped due to changes in the project scope, ready times and due dates may be modified, etc. As a result, the validity of static deterministic scheduling has been questioned and/or heavily criticized (Goldratt, 1997) and issues of project management under uncertainty and risk management have received growing attention (Meredith and Mantel, 2000; Chapman and Ward, 1997).

The recognition that uncertainty lies at the very heart of project management induced a number of research efforts. The uncertainty in activity duration has since long also been the subject of more technical project planning approaches based on stochastic analysis in the absence of resource constraints (the PERT problem) (Elmaghraby, 1977; Adlakha and Kulkarni, 1989), and recently also in the presence of resource constraints (the stochastic resource constrained project scheduling problem, SRCPSP) (Igelmund and Radermacher, 1983; Stork, 2000). From a decision theoretical point of view, a risk situation occurs; the link between decisions and outcomes is probabilistic (Rosenhead et al., 1972).

Given the presence of both random activity durations and resource constraints, some authors do not start from a pre-schedule but construct the project schedule through the application of so-called *scheduling policies* or *scheduling strategies* as time progresses (Igelmund and Radermacher, 1983). Such a policy can be looked at as an on-line decision process which dynamically makes scheduling decisions based on the observed past and the a priori knowledge about the processing time distributions. In this paper, we take as input a pre-schedule which is used to define a resource allocation problem the solution of which should protect the schedule makespan against activity duration variability. Repairing the schedule when deviations from the initial activity duration estimates become known, implies that the activity start times are updated by performing new CPM-calculations based on an earliest start policy with the added restriction that the feasible partially ordered set which determines the policy is compatible with the given pre-schedule.

The organization of the paper is as follows. We start with a discussion of a so-called resource flow network (Section 2) and its relation with a feasible resource-constrained project schedule. We demonstrate that a feasible pre-schedule allows for the construction of one or more resource flow networks, each representing a feasible allocation of the resources to the individual activities. Preference should be given to

a robust resource flow network, i.e. a resource allocation which offers maximal protection of the schedule makespan against schedule disruptions. The two robustness objectives used in this paper are discussed in Section 3: minimizing the expected project duration and maximizing a function of the activity floats. We show that the latter, though a heuristic, still results in a NP-hard problem. Section 4 describes a branch-and-bound procedure which generates a robust resource allocation under both objectives. The procedure starts from direct and transitive technological precedence relations and iteratively adds resource arcs until a feasible flow is obtained which optimizes the robustness objective. The procedure exploits constraint propagation techniques which are discussed in Section 5. An efficient procedure for testing for the existence of a feasible flow is discussed in Section 6. Computational results obtained by the algorithm on a set of test problems are discussed in Section 7. Section 8 provides overall conclusions and offers some suggestions for future research.

2. Resource flow networks

2.1 *Basic definitions and notation*

It is assumed that a set of activities N is to be scheduled on a single renewable resource type with availability a. Activities are numbered from 0 to n ($|N|=n+1$) and activity i has duration $d_i \in \mathbb{N}$ and requires $r_i \in \mathbb{N}$ units of the single renewable resource. Apart from the dummy start activity 0 and dummy end n, activities have non-zero duration; the dummies also have zero resource usage. For any $E \subseteq N \times N$, define graph G_E formed by nodes N and arcs E: $G_E := G(N,E)$. A is the set of pairs of activities between which a finish-start precedence relationship with time lag 0 exists. We assume G_A to be acyclic. TA is the transitive closure of A: $(i,j) \in TA$ if a path from i to j exists in G_A. If $E \subseteq N \times N$, we can obtain the immediate predecessors of activity i by function π_E: $N \rightarrow 2^N$: $i \rightarrow \pi_E(i) = \{j \in N \mid (j,i) \in E\}$, and its immediate successor activities via σ_E: $N \rightarrow 2^N$: $i \rightarrow \sigma_E(i) = \{j \in N \mid (i,j) \in E\}$. We assume A to be minimal, that is, $\neg(\exists (i,j) \in A, k \in N: k \in \sigma_A(i) \wedge j \in \sigma_{TA}(k))$. Without loss of generality, we also require $\forall (i,j) \in A: i<j$. To simplify notation, if $X,Y \subseteq N$, let $(X,Y) := \{(i,j) \mid i \in X \wedge j \in Y\}$, and for any function g defined on N or $N \times N$, if Z is a subset of the support of g, let $g(Z) := \sum_{z \in Z} g(z)$. We may denote a set consisting of one element by its single element and omit duplicated brackets. We define $N_0 := N \setminus \{0\}$, $N_n := N \setminus \{n\}$, and $N_{0n} := N \setminus \{0,n\}$. For the dummy activities it holds that $\sigma_{TA}(0) = N_0$ and $\pi_{TA}(n) = N_n$.

A schedule C is associated with an $(n+1)$-vector of start times $\mathbf{s}$; every $\mathbf{s}$ implies an $(n+1)$-vector of finish times $\mathbf{e}$, $e_i = s_i + d_i$, $\forall i \in N$. A schedule C is characterized by an ordered list Λ_C of time instances or 'decision points' which correspond to its activity start or finish times: $\Lambda_C(t_0, t_1, \ldots, t_{|\Lambda_C|})$; $0 = t_0 < t_1 < \ldots < t_{|\Lambda_C|}$ and $\forall t \in \Lambda_C$, $\exists i \in N$: $t = s_i \vee t = e_i$. Define $\Lambda_{C0} := \Lambda_C \setminus \{t_0\}$, $\Lambda_{C0n} := \Lambda_C \setminus \{t_0, t_n\}$, and $\Lambda_{Cn} := \Lambda_C \setminus \{t_n\}$. Define $N_t := \{i \in N \mid s_i < t \leq e_i\}$, $\forall t \in \Lambda_{C0}$, the activities which are active during period t. Schedule C is feasible if (1) $\forall (i,j) \in A$: $e_i \leq s_j$, and (2) $\forall t \in \Lambda_{C0}$: $r(N_t) \leq a$. A resource-constrained project scheduling (RCPSP)-instance $\Gamma(N,A,a,\mathbf{d},\mathbf{r})$ aims to find a feasible schedule C^* that minimizes e_n.

2.2 *Resource flow network and total resource allocation*

Naegler and Schoenherr (1989), Bowers (1995) and Artigues and Roubellat (2000) present, seemingly independently of each other, the concept of a *resource flow network*, in which the amount of resources being transferred immediately from one activity to another is explicitly recorded. Artigues and Roubellat (2000) and Artigues et al. (2000) use the obtained flow network to insert a new activity into the project, subject to unchanging resource allocations. Bowers (1995) defines "resource constrained float" as the classical CPM total float based on an extended network. Naegler and Schoenherr (1989) solve deterministic time/resource and time/cost trade-off problems (and other objectives) via the correspondence between schedules and resource flows, and duality considerations. In their article, uncertainty is only studied by allowing stochastic resource usage of the activities.

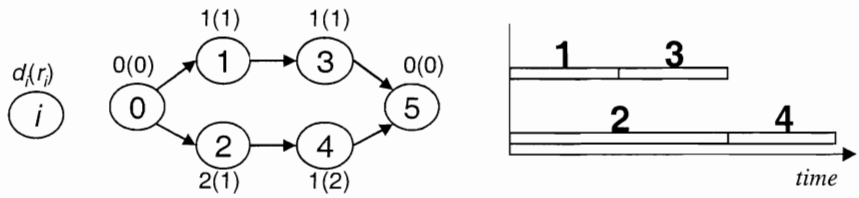
Define $u_i=r_i$, $\forall i\in N_{0n}$, and $u_0=u_n=a$. Define the flows $f_{ij}\in\mathbb{N}$ to represent the number of resources that are transferred from activity i (when it finishes) to activity j (when it starts). A flow f must satisfy the following flow conservation constraints:

$$f(i,N) = u_i \qquad \forall i\in N_n \qquad (2.1)$$

$$f(N,i) = u_i \qquad \forall i\in N_0 \qquad (2.2)$$

We define the resource flow network associated with flow f as the graph $G_{E_f}=(N,E_f)$ where $E_f=\{(i,j)\in N\times N \mid f_{ij}>0\}$.

A small example is in order at this point. In Figure 1(a), an example network is represented in activity-on-the-node format, with a=3. Figure 1(b) shows an early start schedule for the project. This schedule happens to be precedence and resource feasible. Moreover, it is optimal in the sense that it minimizes e_5. The corresponding vector of start times is **s**=(0,0,0,1,2,3). The vector of finish times is **e**=(0,1,2,2,3,3).

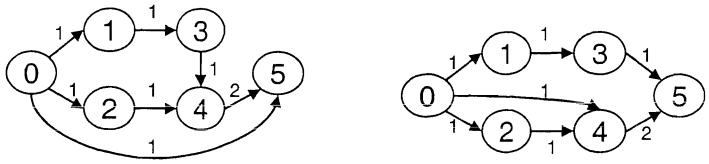


(a) (b)

Figure 1. Example project network and early start schedule.

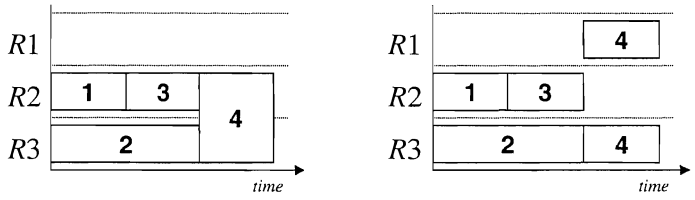
According to our definitions, $u_0=u_5=3$, $u_1=1$, $u_2=1$, $u_3=1$ and $u_4=2$. One possible resource flow, allocating only 2 out of the 3 available units of the resource type to non-dummy activities, sets $f_{01}=f_{02}=f_{05}=f_{13}=f_{24}=f_{34}=1$ and $f_{45}=2$, all other flows to 0. This flow is illustrated by the resource flow network of Figure 2(a). What we will subsequently refer to as a *total resource allocation* is represented in Figure 3(a), where we identify the different resource units as $R1$, $R2$ and $R3$. One of the available resource units is transferred from the end of dummy activity 0 to the start of activity 1, denoting that $R2$ actually executes the activity. A second resource unit is transferred from the end of dummy activity 0 to the start of activity 2, denoting that $R3$ actually performs activity 2. The remaining resource unit ($R1$) is transferred from the end of dummy activity 0 to the start of dummy activity 5, indicating that this resource unit is not used. The resource unit $R2$, released at the completion of activity 1, is transferred to the start of activity 3, indicating that $R2$ will execute activity 3.

The one-unit flows transferred from the completion of activity 2 and activity 3 to the start of activity 4 indicate that $R2$ and $R3$ will perform activity 4. At the completion of activity 4, the two resource units are released as indicated by $f_{45}=2$.



(a) (b)

Figure 2. Two resource flow networks.

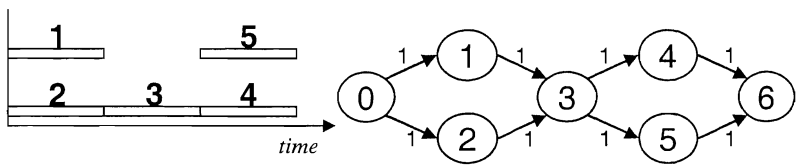


(a) (b)

Figure 3. Two total resource allocations.

The resource flow network shown in Figure 2(b) represents another flow, defined by $f_{01}=f_{02}=f_{04}=f_{13}=f_{24}=f_{35}=1$ and $f_{45}=2$. As shown by the resource allocation indicated in Figure 3(b), this flow uses all three available resource units. The one-unit flow transferred from the end of dummy activity 0 to the start of activity 4 indicates that $R1$ will work on activity 4. Activity 4 is allocated a second resource unit $R3$, made available by the completion of activity 2.

