KATHOLIEKE
UNIVERSITEIT
LEUVEN

DEPARTEMENT TOEGEPASTE ECONOMISCHE WETENSCHAPPEN

RESEARCH REPORT 0256

STABILITY AND RESOURCE ALLOCATION IN PROJECT PLANNING

by
R. LEUS
W. HERROELEN

D/2002/2376/56

Stability and Resource Allocation in Project Planning

Roel Leus • Willy Herroelen

Operations Management Group, Department of Applied Economics, Katholieke Universiteit Leuven, B-3000 Leuven, Belgium

Roel.Leus@econ.kuleuven.ac.be • Willy.Herroelen@econ.kuleuven.ac.be

The majority of resource-constrained project scheduling efforts assumes perfect information about the scheduling problem to be solved and a static deterministic environment within which the pre-computed baseline schedule is executed. In reality, 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 a given baseline schedule against activity duration variability. A branch-and-bound algorithm is developed that solves the proposed resource allocation problem. We report on computational results obtained on a set of benchmark problems.

(*Project management; Project planning; Scheduling; Resource Allocation; Constraint Satisfaction*)

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 focuses on exact and sub-optimal procedures for constructing a workable schedule assuming perfect 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 *pre-schedule* or *baseline schedule*, serves as the baseline for executing the project.

During project execution, 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, workers may be absent, due dates may be modified because of changed customer demands, etc. The recognition that uncertainty lies at the very heart of project planning induced a

number of research efforts based on stochastic analysis, in the absence of resource constraints (the PERT problem) (Adlakha and Kulkarni, 1989; Elmaghraby, 1977). When resource constraints are introduced, 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, 1983a; Stork, 2001).

Mehta and Uzsoy (1998) state that a predictive schedule or pre-schedule serves two important functions. The first is to allocate resources to the different jobs to optimise some measure of (shop) performance. The second, as also pointed out by Wu et al. (1993), is to serve as a basis for planning external activities such as material procurement, preventive maintenance and committing to shipping dates to customers. Especially in multi-project environments, a schedule often needs to be sought *before the start of the project* that is in accord with all parties involved (clients and suppliers, as well as workers and other resources). It may be necessary to agree on a time window for work by sub-contractors; a deterministic schedule is also vital for cash flow projections and for performance appraisal subsequent to project completion. Further discussion of the purposes of a (pre-)schedule can be found in Aytug et al. (2002); they state that, if the level of uncertainty is low enough, an optimisation-based pre-schedule can outperform an on-line dispatching algorithm (but the converse is true once uncertainty exceeds a certain threshold).

Since the schedule is the basis for project management, *stability* of the plan is indispensable. For more details on stability in scheduling, we refer to Herroelen and Leus (2002). It is also crucial, mainly in multi-project environments, to make advance bookings of key staff or equipment to guarantee their availability (Bowers, 1995), based on the pre-schedule, thus making last-minute changes in resource allocation unachievable (contrary to the case of totally dedicated resources). In view of achieving stability, algorithms have been proposed that use a *match-up point*, described by Akturk and Gorgulu (1999) as the time instance "where the state reached by the revised schedule is the same as the initial schedule," when action is undertaken after a machine breakdown. They continue, "the pre-schedule can be followed if no disruption occurs." Robust scheduling on the other hand builds protection into the pre-schedule, so is proactive rather than reactive. This paper studies resource allocation to optimally protect the schedule against activity duration variability, which is an intermediate solution between proactive and reactive scheduling. We consider the case of a single resource type, which can be chosen as the most restrictive or bottleneck resource of the organisation.

The structure of the paper is as follows. Resource allocation solutions correspond with resource flow networks, which are discussed in Section 2. Section 3 describes a branch-and-bound procedure that generates a robust resource allocation. The procedure exploits constraint propagation techniques and an efficient procedure for testing for the existence of a feasible flow. Computational results obtained by the algorithm on a set of test problems are provided in Section 4. Section 5 provides overall conclusions and offers some suggestions for future research.

2 Resource allocation and resource flow networks

Section 2.1 presents some mathematical notation used throughout this paper. Resource flow networks are discussed in Section 2.2. The link with activity duration uncertainty is the subject of Section 2.3.

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 fixed baseline duration $d_i \in \mathbb{N}$ and requires $r_i \in \mathbb{N}$ units of the single renewable resource type, all $r_i \leq a$. Apart from the dummy start activity 0 and dummy end n, activities have non-zero duration; the dummies also have zero resource usage. A is the set of pairs of activities between which a finish-start precedence relationship with time lag 0 exists. We assume graph $G(N,A)$ to be acyclic and equal to its transitive reduction (no redundant arcs are included). Without loss of generality, we also require $\forall (i,j) \in A: i<j$. For any $X \subseteq N \times N$, we can obtain the immediate predecessors of activity i by function π_X: $N \rightarrow 2^N$: $i \rightarrow \pi_X(i) = \{j \in N \mid (j,i) \in X\}$, and its immediate successor activities via σ_X: $N \rightarrow 2^N$: $i \rightarrow \sigma_X(i) = \{j \in N \mid (i,j) \in X\}$, and we define TX as the transitive closure of X, meaning that $(i,j) \in TX$ if a path from i to j exists in $G(N,X)$. 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.