The two resource flow networks shown in Figure 2 represent a complete resource allocation, as all resource units are individually identified. This is not always the case, however. To illustrate, consider a set of non-dummy activities $\{1,2,3,4,5\}$ with $\mathbf{d}=(1,1,1,1,1)$, $r_i=1$ for $i\neq 3$, $r_3=2$, $a=2$ and $A=\varnothing$. Figure 4(a) shows an early start schedule with corresponding resource flow network G_{E_f} shown in Figure 4(b). The network shows the feasible flow values $f_{01}=f_{02}=f_{13}=f_{23}=f_{34}=f_{35}=f_{46}=f_{56}=1$. The identification of this feasible flow leaves undecided whether or not activities 1 and 4 will be executed by the same resource. We will refer to such decisions by using the term "total allocation", as was already done above and in Figure 3.



(a) Early start schedule (b) Resource flow network

Figure 4. Early start schedule and corresponding resource flow network.

We now define $R_f=E_f\setminus TA$. In Figure 2(a), $R_f=\{(3,4)\}$, while $R_f=\varnothing$ in the resource flow network of Figure 2(b). The arcs in R_f denote the arcs in the resource flow network G_{E_f} which do not represent direct nor transitive precedence relations. The condition for the flow f to be feasible can now be written as:

$$G_{TA\cup R_f} \text{ acyclic} \tag{2.3}$$

2.3 *Flow compatible schedules*

A *schedule* C is said to be *compatible* with a flow f, written $C\sim f$, if C respects inequalities $e_i\leq s_j$ for every $(i,j)\in TA\cup R_f$. For such a schedule, define $I_t:=\{(i,j)\in TA\cup R_f \mid j\in N_t\}$, all arcs entering N_t-jobs, and $P_t:=\{(i,j)\in TA\cup R_f \mid e_i<t\leq s_j\}$, the arcs that are 'in parallel' with N_t. Clearly, the early start schedule of Figure 1(b) is compatible with the flows shown in Figures 2(a) and 2(b). For the resource flow network of Figure 2(a), $I_3=(\{0,2,3\},4)$ and $P_3=(\{0,1,2,3\},5)$, while for the resource flow network of Figure 2(b) we have $I_3=(\{0,2\},4)$ and $P_3=(\{0,1,2,3\},5)$.

The following three theorems establish useful relationships between a feasible schedule and a feasible flow.

Theorem 1. *If schedule C is compatible with flow f, then $I_t\cup P_t$ is a network cut in the graph $G_{TA\cup R_f}$, $\forall t\in\Lambda_{C0}$.*

Proof. Define $X=N_t\cup\{i\in N \mid \exists j\in N:(j,i)\in P_t\}$. Clearly, $X\subseteq N$, and it holds that $X\cup\sigma_{TA\cup R_f}(X)=X$, because for any $i\in X$ and $j\in\sigma_{TA\cup R_f}(i)$ we have $(0,j)\in TA$. This implies that $I_t\cup P_t=(N\setminus X,X)$, the single source node is in $N\setminus X$ and the single sink node is in X. A set of arcs that satisfies these conditions is a network cut. □

Corollary. *$\forall t\in\Lambda_{C0}$, it holds that*

$$r(N_t)+f(P_t)=a. \tag{2.4}$$

For the flow that is depicted in Figure 2(a), Figure 5 shows the graph $G_{TA\cup R_f}$ and the three cuts defined by $I_1\cup P_1$, $I_2\cup P_2$, $I_3\cup P_3$. As can be verified, the flow across each of these cuts equals 3.

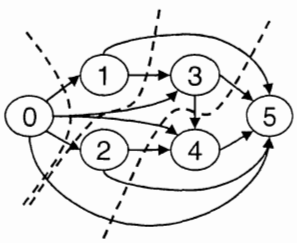


Figure 5. Cuts corresponding to a flow compatible schedule.

Theorem 2. *A schedule C that is compatible with a feasible flow f, is a feasible schedule.*

Proof. Recall that a schedule C is feasible if (1) $\forall(i,j)\in A$: $e_i \leq s_j$, and (2) $\forall t\in\Lambda_{C0}$: $r(N_t)\leq a$. The fact that C is compatible with f implies that Condition (1) is satisfied. Condition (2) holds because $r(N_t) = f(I_t \cup P_t) = a$, for every decision point $t\in\Lambda_{C0}$. □

Define $\theta(E)$, $E\subseteq N\times N$, to be the schedule defined by the ordinary early start CPM-calculations in G_E, provided this graph is acyclic. For a feasible f, $G_{TA\cup R_f}$ is acyclic and thus $\theta(TA\cup R_f)$ is an unambiguous schedule, that is compatible with f. In other words, at least one feasible schedule can be associated with every feasible flow.

Let $S_t:=\{i\in N_{0n} \mid s_i=t\}$, $W_t:=\{i\in N_{0n} \mid e_i=t\}$ and $\Sigma_t:= (\cup_{t^e\leq t} W_{t^e})\cup\{0\}$, $\forall t\in\Lambda_C$. It is clear that $N_{(i)}=N_{(i-1)}-W_{(i-1)}+S_{(i-1)}$, $\forall t_i\in\Lambda_{C0n}$, if we set $(i):=t_i$ and $N_0:=\varnothing$.

Theorem 3. *Every feasible schedule C is compatible with at least one feasible flow f.*

Proof. Consider the following algorithm. Initialize $f_{0n}:=a$, all other flows set to 0. From 1 to $|\Lambda_C|-1$, iterate over all decision points. At any decision point t, try to reroute flow in (Σ_t,n) to enter activities S_t and continue to n, such that $f(\Sigma_t,i)=f(i,n)=r_i$, $\forall i\in S_t$. Denote by a_t the value $f(\Sigma_t,n)$ before such rerouting; rerouting is possible if $a_t\geq r(S_t)$. Note that $a_0=a$ and $a_{(i)}=a_{(i-1)}-r(S_{(i-1)})+r(W_{(i)})$, $\forall t_i\in\Lambda_{C0n}$, and also $S_0=N_{(1)}$. We see that $a_0\geq r(S_0)=r(N_{(1)})$ and $a_{(1)}=a-r(N_{(1)})+r(W_{(1)})$. By induction, if $a_{(i)}=a-r(N_{(i)})+r(W_{(i)})$, then $a_{(i+1)}=a-r(N_{(i+1)})+r(W_{(i+1)})$, for all t_i up to $|\Lambda_C|-2$. Whence, $a_{(i)}\geq r(S_{(i)})$ iff $r(N_{(i+1)})\leq a$, $\forall t_i\in\Lambda_{C0n}$, which holds because C is feasible. Conditions (2.1) and (2.2) hold for constructed f. Also, as every nonzero flow f_{ij} has $e_i\leq s_j$, (2.3) holds. □

Define problem Φ as the problem of finding a feasible flow f^* that minimizes $s_n(\theta(TA\cup R_f))$. Φ is an extension of the disjunctive graph representation of the classical job shop scheduling problem (Roy and Sussman, 1964). An optimal solution to Φ is a flow that has an associated schedule with minimal duration. From the above theorems, every optimal solution to the resource-constrained project scheduling problem Γ is compatible with a feasible flow, hence for any f^* optimal to Φ and C^* optimal to Γ, we have that $s_n(C^*)=s_n(\theta(TA\cup R_{f^*}))$.

2.4 *Reacting to activity duration disruptions*

The remainder of the paper will focus on the consequences of variations in the activity duration vector **d**. Reacting to disruptions in activity durations can be performed by means of the resource flow network, based on the following theorem:

Theorem 4. *A feasible flow f will yield a feasible schedule C independent of the activity durations* **d**.

Proof. Schedule $\theta(TA\cup R_f)$ is compatible with f, independent of **d**, and hence constitutes a feasible schedule by Theorem 2. □

'Repairing' the schedule when deviations in the initial duration data become known, implies that we update the start times by performing new forward pass CPM-calculations. In the terminology of Igelmund and Radermacher (1983), the set of precedence relations $TA\cup R_f$ defines an Earliest Start policy (*ES*-policy). The remainder of this paper will be concerned with the search for a 'good' *ES*-policy (evaluation criteria are discussed in Section 3) with the additional constraint that the

input schedule is compatible with the feasible partially ordered set determining the policy. This compatibility guarantees that the pre-schedule will be realized if everything goes as planned. We construct feasible solutions based on feasible flow considerations, rather than on the explicit "destruction" of all (minimal) forbidden sets (Igelmund and Radermacher 1983; Radermacher 1985; Stork and Uetz 2000), where a forbidden set is a set of precedence unrelated activities that are not allowed to be scheduled simultaneously because of resource constraints.

During the schedule repair process, we make the implicit assumption that the resource allocation remains constant, i.e. the resource flow network is maintained. To grasp the impact of this assumption, refer again to the resource flow networks in Figure 2 and the associated total resource allocations in Figure 3. Suppose that project management is uncertain about the duration of activity 3, and, when activity 3 turns out to take longer than initially projected, will proceed with the same resource allocation. It is obvious that in this case, the flow pattern in Figure 2(b) is more robust with regards to makespan than the pattern in Figure 2(a). Figure 2(b) shows that activity 3 does not need to pass on any resources to non-dummy activities upon its completion and so has a total float of one time period, whereas the resource flow network of Figure 2(a) causes a prolongation of its duration to have an immediate impact on the makespan of the repaired schedule. Our main objective in this paper is to construct a robust baseline schedule through the construction of a robust resource flow network.

The reactive scheduling policy we adopt will be preferred in a multi-project environment, where resources are scheduled across multiple projects, or if stability of the larger part of the schedule is an issue (avoiding system nervousness resulting from complete rescheduling), or also simply if efficiency and ease of calculations are imperative. These efficiency and stability considerations are explicitly referred to by Artigues and Roubellat (2000). Bowers (1995) mentions the cases where specialist resources (e.g. expert staff) cannot be easily transferred between activities at short notice, or when in a multi-project environment it is necessary to book key staff or equipment in order to guarantee their availability.

3. Robust resource allocation

We will assume only duration increases may occur (Gutierrez and Kouvelis (1991) provide a motivation based on expected worker behaviour under Parkinson's Law). Activity i ($i\neq 0,n$) is prolonged with probability p_i, a rational number. If an activity is disturbed, its disturbance length L_i is a random variable with specified *cdf* (cumulative distribution function) F_i; activity disruptions are independent. For an input schedule C, define $R(C)=\{(i,j)\in N\times N \mid (i,j)\notin TA \wedge e_i(C)\leq s_j(C)\}$. We are now ready to define our basic problem.

3.1 *Problem PE*

The basic problem we study in this paper can be posed as follows:

Problem *PE*:
construct a feasible flow f with $R_f\subseteq R(C)$ such that $E[S_n(\theta(TA\cup R_f))]$ is minimized,

where $S_n(\theta(TA \cup R_f))$ is a stochastic variable representing the earliest starting time of activity n in the resource-unconstrained network $GA_{TA \cup R_f}$ and $E[\cdot]$ is the expectation operator. It is clear from the definition of $R(C)$ that for any feasible f as solution to problem *PE*, $C \sim f$. Unfortunately, evaluation of the objective value amounts to the PERT problem, which cannot be efficiently solved (Hagstrom, 1988; Möhring, 2000). For this reason one usually approximates the expected makespan of a given policy by simulation (Stork, 2000). This simulation approach, however, will prove to be time consuming and an approximative problem can be posed as follows.

3.2 *Problem PTF*

Disregarding cumulation of disturbances, S_n can be approximated as follows:

$$S_n = s_n(C) + \max_{i \in N_{0n}} \{ \max\{0;(L_i - TF_i)\} \},$$

which would encourage us to minimize the second term of the right hand side (in expected value). TF_i is the CPM based total float in $G_{TA \cup R_f}$ and our (indirect) decision variable. However, given a tight due date for the project (cfr. infra), a number of activities will remain critical at all times, thus invalidating almost completely the significance of the objective. Another line of action would be to maximize the probability of the project coming in on time. For any activity i, $(1-p_i) + p_iF_i(TF_i)$ is the probability that $L_i \leq TF_i$. Thus,

$$Pr[\text{on time}] = \prod_i \left[(1-p_i)+p_iF_i(TF_i)\right] = \prod_i \left[1+p_i(-\overline{F}_i(TF_i))\right],$$

with $\overline{F}_i = 1-F_i$. Then, writing $\overline{F}_i(TF_i)$ as $\overline{F}_i$,

$$Pr[\text{on time}] = 1 - \sum_i p_i\overline{F}_i + \sum_{i<j} p_ip_j\overline{F}_i\overline{F}_j - \ldots$$

which can be approximated by

$$Pr[\text{on time}] = 1 - \sum_i p_i\overline{F}_i,$$

provided that the p-values are not very close to 1 and/or protection from disturbances is not close to 0. Computational experiments have shown that for the makespan objective, the approximation yields results of exactly the same quality; in the same time we shift from a multiplication to a summation, which is easier to manipulate. Again we take no notice of cumulation of disturbances, which basically implies the assumption that disturbances are either scarce enough or mostly not too long, such that most disturbances are dampened and even completely absorbed while propagated through the network. Although extremely large disturbances have a nonzero chance of occurrence, as well as the cumulation of a number of disadvantageous time overruns, it is our opinion that the robust resource allocation we try to create should not be required to deal with those cases. If and when they occur, management will need to take proper action to change the schedule and/or the resource allocations appropriately (for related but not entirely similar ideas, see Goldratt (1997)).

In conclusion, we define the following problem:

Problem *PTF*:
select R_f *such that* $\sum_{i \in N_{0n}} p_iF_i(TF_i)$ *is maximized,*

where the TF_i are determined recursively as follows:

$$TF_n = \omega \tag{3.1}$$

$$TF_i \le TF_j + s_j(C) - e_i(C) \qquad \forall\ (i,j) \in A \cup R_f \tag{3.2}$$

When it is not necessary to distinguish between problem *PE* and problem *PTF*, we will refer to problem $P=P(C)$, capturing the general aspects of both. By setting ω to zero, we impose a tight due date for the project. The recursion is equivalent with a "single pass" (backward) CPM calculation; this suffices because a forward pass is implicit in the provided starting times. We can reintroduce propagation of disturbances by passing on less then the entire TF_j to predecessors i in (3.2):

$$TF_i \le (TF_j - \gamma p_j E(L_j)) + s_j(C) - e_i(C) \qquad \forall\ (i,j) \in A \cup R_f \tag{3.3}$$

It is to be noted that some float values may become negative in this way. If no corrections are inserted, however, *TF* never decreases from successor to predecessor, and optimization will largely focus only on the back part of the network. Nevertheless, this may be the preferred choice if management itself will deal with disruptions when those threaten to reduce *TF* values too much. γ measures the degree in which expected values of disruptions are propagated throughout the network.

Using the proposed approximations is a simple way to circumvent the need to perform PERT computations in an environment where exact specification of distribution functions is impossible. With this in mind, we would like to make a case for the introduction of total float-related objectives to be studied in the scheduling literature; this section and the next present some means to interpret these objectives in settings characterized by uncertainty.

3.3 *Complexity of problem PTF*

Consider the following decision problem associated with *PTF*, with U a rational number:

Problem *DTF*:
can we find R_f *such that* $\sum_{i \in N_{Rn}} p_i F_i(TF_i) \ge U$?

We will show that every instance of the NP-complete decision problem version of the parallel machine scheduling problem under the weighted completion time objective can be written as an instance of problem *DTF*, with $\omega=\gamma=0$, linear *cdf*s and a given scheduling order of the activities.

In problem *PTF* with $\omega=\gamma=0$, $s_n(C)$, the input schedule makespan, provides an upper bound on the achievable total float by any activity. Assume the *cdf*s to be linear functions $F_i(x)=x/s_n(C)$, for $x \in [0;s_n(C)]$ (and zero density elsewhere). The objective function of problem *PTF* then reduces to $\sum_{i \in N_{Rn}} p_i TF_i$. If all $r_i=1$, the resource units can be viewed as a simultaneously available parallel identical machines, and resource allocation decisions amount to assigning a job to a machine. Let $[k,i]$ denote the job that is scheduled at the i-th position on machine k, and let n_k denote the number of jobs assigned to machine k. If A, the set of precedence related activity pairs is empty (except for dummy start and end activities in the network),

$$TF_{[k,i]} = s_n - e_{[k,i]} - \sum_{j=i+1}^{n_k} d_{[k,j]} \tag{3.4}$$

and the objective function of problem *PTF* can now be written as follows:

$$\sum_{i\in N_{0n}} p_i TF_i = \sum_{k=1}^{a}\sum_{i=1}^{n_k} p_{[k,i]} TF_{[k,i]} = \sum_{i\in N_{0n}} p_i(s_n - e_i) - \sum_{k=1}^{a}\sum_{i=1}^{n_k} p_{[k,i]} \sum_{j=i+1}^{n_k} d_{[k,j]}\,. \qquad (3.5)$$

In other words, under the given assumptions, problem *DTF* boils down to verifying whether there exists a set R_f such that

$$\sum_{i\in N_{0n}} p_i(s_n - e_i) - U \geq \sum_{k=1}^{a}\sum_{i=1}^{n_k} p_{[k,i]} \sum_{j=i+1}^{n_k} d_{[k,j]}\,. \qquad (3.6)$$

Consider now the parallel machine scheduling problem $Pm||\Sigma w_j C_j$ of minimizing the weighted finishing time, for which the associated decision problem *Dm* has been shown to be NP-complete even for the 2-machine case (Bruno et al., 1974). For a set of jobs *J* to be processed, the expression to be minimized by *Pm* is

$$\sum_{i\in J} w_i C_i = \sum_{k=1}^{m}\sum_{i=1}^{n_k} w_{[k,i]} \sum_{j=1}^{i} d'_{[k,j]}\,, \qquad (3.7)$$

where C_i represents the finish time of job *i* in a machine schedule, d'_i denotes the duration of job *i* and w_i denotes its (integer) weight. Without loss of generality we can subtract the constant term $\sum_{i\in N_{0n}} w_i d'_i$ from (3.7). *Dm* comes down to answering the question whether for a given integer *L* there exists a schedule for *Pm* such that

$$\sum_{k=1}^{m}\sum_{i=1}^{n_k} w_{[k,i]} \sum_{j=1}^{i-1} d'_{[k,j]} \leq L. \qquad (3.8)$$

For a *given* selection of jobs to be processed on a particular machine, the *weighted shortest processing time first* (WSPT) rule, where the jobs are ordered on the machine in nondecreasing order of the ratio d'_j / w_j, minimizes the weighted sum of completion times (Pinedo, 1995). In other words, it is the assignment of jobs to the machines, not the ordering of the assigned jobs on a machine, that makes the scheduling problem complex. Thus it can be seen that the limit in (3.8) can be achieved if and only if a schedule exists for which

Eq. (3.8) *holds* and *every machine has a WSPT order.* (3.9)

Answering this question reduces to solving a *restricted problem DTF* with *reversed time horizon*. Order all jobs in *nonincreasing* order of the ratio (d'_j / w_j), which can be done in $(O(|J|\log|J|))$. Construct the input schedule *C* by putting all the jobs in this order in a contiguous chain. Choose $a{:=}m$ and assign probabilities $p_i{:=}w_i / \sum_j w_j$ and durations $d_i{:=}\ d'_i$. A machine allocation satisfying (3.9) can be found if and only if the answer to *DTF* is 'yes' when $U{=}{-}L / \sum_j w_j + \sum_{i\in N_{0n}} p_i(s_n - e_i)$. In other words, *Dm* polynomially reduces to *DTF*, which is therefore *NP*-complete.

Our findings in the previous paragraph indicate that problem *PTF*, though a heuristic, is hard to solve. It should be understood, however, that more complex distributions may require a longer encoding than the linear function we opted for, and hence may allow an exact solution algorithm to be polynomial in the input size. Also, input schedules will often be (semi-)active, which reduces the number of solutions to be considered. Nevertheless, it can be shown that every instance of *DTF* with arbitrary input schedule can be reduced to an adapted instance of *DTF* with active input schedule and larger number of jobs, so this need not worry us too much.

Our argument made above provides a sufficient rationale for the development of the branch-and-bound algorithm for solving both problems *PE* and *PTF*, which will be discussed in Section 4. During its search process, the branch-and-

bound procedure relies on constraint satisfaction concepts, which will be briefly introduced in the next paragraph, and further explored in Section 5.

3.4 *Robust resource allocation as a constraint satisfaction problem*

We will now restate problem P in a constraint satisfaction context. A *constraint satisfaction problem* (csp) is defined by a triple (F,D,Z) where F is a finite set of variables, D is a function which maps every variable k in F to a set of possible values D_k, called its domain, and Z is a set of constraints on variables in F (Tsang, 1993). The csp comes down to assigning to each variable a value from within its domain, such that the assignment satisfies all constraints. A *constraint satisfaction optimization problem* (csop) is defined as a csp together with an optimization function g which maps every solution tuple to a numerical value; the csop aims to identify a value assignment with optimal objective function. In this paper, the csop under consideration is P and the set of decision variables is the set of flows $F=\{f_{ij} \mid (i,j)\in TA\cup R(C)\}$. For $f_{ij}\in F$, $D_{ij}=[0;\min\{u_i,u_j\}]$ is the domain initially associated with f_{ij} (see Artigues, 2000), and Z contains constraint sets (2.1) and (2.2) and the requirement that $C\sim f$, which is implicit from F. Eq. (2.3) is also satisfied because every nonzero flow f_{ij} has $e_i\leq s_j$. The two optimization functions g_{PE} and g_{PTF} correspond with problem PE and PTF, respectively. The link between a solution f and g_{PE} and g_{PTF} is only through R_f, we write $g_{PE}(f)$ and $g_{PTF}(f)$, or $g(f)$ if precision is not necessary.

Theorem 3 states that the solution space for P will never be empty, provided that C is feasible. The following two observations are also useful.

Observation 1. *For two feasible flows f_1 and f_2, if $R_{f_1}\subset R_{f_2}$, then $g(f_1)$ can never be worse than $g(f_2)$.*
Proof. Immediate.

This observation will allow us to construct the flow f by selecting subset minimal feasible R_f. We will iteratively add arcs from $R(C)$ to R_f, until f is feasible. A similar remark appears in Igelmund and Radermacher (1983) about forbidden sets to be considered for the development of *ES*-strategies.

Observation 2. *For any feasible flow f, we can always find a feasible integer flow f°, such that $R_{f^\circ}\subseteq R_f$.*

Proof. f is a maximal flow in the network $G_{TA\cup R_f}$, and all capacities and lower bounds are integer. Thus, an integer maximal flow f° in the same network exists. f° may or may not use all arcs $TA\cup R_f$, hence $R_{f^\circ}\subseteq R_f$. □

This observation enables us to restrict the domains of the flows to natural numbers without loss of better solutions (and which is in line with the interpretation we gave to the flow values in Section 2). For $f_{ij}\in F$, D_{ij} can be represented by its lowest entry LB_{ij} and highest entry UB_{ij}, and we will represent the domains as intervals. A rationale for this approach will be provided in Section 5. Henceforward, an interval domain will refer only to the integer numbers contained in the interval.

The csop can be solved by enumerating all potentially valid assignments and storing the feasible one with minimal objective function value. Unfortunately, this

method is not practical due to the size of the search space. Thus, we are interested in methods to reduce the search space prior to starting and also during the search process. The basic idea of constraint propagation is to make implicit constraints more visible, thus allowing to detect and remove *inconsistent* variable assignments that cannot participate in any solution (Dorndorf et al., 2000). In Section 4, we will discuss a branch-and-bound procedure that efficiently enumerates the relevant sets R_f. This procedure relies on constraint propagation for search space reduction; the applied techniques will be examined in Section 5.