A schedule S is defined by an $(n+1)$-vector of start times $\mathbf{s}(s_0,\ldots,s_n)$; every $\mathbf{s}$ implies an $(n+1)$-vector of finish times $\mathbf{e}$, $e_i = s_i + d_i$, $\forall i \in N$. With every schedule S, we associate a set $\delta(S)$ of time instances or 'decision points', which correspond with its activity start and finish

times: $t \in \delta(S)$ if $\exists i \in N$: $t=s_i$ or $t=e_i$. Define $N_t:=\{i \in N \mid s_i<t \leq e_i\}$, the activities that are active during period t. Schedule S is feasible if

$$(1)\ \forall (i,j) \in A: e_i(S) \leq s_j(S), \quad \text{and} \quad (2)\ \forall t \in \delta(S): r(N_t) \leq a. \tag{2.1}$$

An RCPSP-instance $\Gamma(N,A,a,\mathbf{d},\mathbf{r})$ aims to find a feasible schedule that minimizes e_n (in this case for a single resource type).

2.2 Resource flow networks

Artigues and Roubellat (2000) present a *resource flow network*, in which the amount of resources being transferred immediately from one activity to another is explicitly recorded; they use this network to insert new activities into the project with constant resource allocation. Bowers (1995) defines 'resource-constrained float' as the CPM total float based on the technological precedences combined with the flow network. Naegler and Schoenherr (1989) solve deterministic time/resource and time/cost trade-off problems 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. Schwindt (2001) and Neumann et al. (2002) use the network representation to test whether a schedule is feasible, in the context of sequence-dependent changeover times. They refer to a model for aircraft scheduling presented in Lawler (1976).

Define $u_i=r_i$, $\forall i \in N\setminus\{0,n\}$, and $u_0=u_n=a$. A resource flow f associates with each pair $(i,j) \in N \times N$ a value $f_{ij}:=f(i,j) \in \mathbb{N}$. These values must satisfy the flow conservation constraints:

$$f(i,N) = u_i \qquad \forall i \in N\setminus\{n\} \tag{2.2}$$

$$f(N,i) = u_i \qquad \forall i \in N\setminus\{0\} \tag{2.3}$$

f_{ij} represents the (discrete) number of resources that are transferred from activity i (when it finishes) to activity j (when it starts). For a flow f, define the set of activity pairs $E_f=\{(i,j) \in N \times N \mid f_{ij}>0\}$, containing the arcs that carry flow in the resource flow network. We also define $R_f=E_f\setminus TA$: the arcs in R_f are the flow-carrying arcs that do not represent direct nor transitive precedence relations. We call flow f feasible when condition (2.4) holds: extra precedence constraints implied by R_f do not prevent execution of the project if f is feasible.

$$G_f(N, TA \cup R_f) \text{ acyclic} \tag{2.4}$$

A small example is in order at this point. In Figure 1, an example network is represented in activity-on-the-node format, we assume $a=3$. According to our definitions, $u_0=u_5=3$, $u_1=u_2=1$ and $u_3=u_4=2$. One possible resource flow sets $f_{02}=f_{15}=f_{23}=f_{41}=f_{43}=1$ and $f_{04}=f_{35}=2$, all other flows to 0; this flow is illustrated in Figure 2(a). We see for instance that one of the available resource units is transferred from the end of dummy activity 0 to the

start of activity 2. This unit is released at the completion of activity 2 and transferred to the start of activity 3. The resource flow network shown in Figure 2(b) represents an alternative resource allocation. In Figure 2(a), Rf={(4,1),(4,3)} while Rf={(1,3),(4,1)} in the resource flow network of Figure 2(b); arcs in Rf are dashed.

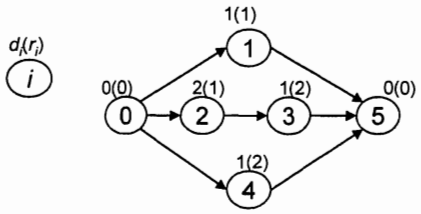

Figure 1. Example project network.

Define $\theta(X)$, $X \subseteq N \times N$, to be the schedule in which each activity i starts at time $s_i = \max_{j \in \pi_{A \cup X}(i)} \{s_j + d_j\}$, provided graph $G(N, A \cup X)$ is acyclic: the arcs in X represent extra precedence constraints, in addition to A. A solution to an RCPSP-instance can be obtained by finding a feasible flow f that minimizes $s_n(\theta(A \cup Rf))$ (evidence for this follows from the material presented in Section 2.3), and we see that we obtain an extension of the disjunctive graph representation of the classical job shop scheduling problem (Roy and Sussman, 1964).