4. A branch-and-bound procedure

As the link between a solution tuple f and $g(f)$ is only through R_f, we will solve problem P by considering all subsets $E \subseteq R(C)$ that allow a feasible flow in network $TA \cup E$; one such set corresponds with at least one and mostly multiple f, with $R_f \subseteq E$. Such E will be referred to as 'feasible'. The details of the flow feasibility test are described in Section 6. Observation 1 enables us to restrict our attention to subset minimal feasible R_f. The branch-and-bound (B&B) procedure will solve problem P by iteratively adding arcs from $R(C)$ to R_f, until f is feasible.

Every arc in $TA \cup R(C)$ will have one of three states at any node in the B&B-tree: *required*, *forbidden* or *undecided*. If an arc is required, it has a lower bound greater than 0 on the flow it carries. If an arc is forbidden, the upper bound on its flow is 0. An undecided arc has both zero lower bound and nonzero upper bound. These bounds are established through constraint propagation (to be discussed in Section 5), in conjunction with branching decisions. Let α_p denote the set of required arcs and let ν_p denote the set of forbidden arcs in $R(C)$ at level p of the search tree. R_f contains all arcs in α_p. We will subsequently refer to the network $G_{TA \cup R_f}$ as the *partial network*.

If a feasible flow can be obtained in this partial network, we fathom the current node and backtrack. Otherwise, we will need further decisions to uniquely discern a feasible subset minimal set R_f that corresponds with the branching decisions up to the current level in the search tree. The branching decision itself boils down to the selection of an undecided arc $(i,j) \in R(C)$: the left branch is to set $LB_{ij}:=1$, so (from the current search node down the search tree) to include (i,j) in the partial network; the right branch is to set $UB_{ij}:=0$, so to forbid any flow across (i,j) and prohibit inclusion of (i,j) in α_p. Note that such binary branching suffices for our purposes: either an arc is in R_f, or it is not. The *amount* of flow across an arc is not important, only the question whether the flow is zero or nonzero.

For g_{PTF}, we continuously update TF_i values for all activities i in the partial network. We note that we can suffice with network $G_{A \cup \alpha_p}$, at level p. The corresponding objective function is updated as well, and at any node, this value is an *upper bound* on the maximal objective value achievable from that search node. For g_{PE}, the deterministic critical path length obtained with activity durations set equal to their expected values, is a *lower bound* for each convex cost function, hence also for the project makespan (Stork, 2000). In order to obtain a lower bound at every node of the search tree, we maintain a set of earliest starting times in $G_{A \cup \alpha_p}$ at level p based on expected activity durations; these earliest starting times are continuously updated. g_{PE} itself is computed via simulation, as soon as the partial network is found to admit a feasible flow.

Inspired by the single machine scheduling relaxation lower bound for stochastic project scheduling presented by Stork (2000), we can improve the efficiency of our algorithms. The single machine bound considers sets of precedence unrelated activities that are pairwise incompatible because of resource constraints, for which the smallest expected head plus smallest expected tail added to the sum of the expected durations of the activities, is a lower bound on the expected project makespan. In our problem, the sequencing problem for such sets of activities has already been completely solved, and it is evident that either directly or transitively, a precedence constraint $i \rightarrow j$ will be included for all pairs of incompatible activities (i,j) for which $s_j \geq e_i$. We may thus just as well include all those pairs into A from the outset. We shall refer to this extra measure as the *single machine rule*.

If, at level p of the search tree, $(i,j) \in TA \cup \alpha_p$, then combinations of the precedence relations defined by $TA \cup \alpha_p$ imply extra transitive relations. This is the case, for example, for all relationships (i,k), with $k \in \sigma_{TA \cup \alpha_p}(j) \setminus \sigma_{TA \cup \alpha_p}(i)$. All such inherent precedence relations discernible in $TA \cup R(C)$ are combined into β_p at level p. In appendix A, an algorithm for obtaining these implicit precedence relations is described; the set of *implicit arcs* is continuously updated rather than recalculated when needed. For any node in the search tree, it is clear that $\alpha_p \subseteq \beta_p$ and that we can extend R_f to include all arcs $\beta_p \setminus (TA \cup \nu_p)$ without incurring a change in objective value, compared with α_p. Branching now only needs to be done on arcs that are undecided and not implicit. Additionally, for *PTF* with $\gamma=0$, we can include as implicit all arcs $(i,j) \in R(C)$ for which TF_i becomes smaller than or equal to $s_j - e_i$: equation (3.2) will never be the only binding constraint. Note that for general γ, the float values may be negative. The inclusion of implicit arcs then goes as follows. Denote by $minTF_i$ the TF_i values that are obtained in the network $TA \cup R(C)$; it is clear that these quantities are lower bounds on TF_i. Hence, for general γ, we can include as implicit all arcs $(i,j) \in R(C)$ for which TF_i is set no larger than the expression $minTF_j - \gamma p_j E(L_j) + s_j - e_i$; this may also be more effective for the case $\gamma=0$. We shall refer to this rule as the *float dominance rule*.

After every branching decision, we will use constraint propagation to tighten the domains of the decision variables. A thorough discussion of this issue will be the subject of Section 5. We may detect infeasibilities through constraint propagation when an upper bound is set below a lower bound, or vice versa. In these cases, we could fathom the branch and backtrack. Remark that, by its very definition, we will not lose any solution tuple by the application of constraint propagation.

If no infeasibility is detected via the propagation of constraints, it could still turn out that the partial network cannot be completed to generate a feasible flow, simply because too many arc flows were forbidden. A fast detection of these situations allows to terminate the exploration of this part of the search tree. We will therefore resort to a second network: the *residual network* $G_{TA \cup R(C) \setminus \nu_p}$. As long as the residual network allows a feasible flow, it will be possible to select a set $R_f \subseteq R(C) \setminus \nu_p$ that allows a feasible flow (with the same "left branch" decisions, or externally imposed lower bounds on the flow across some arcs). At every level of the search tree, constraint propagation will be implemented only after this test is applied.

For the choice of the branching arc we will make use of the particular implementation of the test for the existence of a feasible flow. We shall defer the discussion of this topic until Section 6. A concise pseudo-code version of the branch-and-bound algorithm can be found in Appendix B.

5. Constraint propagation techniques

In the following, we shall develop some consistency concepts that will prove to be useful in reducing the search space. We shall restrict ourselves to administration of current domains, and not evaluate multi-dimensional assignments during constraint propagation (a similar decision was made by Nuijten, 1994). This approach was termed 'domain consistency' by Dorndorf et al. (2000), and suffices for our purposes.

5.1 *Some consistency concepts*

Define a csp to be consistent for a constraint c if $\forall k \in F$, $\forall q_k \in D_k$: $\exists(\tilde{q}_1,\ldots,\tilde{q}_{k-1},\tilde{q}_{k+1},\ldots,\tilde{q}_{|F|}) \in (\times_{l\in F, l\neq k} D_l)$: c holds for solution ($k=q_k$ and $\forall l \in F\setminus\{k\}$: $l=\tilde{q}_l$)). For problem $P=P(C)$, consider the following consistency definitions.

Definition 1: *outflow-consistency*.
$P(C)$ is outflow-consistent if the csp is consistent for (2.1).

Definition 2: *inflow-consistency*.
$P(C)$ is inflow-consistent if the csp is consistent for (2.2).

Definition 3: *schedule-consistency at period $t°$*.
$P(C)$ is schedule-consistent at period $t°$ if the csp is consistent for (2.4) with $t=t°$.

For $f_{ij}\in F$, D_{ij} can be represented by its lowest entry LB_{ij} and highest entry UB_{ij}, since the consistent domain for the described constraints will always take the form of an interval on $\mathbb{N}$. This can be seen easily: all domains are initialized as an interval, and all considered constraints are of the form $\sum_i x_i = e$, e integer. Constraint propagation can only cut away upper and lower parts of intervals, but never rule out only intermediate values (bound and domain consistency, as defined by Dorndorf et al. (2000), are equivalent in our case).

We propagate constraints to achieve desired consistency more or less as described in Davis (1987), which is related to algorithm AC-3 to obtain arc-consistency in binary constraint satisfaction problems (Mackworth, 1977). The idea is to use a queueing structure, where a constraint is added to the queue when one of its arguments is changed, and removed when propagated. We tighten the upper and lower arc flow bounds as follows.

Consider constraints (2.1), for a particular i-value (outflow consistency). We can achieve consistency of P for this constraint by tightening our bounds in the following way:

$$LB_{ij} := \max\{LB_{ij}\,;\, u_i - \textstyle\sum_{k\neq j} UB_{ik}\} \qquad \forall j \in N_0 \qquad (5.1)$$

$$UB_{ij} := \min\{UB_{ij}\,;\, u_i - \textstyle\sum_{k\neq j} LB_{ik}\} \qquad \forall j \in N_0 \qquad (5.2)$$

Consistency is achieved if we iterate (5.1) and (5.2) as long as either one of the expressions changes the argument of any other. It is clear that no feasible values are deleted from the domains, and also that after tightening the bounds, the csp is consistent for the constraint under consideration.

For constraints (2.2) (inflow consistency) for one selected i, we obtain similar equations:

$$LB_{ji} := \max\{LB_{ji} ; u_i - \sum_{k \neq j} UB_{ki}\} \qquad \forall j \in N_n \qquad (5.3)$$

$$UB_{ji} := \min\{UB_{ij} ; u_i - \sum_{k \neq j} LB_{ki}\} \qquad \forall j \in N_n \qquad (5.4)$$

For constraints (2.4) (schedule consistency) for a particular t, identical reasoning yields, with P_t defined here as $TA \cup R(C) \mid e_i < t \leq s_j$:

$$LB_{ij} := \max\{LB_{ij} ; a - r(N_t) - \sum_{(k,l) \in P_t, \neq (i,j)} UB_{kl}\} \qquad \forall (i,j) \in P_t \qquad (5.5)$$

$$UB_{ij} := \min\{UB_{ij} ; a - r(N_t) - \sum_{(k,l) \in P_t, \neq (i,j)} LB_{kl}\} \qquad \forall (i,j) \in P_t \qquad (5.6)$$

Tightness of (5.5) and (5.6) clearly depends on $r(N_t)$ ($r(N_t)=a$ allows to eliminate all arc flows in P_t) and the constrainedness of the other arcs in P_t. Following Davis (1987), we choose constraints off the queue in fixed sequential order, rather than LIFO or best-first. FIFO would be a valid alternative, but requires more memory manipulation: for every update, we now register whether it is on the queue or not simply by triggering an associated boolean variable. In the appendix of Davis (1987) (lemma B.13), theoretical evidence is provided for applying independent updates first, "independent" meaning that the input arguments are not output of other updates. We shall come back to this below.

Consider the example schedule of Figure 6, where resource availability per period is 3 units; the Gantt Chart does not imply resource allocations.

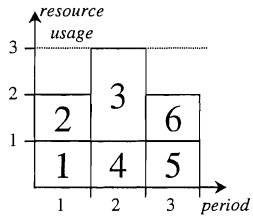


Figure 6. Example schedule.

The flow domains are initialized as mentioned above for every element in $TA \cup R(C)$, for instance $LB_{02}=0$ and $UB_{02}=2$. Schedule consistency for period 2 tells us that $UB_{ij}=0$ for every $(i,j) \in (\{0,1,2\},\{5,6,7\})$. Inflow consistency yields $LB_{01}=LB_{02}=1$, and outflow consistency gives $LB_{57}=LB_{67}=1$. Outflow consistency of activity 0 and inflow consistency of activity 7 tighten UB_{03} and UB_{37} to 1. The domains are now consistent according to definitions 1, 2 and 3. If we externally impose $LB_{03}:=1$, then outflow consistency tightens $UB_{04}=0$, because of outflow consistency in activity 0. If we additionally choose $LB_{13}=1$, then $UB_{14}=0$ (outflow consistency in 1), $LB_{24}=1$ (inflow consistency in 4), and $UB_{23}=0$ (outflow consistency in 2).

In the following theorem, we prove the intuitive idea that upper bounds can never be tightened because of transitive updates, caused solely by an initial upper bound revision.

Theorem 5. *Consider a set of values X containing integer elements such that $\sum_{i \in X} i = b$, b integer, and consider corresponding upper and lower bounds UB_i and LB_i for every $i \in X$, that are jointly consistent for the single constraint. For any $i^* \in X$, if we externally lower UB_{i^*} to any value within $[LB_{i^*};UB_{i^*}[$, then LB_j values for $j \neq i^*$ may need to be updated to maintain (domain) consistency, but it is never required to update LB_{i^*} or UB_j, for any j.*

Proof. UB_{i^*} is an argument to the updates of LB_j, $j \neq i^*$. Hence, these values might need to be updated when tightening UB_{i^*}. If no LB_j are updated, consistency is not lost and the theorem holds. Otherwise, at least one variable j^* is updated. Denote the new lower bounds by LB'_j, for all j. It holds that

$$LB'_{j^*} = b - \sum_{i \in X \setminus \{j^*\}} UB_i \,. \tag{5.7}$$

Consider now any $k \in X \setminus \{j^*\}$. UB_k is still consistent if

$$\exists (\tilde{x}_1, \ldots, \tilde{x}_{k-1}, \tilde{x}_{k+1}, \ldots, \tilde{x}_{|X|}) \in (\times_{i \in X, i \neq k} [LB'_i; UB_i]) : UB_k = b - \sum_{i \in X \setminus \{k\}} \tilde{x}_i \,.$$

Such values can be chosen, as (5.7) holds. For j^*, the same reasoning can be followed if (5.7) holds for more than one element of X. Otherwise, if only variable j^* is updated, its upper bound need not be recalculated anyway because its own lower bound is not an argument to its upper bound update expression. Also, since UB_k-values remain unchanged, none of the arguments of the update-expression for LB_{i^*} is ever altered, so it remains constant. □

An alternative line of proof shows that candidate upper bound update $UB_k^{cand} := b - \sum_{i \in X \setminus \{k\}} LB'_i$ is larger than or equal to UB_k, because of (5.7). A theorem similar to Theorem 5 can be shown to hold for lower bound tightening. These theorems will allow us to reduce the constraint propagation effort: if for any $(i,j) \in TA \cup R(C)$, UB_{ij} is tightened because of outflow consistency, we do not have to apply update (5.1) for activity i to guarantee consistency. Similar conclusions can be derived mutatis mutandis for lower bounds and other consistency concepts. The theorem can be extended to imply that if a bound is tightened, we should re-add to the queue all updates that carry the bound as an argument, except the ones for which the bound under consideration has just been used as input to the current update itself. As we actually use only one boolean variable per set of updates, e.g for all $j \in N_0$ together in (5.1), we will not fully implement this extension.

A small example will illustrate the impact of the theorem. Consider five variables $x_1, \ldots, x_5$, that sum up to $b=8$. A consistent set of domains is $d_1=[2;3]$, $d_2=[0;2]$, $d_3=[0;2]$, $d_4=[1;1]$ and $d_5=[1;2]$. If we tighten UB_3 to 0, consistency can be maintained via $LB_1:=3$, $LB_2:=2$ and $LB_5:=2$. No upper bounds, nor LB_3, need to be changed to maintain consistency.

5.2 *Application of the consistency updates in the branch-and-bound procedure*

At level 0 of the search tree of the branch-and-bound algorithm discussed in Section 4, we tighten the domains of F by making them consistent for some set of constraints X_0. As already mentioned, if we branch on f_{ij} at level p, the left branch is to impose $LB_{ij}:=1$, the right branch is to set $UB_{ij}:=0$; call the applied constraint c_p. We propagate c_p and make the domains consistent for $X_p := X_0 \cup (\cup_{i=0,\ldots,p} c_p)$; X_p will denote the set of constraints at level p of the search tree.

At level 0, we make the csp consistent for X_0 by repeated application of updates from (5.1)-(5.6), depending on the selection of X_0. The queue of bound updates is initialized to include (5.1)-(5.4) for all relevant i-values, and (5.5)-(5.6) for all selected time periods t. After execution of an update, this update is removed from the queue. If a bound is tightened, all updates that carry the bound as an argument are re-added to the queue, except the ones that can be omitted by Theorem 5. At level $p>0$, we make the csp consistent for X_p in the knowledge that it is consistent for

X_{p-1}. If we decide to branch on flow f_{ij}, the domain of f_{ij} is reduced on one side (the branching decision is implemented), and the queue of updates is initialized with only the updates having LB_{ij} (left branch) or UB_{ij} (right branch) as argument. In effect, rather than adding a new constraint, we split up the domain into two disjoint subsets, one of which is singleton {0}, which is unlike the classical approach in constraint satisfaction to branch on every single domain value separately.

If we choose to pursue inflow and outflow consistency, based on the re-ordering of N in increasing starting times and the wish to execute independent and least dependent updates first, we shall run through the updates to check their boolean update indicator in the following way.

1) i:=n–1;
2) check outflow-indicators (LB and UB separately) for i and inflow-indicators (LB and UB separately) for n–i; if marked, perform updates, unmark current update and mark relevant updates (as described above);
3) if i>0 decrease i and go to 2, else if some indicators are marked go to 1, else return: in- and outflow consistency is guaranteed.

For schedule consistency, we will simply run through the selected time periods in chronological order, after (one run of) the updates for flow consistency.

6. Testing for the existence of a feasible flow

As already mentioned, we construct feasible solutions based on a check for the existence of a feasible resource flow, rather than on the explicit "destruction" of all (minimal) forbidden sets. In this section, we will present a simple way to test for the existence of a feasible flow in a given network. The feasibility test applies maximal flow computations to a transformed version of the network. This transformed network will also help us in choosing a branching arc.

6.1 *Feasible flow test*

In this section, we will present a simple way to test for the existence of a feasible flow in a given network. Consider a network $G_{TA\cup E}$, $E\subseteq R(C)$. The flow generation and conservation constraints (2.1) and (2.2) impose strict requirements on the flow passing through any node. We construct a new network $G'_{TA\cup E}$ as follows. Following Ford and Fulkerson (1962), we switch from bounds on node flow to bounds on arc flow by duplicating each node $i\in N_{0n}$. We replace node i by two nodes i_s and i_t and add arc (i_t,i_s) to the network, with the upper bound on flow (i_t,i_s) equal to its lower bound, both being equal to u_i. Rename nodes 0 and n as 0_s and n_t, respectively. All arcs entering i in $G_{TA\cup E}$ now lead to i_t, all arcs leaving i now emanate from i_s. A similar network transformation was suggested by Naegler and Schoenherr (1989). It is clear that a feasible flow f exists in $G_{TA\cup E}$ with $R_f\subseteq E$ if and only if the arcs of $G'_{TA\cup E}$ can carry a feasible flow that respects all arc flow bounds. Node i_t can be interpreted as the start of activity i, node i_s as its completion. Figure 7(a) shows the network $G'_{TA\cup E}$ for the project shown in Figure 1 for the case $E=\emptyset$.

We will verify the existence of a feasible flow by solving a maximum flow problem on a transportation network $G_{TA\cup E}{}^*$ which is obtained from $G'_{TA\cup E}$ in the following way. Augment network $G'_{TA\cup E}$ with a source node s, a sink node t, and an arc (t,s). Every arc (i_t,i_s) in $G'_{TA\cup E}$ is replaced by arcs (i_t,t) and (s,i_s). Capacity function

c assumes the following values: $c(s,i_s)=c(i_t,t)=u_i$, $\forall i \in N$, all other capacities equal to ∞. Figure 7(b) shows the network G_{TA}^* obtained from the network G'_{TA} of Figure 7(a).

Denote by $\mu(E)$ the maximal s-t flow value in $G_{TA\cup E}^*$, and h a corresponding maximal flow. It is clear that h satisfies the following two conditions:

$$h(i_s,\{j_t \mid j \in N\}) \le u_i \qquad \forall i \in N_n \qquad (6.1)$$

$$h(\{i_s \mid i \in N\},i_t) \le u_i \qquad \forall i \in N_0 \qquad (6.2)$$

If we define $\mu_{max}:= a+r(N_{0n})$, we see that $\mu(E) \le \mu_{max}$. Equality $\mu(E)=\mu_{max}$ holds if and only if a maximal s-t flow (henceforward denoted as "maximal flow") in $G_{TA\cup E}^*$ saturates all source and sink arcs, so that conditions (6.1) and (6.2) are satisfied as an equality. The following theorem now imposes an interesting condition on a feasible flow f.

Theorem 6. *For $E \subseteq R(C)$, a feasible flow f exists with $R_f \subseteq E$ if and only if $\mu(E)=\mu_{max}$.*

Proof. (for the "if" part of the condition) Call h a flow that realizes $\mu(E)$. For every $(i,j) \in TA\cup E$, set $f_{ij}:=h(i_s,j_t)$. The constructed f only uses $TA\cup E$ and from equality in (6.1) and (6.2) follows equality in (2.1) and (2.2). Also, as $TA\cup R(C)$ leads to an acyclic graph and $E \subseteq R(C)$, condition (2.3) is met.
(for the "only if" part of the condition) Analogously, h-values can be derived from feasible f that satisfy (6.1) and (6.2) as an equality. □

The maximal flow in network G_{TA}^* of Figure 7(b) amounts to $\mu_{max}=8$. Theorem 6 allows us to conclude that a feasible flow is attainable in G_{TA}. The flow network corresponding with one such flow was shown Figure 2(b).

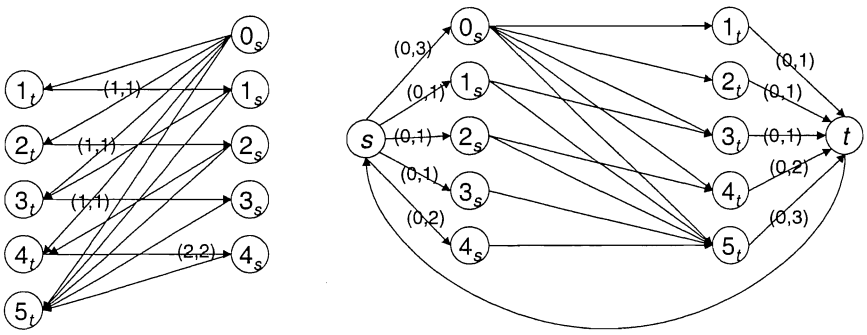


Figure 7. G'_{TA} and G_{TA}^* for the example problem ($E=\varnothing$). Lower and upper bounds on arc flow are as indicated, otherwise $(0,\infty)$.

We will apply Theorem 6 during the course of the branch-and-bound algorithm described in Section 4 to test for the existence of a feasible flow in both the partial network $G_{TA\cup R_f}$ and the residual network $G_{TA\cup R(C)\setminus v_p}$. For this purpose, we will use the *extended* networks $G_{TA\cup R_f}{}^*$ and $G_{TA\cup R(C)\setminus v_p}{}^*$. At any time during the course of the algorithm, LB_{ij} and UB_{ij} are imposed as lower and upper bounds on the flow on (i_s,j_t) in both the extended partial and the extended residual network, if that arc is present in the network under consideration. These bounds have been tightened based on the branching decisions and constraint propagation. As

mentioned above, if $\mu(TA \cup E)=\mu_{max}$ for any $E \subseteq R(C)$, then (2.1) and (2.2) hold for the associated flow in $G_{TA \cup E}$, and consequently inflow and outflow consistent bounds will inherently hold. Also, for every feasible f, it holds that $R_f \subseteq R(C)$, with C the input schedule, such that equation (2.4) is valid. Consequently, imposing any of the described consistency concepts on the upper and lower bounds applied in the partial and residual network, will never eliminate parts of the solution space of (minimal) feasible flows from consideration.

At level 0 of the search tree, flow (i_s,j_t) in both extended networks is initialized at LB_{ij} for every (i,j) that is admitted in the network under study, and we use a simple and efficient version of the classical labelling algorithm (Ford and Fulkerson, 1962) to maximize flow, the shortest augmenting path algorithm (Edmonds and Karp (1972), Ahuja et al. (1993)), which can be very easily implemented because of the bipartite network structure of the extended graphs. This structure invalidates the need for distance labels. At any newly entered node, we first check the extended residual network to see if flow can be set to μ_{max}. If so, we check whether the partial network admits a feasible flow, else backtrack.

The flows in the two extended networks are maintained on an incremental basis, rather than that they be recalculated restarting from the lower bounds every time a feasibility check is required. Whenever an upper bound UB_{ij} is tightened by amount Δ, we maintain a feasible flow by reducing flow on arcs (s,i_s), (i_s,j_t), (j_t,t) and (t,s) by Δ. Whenever a lower bound LB_{ij} is tightened, we try to reconstitute a feasible flow by looking for a feasible circulation in the corresponding extended network that contains (i,j) as a forward arc. Again we will iterate across shortest augmenting paths from j to i, so actually along shortest augmenting cycles. An augmenting cycle that contains (t,s) as a backward arc (which lowers total flow) will only be considered if all other possibilities have been checked, on the other hand, augmenting cycles that increase total flow will be tried first (possibly at the expense of an augmenting cycle that is 2 arcs longer than achievable). This incremental approach supports the use of the shortest augmenting path algorithm, which can be implemented with very little overhead.

Every time a bound is tightened, because of constraint propagation or branching decisions, we keep the flow in the extended partial network feasible, but not necessarily maximal, contrary to the extended residual network. The reason is that verification of the existence of a feasible flow in the partial network will regularly not be necessary, namely in those cases where the residual network does not admit a feasible flow anymore. Only when all updates have taken place and are acceptable, we maximize flow in the extended partial network. When backtracking, we again keep partial flow feasible but not necessarily maximal, for the same reason.

Remark that when a lower bound equal to zero is tightened, this implies in the partial network also an increase in the corresponding upper bound, from 0 to the value of the bound in the residual network. Augmenting cycles to adapt flow to a tightened lower bound in the extended residual network are only considered if they do not include arc (t,s) backward (forward is impossible given the fact that maximal flow already equals μ_{max}). This comes down to verifying whether exists a flow that respects an additional lower bound on (t,s), equal to current flow. The trivial augmenting cycle to deal with tightened upper bounds does require an extra pass of a separate maximal flow algorithm in the extended residual network.

6.2 *Branching choice*

In case the partial network at hand does not yield a feasible flow and the residual network indicates that at least one such flow can still be found by further branching, it was mentioned earlier that we shall choose our branching arc from the set of arcs that are not implicit and not forbidden; we shall term these last arcs the *eligible* arcs. Actually, roughly speaking, we wish to increase the flow in the (extended) partial network by adding arcs to it, at least in the left branch. However, it turns out that the set of eligible arcs that might allow the flow in the partial network to increase, is rather limited. When our labelling algorithm to obtain maximal flow in the partial network terminates, we are left with a number of nodes i_s ($i \in N_n$) that are labelled, for some of which the corresponding arcs (s,i_s) are not yet saturated, and for all unlabeled i_s the incoming arc is saturated. In the same way, the nodes i_t ($i \in N_0$) that are labelled are reachable via an augmenting path from s, and the unlabelled i_t are not. In order to obtain an increase in flow in the partial network, we need to detect an augmenting path from s to t. In the knowledge that such a path did not exist before branching, we see that a new augmenting path will only arise if the branching arc itself or one of the other arcs that become implicit by the branching decision, has a labelled origin node and an unlabelled destination node, for the arc in question can only figure as forward arc in a flow augmenting path. Remark that when we have to choose an arc for branching, flow in the partial network will be maximal, not only feasible.

Note that the set of eligible arcs (i,j) with labelled i_s and unlabelled j_t suffices to guarantee correct execution of the branch-and-bound procedure at all times. We can extend this set to include all arcs (k,l) such that either k_s is labelled, or there exists an implicit predecessor m of k with m_s labelled, and either l_t is unlabelled, or we can find an implicit successor m of l with m_t unlabelled. A last option would be to evaluate the entire set of all eligible arcs for inclusion. Remark that other options are still possible: we might include (i,j) if setting (i,j) required will, via constraint propagation and/or transitive precedences, cause other arcs (k,l) to become implicit as well, but such reasoning would entail excessive computations.

Once we have constructed a set of arcs to choose from, we shall select one by means of an evaluation criterion, based on cost and/or benefit implications. *Cost* considerations would look at the deterioration of the objective function by inserting the arc in the partial network; this value is only approximate, as other arcs might be inserted as well because they become required in the constraint propagation phase following the branching choice. Remark that arcs that become implicit in the left branch because of the branching arc itself, do not induce extra costs (by their very definition). The *benefit* of adding an arc to the *extended* partial network is that it advances towards a feasible flow in the *original* partial network. This benefit can be approximated by looking at the (extended) residual network, and summing flow in this network for the arcs foreseen to be added in the partial network, which can be considered as an indicator for the extra amount of flow we might be able to send across the partial network. We refer the reader to Section 7 for further details. We initially limit the set of candidate arcs to include only the arcs that have nonzero flow in the residual network; this set will almost never be empty. The opposite approach, start with considering only arcs that have *no* flow in the residual network, would aim to identify quickly from the outset infeasible allocations, but is not considered.

Two more selection rules will be introduced in our branch-and-bound procedure. If cost of insertion *seems* to be zero, we might immediately select the arc and disregard all other alternatives (immediate left branch rule). If on the other hand cost of insertion will *certainly* induce the objective value to deteriorate at least until the best found feasible solution so far, we might also contemplate selection of the arc in question to be branched on: we will then be able to immediately prune the left branch, and continue the right branch (immediate right branch rule). This last rule could also have been implemented as a separate module that simply forbids arcs that are overly costly, but it can be nicely fit in to the branching framework

7. Computational experiments

We have implemented our algorithms in C++, using the Microsoft Visual C++ 6.0 programming environment, on a Dell XPS B800r personal computer with Pentium III processor. We will compare the performance of our algorithms with the one presented in Artigues et al. (2000), which tries to construct a flow that is compatible with an input schedule. The algorithm is presented in Appendix C (`AMR`). In Bowers (1995), it is noted that the float values resulting form the resource network are conditional on the particular resource allocation employed, as was mentioned already by Wiest (1964). The way in which the resource links are actually derived remains unclear however, and the author suggests this to be a management decision. Naegler and Schoenherr (1989) are mainly concerned with selecting the appropriate time/resource trade-offs, and do not offer an algorithm to derive an initial feasible flow. A *practical* worst case allocation for a given input schedule would load the resource units as heterogeneously as possible. We will use the algorithm `worst_case` outlined in Appendix C as a reference.

The probability p_i of disturbance of activity i will be drawn from a uniform distribution on the interval [0;0.7]. We assume exponential activity disturbance lengths, with average length *if* disturbed equal to the activity duration itself. The scheduling logic is branch-and-bound (Demeulemeester and Herroelen, 1992, 1997); the scheduling algorithm is truncated after 1 minute of CPU-time.

The scheduling problems will be generated by *RanGen*, a recently developed network generator for activity-on-the-node networks, which has the advantage of being able to generate so-called "strongly random" networks. We specify values for the order strength *OS* (values 0.2 and 0.5), the resource factor *RF* (0.8) and the resource constrainedness *RC* (0.2 and 0.4); for a thorough discussion of the network generator and the parameters involved, we refer to Demeulemeester et al. (2000). For a specified number of activities, for each of the 2×1×2 parameter settings, we generate 25 problem instances, resulting in 100 instances in total. The alert reader might note that not the entire domain of the problem parameters is covered, our choices are logical however: if *OS* is large, a lot of precedence constraints are present from the outset, and $R_f=\varnothing$ will regularly already yield a feasible flow. If *RF* is low, only a small number of activities have a nonzero resource usage, and little options remain for allocation. If *RC* is low, more activities can in general be scheduled in parallel, and this increases the number of possible allocations. For larger resource constrainedness, a general *ES*-policy would still have to make sequencing decisions, but our input schedule will already have already made most important choices.

Problem *PTF* with $\gamma=0$ relies only on integer arithmetic with regards to float computations, whereas $\gamma>0$ and *PE* require floating point values for the floats and the early-start times, respectively; we will not study the case $\gamma=0$ separately in what

follows. We will reduce the float values to integers by rounding them to the lower integer after multiplication with factor 100,000. All upper and lower bounds are maintained as index to the "bound"-array. This array consists of successive four-dimensional entries, the first of which is the bound value itself and the second the position in the array where the previous bound can be found; the third and fourth entry identify the arc the bound value is associated with. If a bound is changed for the first time at the current level of the search tree, we create a new entry in the array; updating bounds within one level is executed at the same entry. Similarly, we maintain a list per level of all activity pairs that were made implicit at that level. These data structures will allow efficient backtracking. The need to undo a required or forbidden status for an arc on backtracking will be spotted while restoring the bounds themselves. On storing a set R_f in case of a new incumbent, we will retain not all implicit arcs, but only the arcs that carry flow in the partial network.

7.1 *The number of simulations for PE*

Examination of code execution by means of a profiler has shown that for problems with 30 (non-dummy) activities, some 95% of CPU-time is absorbed by the evaluation of the objective in the *PE* solution algorithm by means of simulation. This part of the algorithm is clearly the bottleneck with regards to time consumption. Maximal flow computations and consistency updates make up the larger part of the remainder of the running time, the former requiring about half the time of the latter.

Stork (2000) works with gamma distributions for activity durations and states that 200 samples turn out to provide a reasonable trade-off between precision and computational effort. The author points out that for strongly varying processing times, as in the case of the exponential distribution, running times may increase. This is certainly also the case for our mixture of exponentials. Moreover, examining the standard deviation of the percentage deviation of simulated makespan versus the "true" value obtained from 2000 simulations, we conclude that the required number of simulation runs decreases with the number of activities, and will implement the following number of runs: 350 simulations (40 activities), 400 simulations (30 activities) and 450 simulations (20 activities), which has the additional advantage of reducing the simulation effort for larger problem instances. These choices correspond with a standard deviation of just below 1%. The final evaluation of the performance of an allocation after termination of the algorithm will be carried out by means of 2000 simulations.

7.2 *The branching choice*

A first choice that has to be made with regards to what activity pair to branch on, is which set of activity pairs to choose from. For problem *PTF* with γ=0.1, for projects with 30 activities, we obtain the results shown in Table 1, considering only the activity pairs with residual flow, and choices based on "cost/benefit". All problems are solved to optimality. It is clear that we opt for the smallest subset of arcs to evaluate for branching.

Table 1. What subset of the eligible arcs to choose from.

	labelled origin, unlabeled destination	labeled-unlabeled, direct and transitive	all eligible arcs considered
avg. nr. nodes	2190	3053	5132
avg. CPU (sec)	0.08683	0.1521	0.19681

Next we wish to decide what evaluation criterion to use to evaluate candidate branching arcs, in the same setting as described above. We refer to Table 2a for computational results with regards to benefit and cost/benefit considerations in this choice. If only cost of insertion is the evaluation criterion, the average number of nodes is 2307, average CPU-time is 0.08794 seconds. When we simply take the first encountered "allowed" arc (no evaluation criterion), the average number of nodes is 6957 and the average CPU-time is 0.2183 seconds. Alternatives for the benefit (extra flow) evaluation of an arc, are to consider only the flow in the residual network on that arc, or the sum of all residual flow of arcs that will immediately (by precedence, not constraint propagation) become implicit but were not before.

Table 2a. Choice of the evaluation criterion for *PTF*.

		residual flow on new implicit arcs	only residual flow on arc itself
cost/benefit	nr. nodes	2190	2172
	CPU (sec)	0.08683	0.08524
only benefit	nr. nodes	1842	1957
	CPU (sec)	0.05879	0.05839

From Table 2a, we conclude that cost considerations are not that important on evaluating the branching decision: not only are they time consuming, but they also require more nodes in the search tree. It appears more important to quickly obtain a feasible flow, than to try to obtain such a feasible flow in the most cost-effective way; this occurs in spite of the advantage of additional immediate right and left branch rules discussed above. Table 2b gives the corresponding results for problem *PE*, from which similar conclusions can be drawn. A CPU-time limit of 150 seconds is imposed. We have included in the table the number of problems for which the optimum is guaranteed (was 100 above).

Table 2b. Choice of the evaluation criterion for *PE*.

		residual flow on new implicit arcs	only residual flow on arc itself
cost/benefit	nr. nodes	16125	16006
	CPU (sec)	19.17	19.20
	optimal	91	91
only benefit	nr. nodes	12588	13055
	CPU (sec)	16.40	16.81
	optimal	93	93

7.3 The effect of constraint propagation

For the best obtained setting in the previous paragraph, we present the computational results for *PTF* with regards to the different options for constraint propagation in Table 3a. It is to be noted that schedule consistency with a–N_t=0 is not implemented as a separate consistency update, but simply forbids all arcs in P_t at initialization.

Table 3a. The impact of constraint propagation in *PTF*.

	schedule and flow consistency	only flow consistency	only schedule consistency	none
avg. nr. nodes	1842	2499	5914	7981
avg. CPU (sec)	0.05879	0.0648	0.1215	0.14042

Table 3b gives the corresponding output for *PE*.

Table 3b. The impact of constraint propagation in *PE*.

	schedule and flow consistency	only flow consistency	only schedule consistency	none
avg. nr. nodes	12588	17149	25046	24158
avg. CPU (sec)	16.04	16.82	16.4	17.43
optimal	93	92	93	92

For both *PTF* and *PE*, maintaining both schedule and flow consistency with regards to the domains of the flow values during the course of the algorithms is clearly the most efficient. We notice a trade-off between speed of calculations and tightness of the domains: considering the large difference in average number of nodes of the search tree, only a less than proportionate gain in average CPU-time is obtained. This is because the number of leaves in the search tree to be evaluated in the search tree remains the same, and simulation makes up the largest part of computational effort, and also because pursuing consistency consumes extra time per node of the tree. Nevertheless, judging from the computational results, this is more than offset by the benefits: consistency will lead to less infeasible branches (forbidden arcs are recognized sooner), shorter branches (required arcs are identified sooner and hence more arcs become implicit), and also the domains are tighter in the maximal flow computations, such that these take less computation time. Another reason for the

limited time-impact with *PE* is that we do not exclude problems that were interrupted because of the time limit, for the average computation.

7.4 Speeding up the algorithms?

Table 4 illustrates the impact of the float dominance rule presented in section 4 on the search effort in *PTF*. This rule was already implemented in the code tested in the previous paragraphs.

Table 4. Impact of the float dominance rule.

	PTF float dominance ON	*PTF* float dominance OFF
avg. nr. nodes	1842	2427
avg. CPU (sec)	0.05879	0.0631
optimal	100	100

Table 5 shows the impact of the single machine rule in section 4 for both *PTF* and *PE*. The rule strengthens the initial bound based on Jensen's inequality in the case of *PE*, and tightens the upper bound corresponding with the current solution in *PTF*. However, the impact is minimal for both *PTF* and *PE*.

Table 5. Impact of the extended lower bound.

	PTF single machine rule OFF	*PTF* single machine rule ON	*PE* single machine rule OFF	*PE* single machine rule ON
avg. nr. nodes	1842	1840	12588	12516
avg. CPU (sec)	0.05879	0.05851	16.04	15.96
optimal	100	100	93	93

Finally, we have also implemented *AMR* as initial solution in *PTF* and *PE*. The results can be found in Table 6. Again we achieve noticeable but limited improvements. We suspect that the larger time gain for *PTF* with respect to *PE*, is due to the quality of the upper bound in *PTF* which is better than the quality of the lower bound in *PE*.

Table 6. Impact of initial solution.

	PTF without initial solution	*PTF* with initial solution	*PE* without initial solution	*PE* with initial solution
avg. nr. nodes	1840	1790	12516	12384
avg. CPU (sec)	0.05851	0.049858	15.96	15.92
optimal	100	100	93	93

7.5 *Expected makespan comparisons*

Problem *PTF* was presented as an approximation for *PE*, and has a parameter γ, which quantifies the degree in which we propagate (expected values of) disruptions throughout the network, as discussed in Section 3.2. The performance of *PTF* as a function of γ is investigated in Figure 8. These results stem from a dataset consisting of 200 problems with 30 activities, made up of 25 problems per setting of *OS*×*RF*×*RC*, where *RF* now takes on 2 values (0.7 and 0.9) instead of 1, the other parameters remaining unchanged. We doubled the size of the dataset compared with the previous sections, such that the pattern in the graphs be clear.

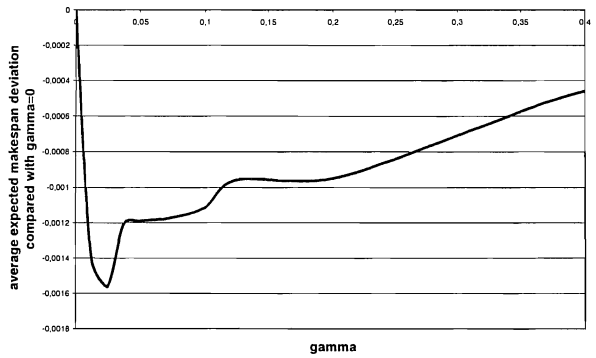


Figure 8. Average expected makespan deviation compared with γ=0.

The value of γ can be found on the abscissa of the graph, the ordinate gives the average expected makespan deviation compared with the case γ=0. We notice that nonzero γ-values allow us to obtain better results than the base case of equation (3.2), and this for a whole range of values, as the function only takes on positive values beyond γ=0.75. Small values of the parameter do yield the best results, we advocate γ=0.025. We did not examine the behaviour of optimal γ as a function of problem size. Figure 9 shows the average expected makespan deviation as a function of ω_0. Parameter ω_0 influences TF_n: $\omega = \omega_0.s_n$. Apart from a small bump, the obtained curve is smooth and we will choose ω_0=0.2 in the following.

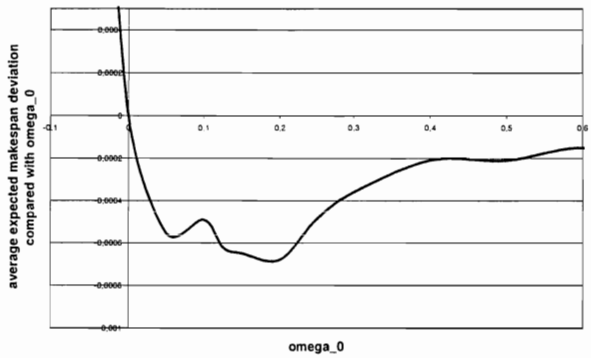


Figure 9. Average expected makespan deviation compared with ω_0=0.

We shall now turn to a comparison of the different allocation algorithms (*PE*, *PTF* with γ=0.025 and ω_0=0.2, *AMR*, and *worst-case*), as a function of problem size. Tables 7-9 provide an overview of the performance of the various algorithms for problems with 20, 30 and 40 non-dummy activities; our datasets are still composed of 200 activities. CPU-time for the simple heuristics *AMR* and *worst-case* is negligible. The average number of nodes for *PTF* strongly exceeds the number for *PE* for n=41, as the number of nodes that can be visited within the time limit is around 85,000 for *PE*, and some 2,500,000 for *PTF*.

Table 7. Comparison of various allocation algorithms for n=21.

	PE	*PTF*	*AMR*	*worst-case*
avg. nr. nodes	2546	64	-	-
avg. CPU (sec)	2.93	0.004	-	-
avg. dev. *PE*	0%	0.55%	1.17%	1.65%
optimal	99%	100%	-	-

Table 8. Comparison of various allocation algorithms for n=31.

	PE	*PTF*	*AMR*	*worst-case*
avg. nr. nodes	14793	6120	-	-
avg. CPU (sec)	22.9	0.23	-	-
avg. dev. *PE*	0%	0.63%	1.25%%	1.68%%
optimal	88.5%	100%	-	-

Table 9. Comparison of various allocation algorithms for n=41.

	PE	*PTF*	*AMR*	*worst-case*
avg. nr. nodes	19524	74413	-	-
avg. CPU (sec)	37.23	3.64	-	-
avg. dev. *PE*	0%	0.70%	1.32%	1.58%
optimal	80%	98%	-	-

Problem *PE* is efficiently approximated by *PTF* and this approximation performs better than the simple allocation heuristics, although some improvements seem still in order; choice of the parameters ω_0 and γ as a function of n is one area of possible improvement. At the same time, *PTF* can be solved a lot quicker than *PE* (although this difference decreases with n). *AMR* turns out to perform significantly better than *worst-case* in our computational experiments. One reason for this might be the fact that activities are always scanned for available resources starting from 0, and since the activity index is an indication of the position in the schedule, flows will tend to introduce always more or less the same "slack", thus spreading available excess capacity evenly across the different resource units. Information about disturbance probabilities is not taken into account, however.

7.6 Dependent activity durations

When a reasonable amount of disturbances occur, which was our initial experimental setting, the assumption of independent activity durations is often unrealistic, especially in a project environment, but perhaps less in a shop scheduling environment; this issue was discussed above. Cumulation of disturbances will normally occur, but management will do its best to reduce the effects of such undesirable events. Thanks to extra management attention, the successor activities of a disrupted activity will more regularly be on time. We have implemented an *activity crashing* mechanism related to the one used in the experiments of Herroelen and Leus (2002). If an activity is disrupted, we crash up to $\lfloor n/5 \rfloor$ activities chosen from the direct or close transitive successors of the disrupted activity. Crashing means that the disturbance probability is set to 0. During project execution, a maximum of $\lceil n/10 \rceil$ crashing operations may occur. The following computational results are obtained; *PTF* is implemented with $\omega_0=\gamma=0.2$.

Table 10. Comparison of various allocation algorithms, with crashing. Table gives average deviation from *PE*.

	PTF	*AMR*	*worst-case*
n=21	0.3095%	0.7604%	1.1662%
n=31	0.1562%	0.8619%	1.0608%
n=41	0.3129%	0.8024%	0.8553%

Compared with *AMR* and *worst-case*, *PTF* performs better than before; fine-tuning of the parameters in function of n might induce even better performance. The activity duration behaviour could have been implemented also in the evaluation function of *PE*; the algorithm would then have consumed even more time. Moreover, the exact crashing pattern is not really known beforehand. *PTF* offers a simple way to incorporate management actions into the resource allocation logic.

8. Conclusions and suggestions for further research

This paper has proposed a model for resource allocation when uncertainty is present. Contrary to existing literature, we propose to fit the allocation onto a deterministic schedule, and to model activity duration variability in terms of disruptions. Using the model proposed in this paper, management can compute a resource allocation that is best protected against uncertainty and evaluate the benefit

of adding extra resources to a project, even when this may not be necessary from a deterministic scheduling point of view. We have restricted ourselves to the case of a single resource type because this environment is associated with a single resource network, and our analysis was mainly about such networks. The extension to multiple resource types is evident. Preliminary results indicate that robust resource assignment to multipe resource types yields larger benefits than the single resource case when compared with random assignment, because co-ordination across multiple resource types can be exploited.

Ideally scheduling and resource allocation would be performed in parallel. This would also allow to formally consider a trade-off between (initial) schedule length and robustness. We conjecture, given the complexity of the problem, that such approaches could best be initiated in more specific machine scheduling environments. We would also like to make a case for the introduction of the "weighted total float" and related objectives to be studied in the scheduling literature. This paper has illustrated ways in which to interpret such objectives in a stochastic environment.

With regards to constraint propagation, an option that is regularly mentioned in the literature (Dorndorf et al. 2000) is to obtain a certain degree of concistency, but not to pursue completion of the constraint propagation at every step, because this might be too time consuming. Another possibility would be to let the consistency concept at hand vary throughout the search. These ideas require further research. Nevertheless, it has been shown in this paper that constraint propagation offers a valid way to reduce the search space of the robust resource allocation problem, which does not allow for efficient dominance criteria or lower bounds.

When it comes to optimal makespan protection, more than one pre-schedule can be used, and the best allocation can then easily be selected. An alternative to working with one pre-schedule as input would be to discard of the input schedule completely, and construct a general *ES*-policy. This problem can also be cast into a flow network setting: for two precedence unrelated activities i and j, both arc (i,j) and (j,i) should initially be allowed to enter R_f, but one of the options should be eliminated from the moment when the other becomes implicit; this is analogous with orienting a disjunctive arc in the corresponding representation of a job shop scheduling problem. Such a search procedure will be rather intricate to implement, but may offer an alternative to branching based on forbidden sets.

Often, the flow in the partial network simply mimics the residual network, and becomes feasible only when all flow-carrying arcs in the residual network are inserted. It may be interesting to study the branching scheme proposed in this paper, where the fathoming criterion is that all arcs from the residual network be inserted, rather than that a maximal flow test for feasibility be carried out at every level in the search tree. If a non-minimal solution is reported, the binary branching scheme will later on during the search cover the minimal one(s) as well.

References

Adamopoulos, G.I., Pappis, C.P. and Karacapilidis, N.I. (2000). A methodology for solving a range of scheduling problems under uncertainty. *In*: Slowinksi and Hapke.

Adlakha, V.G. and Kulkarni, V.G. (1989). A classified bibliography of research on stochastic PERT networks: 1966-1987. *INFOR*, **27**, 3, 272-296.

Ahuja, R.K., Magnanti, T.L. and Orlin, J.B. (1993). *Network flows. Theory, algorithms, and applications*. Prentice-Hall.

Artigues, C. (2000). Insertion techniques for on and off-line resource constrained project scheduling. *Seventh international workshop on project management and scheduling (PMS 2000)*. April 17-19, 2000. Osnabrück, Germany.

Artigues, C., Michelon, P. and Reusser, S. (2000). Insertion techniques for static and dynamic resource constrained project scheduling. *LIA report 152, Laboratoire d'Informatique d'Avignon.*

Artigues, C. and Roubellat, F. (2000). A polynomial activity insertion algorithm in a multi-resource schedule with cumulative constraints and multiple modes. *European Journal Of Operational Research*, **127**, 2, 297-316.

Bowers, J.A. (1995). Criticality in resource constrained networks. *Journal of the Operational Research Society*, **46**, 80-91.

Brucker, P., Drexl, A., Möhring, R., Neumann, K. and Pesch, E. (1999). Resource-constrained project scheduling: notation, classification, models and methods. *European Journal of Operational Research*, **112**, 3-41.

Bruno, J., Coffman, E.G. Jr. and Sethi, R. (1974). Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, **17**, 7, 382-387.

Chapman, C. and Ward, S. (1997). *Project risk management: processes, techniques and insights*. Wiley.

Davis, E. (1987). Constraint propagation with interval labels. *Artificial Intelligence*, **32**, 281-331.

Demeulemeester, E. and Herroelen, W. (1992). A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management Science*, **38**, 1803-1818.

Demeulemeester, E. and Herroelen, W. (1997). New benchmark results for the resource-constrained project scheduling problem. *Management Science*, **43**, 1485-1492.

Demeulemeester, E., Vanhoucke, M. and Herroelen, W. (2000). A new random network generator for activity-on-the-node networks. *Research report 0032, Department of Applied Economics, KU Leuven.*

Dorndorf, U., Pesch, E. and Phan-Huy, T. (2000). Constraint propagation techniques for the disjunctive scheduling problem. *Artificial Intelligence*, **122**, 189-240.

Edmonds, J. and Karp, R.M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, **19**, 248-264.

Elmaghraby, S.E. (1977). *Activity networks: project planning and control by network models*. Wiley.

Ford, L.R. Jr. and Fulkerson, D.R. (1962). *Flows in networks*. Princeton University Press.

Goldratt, E.M. (1997). *Critical chain*. The North River Press.

Gutierrez, G.J. and Kouvelis, P. (1991). Parkinson's law and its implications for project management. *Management Science*, **37**, 8, 990-1001.

Hagstrom, J.N. (1988). Computational complexity of PERT problems. *Networks*, **18**, 139-147.

Herroelen, W., De Reyck, B. and Demeulemeester, E. (1998). Resource-constrained project scheduling: a survey of recent developments. *Computers and Operations Research*, **25**, 4, 279-302.

Herroelen, W. and Leus, R. (2002). On the merits and pitfalls of critical chain scheduling. To appear in *Journal of Operations Management*.

Igelmund, G. and Radermacher, F.J. (1983). Preselective strategies for the optimization of stochastic project networks under resource constraints. *Networks*, **13**, 1-28.

Kolisch, R. and Padman, R. (2001). An integrated survey of deterministic project scheduling. *Omega*, **49**, 3, 249-272.

Lootsma, F.A. (1997). *Fuzzy logic for planning and decision making*. Kluwer Academic Publishers.

Mackworth, A.K. (1977). Consistency in networks of relations. *Artificial Intelligence*, **8**, 1, 99-118.

Meredith, J.R. and Mantel, S.J. Jr. (2000). *Project management. A managerial approach. Fourth edition*. Wiley & Sons.

Möhring, R.H. (2000). Scheduling under uncertainty: bounding the makespan distribution. *Working Paper 700/2000, Department of Mathematics, TU Berlin*.

Naegler, G. and Schoenherr, S. (1989). *Resource allocation in a network model – the Leinet system*. In: Slowinski and Weglarz.

Nuijten, W.P.M. (1994). *Time and resource constrained scheduling. A constraint satisfaction approach*. Ph.D. Thesis, TU Eindhoven.

Pinedo, M. (1995). *Scheduling. Theory, algorithms, and systems*. Prentice-Hall.

Radermacher, F.J. (1985). Scheduling of project networks. *Annals of Operations Research*, **4**, 227-252.

Rosenhead, J., Elton, M. and Gupta, S.K. (1972). Robustness and optimality as criteria for strategic decisions. *Operations Research Quarterly*, **23**, 4, 413-431.

Roy, B. and Sussman, B. (1964). Les problèmes d'ordonnancement avec contraintes disjonctives. *Note DS n° 9 bis, SEMA, Paris*.

Slowinski, R. and Hapke, M. (2000) (eds.). *Scheduling under fuzziness*. Physica-Verlag.

Slowinski, R. and Weglarz, J. (1989) (eds.). *Advances in project scheduling*. Elsevier.

Stork, F. (2000). Branch-and-bound algorithms for stochastic resource-constrained project scheduling. *Working Paper 702/2000, Department of Mathematics, TU Berlin.*

Stork, F. and Uetz, M. (2000). On the representation of resource constraints in project scheduling. *Working Paper 693/2000, Department of Mathematics, TU Berlin.*

Tsang, E. (1993). *Foundations of constraint satisfaction*. Academic Press.

Wiest, J.D. (1964). Some properties of schedules for large projects with limited resources. *Operations Research*, **12** (May-June), 395-418.

Wiest, J.D. and Levy, F.K. (1977). *A management guide to PERT/CPM. 2nd edition.* Prentice-Hall.

Appendix A

Suppose we re-order N such that $i<j \Rightarrow s_i \leq s_j$, hence any $(i,j) \in TA \cup R(C)$ has $i<j$. At initialization of the search, define set of implicit arcs β_0 as follows.

```
for i:=n-1 downto 0 do
        add every (i,j)∈TA∪α0 to β0
        ∀(i,j),(j,k)∈β0, add (i,k) to β0
```

β_p, $p>0$, is initialized as β_{p-1}. Every time an arc (i,j) is set required because of a branching decision, constraint propagation or a dominance rule, we update as follows:

```
∀(k,i)∈βp, add (k,j) to βp
∀(j,l)∈βp, add (i,l) to βp
∀(k,i),(j,l)∈βp, add (k,l) to βp
```

Appendix B

A concise pseudo-code version of the branch-and-bound algorithm follows. Functions `implement_branching_arc` and `make_consistent` update upper and lower bounds on individual flows. `make_consistent` will regularly call upon `add_arc_flow` and `remove_arc_flow` in order to adapt flow in both the residual and the partial network to be compatible with all most recent bound values. On backtracking, we keep the flow in the extended partial network feasible (remove flow from arcs that became required by branching) and maximal in the extended residual network. `inconsistency` is a boolean that is triggered whenever an infeasibility is detected; if `(inconsistency)`, we will always terminate the current branch of the search tree and backtrack. Similar implications hold when `dominated_solution` is true. Values TF_i for problem *PTF* and earliest start values for *PE* are updated continuously, whenever an arc becomes required. Each time these values are updated, we check whether we can prune the current node of the search tree by bound arguments on the best objective function value obtainable from the node and if so, `dominated_solution` is triggered. A similar check is performed before entering any right branch, which is useful in the cases where a new incumbent was found at the left branch.

```
branch_and_bound_allocation(schedule C)
        level=0;
        initialize_bounds;
        make_consistent;
        init_flow(residual); maxflow(residual);
        if (residual_flow<mu_max) return problem_infeasible;
        init_flow(partial); maxflow(partial);
        if (partial_flow==mu_max) goto FEASIBLE_SOLUTION;
BRANCHING_CHOICE:
        choose_branching_arc;
        level++;
        branching_decision(level)=LEFT;
        status_required(branching_arc);
```

```
      add_arc_flow(branching_arc,residual);
      if (inconsistency OR dominated_solution) goto BACKTRACK;
      add_arc_flow(branching_arc,partial);
      if (partial_flow==mu_max) goto FEASIBLE_SOLUTION;
      make_consistent; if (dominated_solution) goto BACKTRACK;
      maxflow(partial); if (partial_flow<mu_max) goto BRANCHING_CHOICE;
FEASIBLE_SOLUTION:
      evaluate_and_store;
      if (level==0) return optimal_solution_found;
BACKTRACK:
      remove_arc_flow(branching_arc,partial);
      undo_decisions(level);
      level--;
      if (branching_decision(level)==LEFT) goto RIGHTBRANCH;
      else
            if (level==0) return optimal_solution_found;
            goto BACKTRACK;
RIGHTBRANCH:
      level++;
      branching_decision(level)=RIGHT;
      status_forbidden(branching_arc);
      remove_arc_flow(branching_arc,residual);
      if (inconsistency OR dominated_solution) goto BACKTRACK;
      make_consistent; if (dominated_solution) goto BACKTRACK;
      maxflow(partial);
      if (partial_flow<mu_max) goto BRANCHING_CHOICE;
      else goto FEASIBLE_SOLUTION;
```

Appendix C

Artigues et al. (2000) present a simple method to obtain a feasible resource flow by extending a parallel schedule generation scheme to derive the flows during scheduling. Uncoupled from the schedule generation, this algorithm looks as follows. The algorithm is an implementation of the flow-rerouting described in the proof of Theorem 3. f_{0n} is initialized with value a, all other flows are set to 0. Condition (*) is not mentioned in the reference but seems logical.

```
AMR(schedule C)
      for i:=1 to |Λ_C| do
            for j:=1 to (n-1) do
                  if (s_j==t_i)
                        req:=r_j; k:=0;
                        while (req>0) do
                              if e_k(C) ≤ s_j(C)   (*)
                                    m:=min{req, f_kn}; req-=m;
                                    f_kn-=m; f_kj+=m; f_jn+=m;
                               k++;
```

A practical worst case allocation for a given input schedule would load the resource units as heterogeneously as possible. We will use the following algorithm as a reference. We identify every unit of the single resource type separately. All flows f_{ij} are initialized to 0.

```
worst_case(schedule C)
        for i:=1 to a do
                end(i):=0; assignm(i):=0;
        for i:=1 to n do
                req:=u_i; k:=1;
                while (req>0) do
                        if end(k) ≤ s_i
                                req--; end(k):=e_i;
                                f_assignm(k),i++; assignm(k):=i;
                        k++;
```