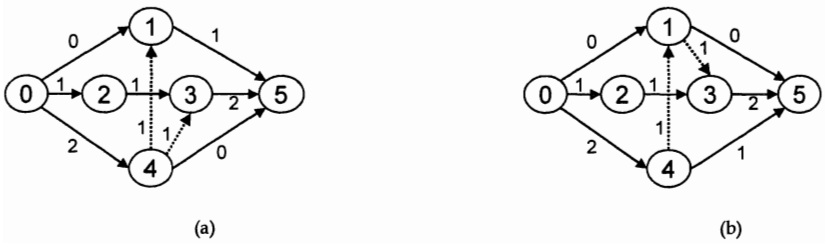

Figure 2. Two resource flow networks. Flow values are indicated on the arcs.

An important point to make is that, contrary to the job shop problem, we often have more than one possible resource allocation corresponding with a single schedule, an observation that is the starting point of this paper. Both resource flows in Figure 2, for instance, result in the same schedule depicted in Figure 3.

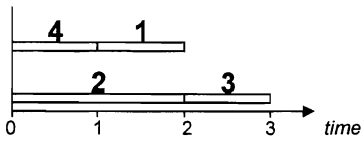

Figure 3. A schedule for the example project.

2.3 Activity disruptions and stability

We assume that all uncertainty during project execution can be represented by variability in task durations. The sources of this uncertainty are manifold, as discussed in the introduction. The stochastic variable representing the duration of activity $i \in N$ is denoted by D_i, these variables are collected in vector **D**. During the schedule repair process, the resource allocation remains constant, i.e. the same resource flow is maintained. This reactive scheduling policy is preferred when specialist resources (e.g. expert staff) cannot be easily transferred between activities at short notice, for instance in a multi-project environment, where it is necessary to book key staff or equipment in advance. Artigues and Roubellat (2000) also refer to the desire to ensure schedule stability (avoiding system nervousness resulting from complete rescheduling), and limited computation time, especially in case of on-line scheduling.

Igelmund and Radermacher (1983a) present different scheduling policies for stochastic project networks under resource constraints, all based on the concept of forbidden sets, which are sets of precedence unrelated activities that are not allowed to be scheduled simultaneously because of resource constraints. A set of policies of interest to us is the set of Earliest Start policies (*ES*-policies). The idea is to extend the given partially ordered set $G(N,A)$ to a partially ordered set $G(N,A \cup X)$, such that no forbidden set remains precedence unrelated and could thereby be scheduled in parallel. The condition for feasibility of the policy is that $G(N,A \cup X)$ still be acyclic. Then, in order to obtain a feasible schedule $\mathbf{S}(\mathbf{d})$ for a given scenario **d** of activity durations, an *ES*-policy simply computes earliest activity start times in the graph by performing a forward CPM (longest path) pass (Stork, 2001). We let G_X represent graph $G(N,TA \cup X)$. The following theorem is intuitive:

Theorem 1. *For any feasible resource flow f, $X=R^f$ defines a feasible ES-policy. Conversely, if X defines a feasible ES-policy, a feasible flow f exists with $TA \cup R^f \subseteq T(A \cup X)$.*

The proofs of the theorems are relegated to Appendix A; the proof of Theorem 1 draws in part from Möhring (1985). Theorem 1 ensures that a complete search of feasible flows does not overlook any *ES*-policy. For an illustration, we refer again to the resource flow networks in Figure 2. Suppose that project management is uncertain about the duration of activity 4. It is obvious that in this case, the flow pattern in Figure 2(a) is more robust with regard to expected makespan than the pattern in Figure 2(b): with respect to a projected makespan of 3 time units, activity 4 has a total slack (cfr. Wiest and Levy, 1977) of 1 in the first resource allocation, and 0 in the second. As a result, an increase in d_4 will have an immediate impact on the makespan of the repaired schedule in 2(b), while a buffer of size 1 is provided in 2(a).

Given the multitude of constraints involved in practical schedule development, we perform scheduling and resource allocation sequentially. We impose the constraint that resource allocation be compatible with a pre-determined pre-schedule S: this compatibility guarantees that the pre-schedule will be realized if everything goes as planned. Define $R(S)=\{(i,j)\in N\times N \mid (i,j)\notin TA \wedge e_i(S) \leq s_j(S)\}$. A feasible flow f is said to be *compatible* with a feasible schedule S, written $f\sim S$, if $\forall(i,j)\in TA\cup Rf$: $e_i(S) \leq s_j(S)$, or in other words if $Rf \subseteq R(S)$. As mentioned before, a pre-schedule S serves the purpose of co-ordinating with external parties: whatever was the basis for the development of this baseline schedule *before* project execution, the baseline will serve as a guideline *during* execution for all persons engaged.

In situations where a pre-schedule is valuable (hinted at in the introduction, mainly when the resources are not entirely *dedicated* to the project), it will be of interest that activities are not started as soon as feasible but rather that it is attempted to respect the pre-schedule to the best extent possible, in order to avoid system nervousness and constant resource rescheduling, in other words, to maintain *stability* in the system. As a result, activities are started at the maximum of the ending times of the predecessors *and* their pre-schedule starting time. Other scheduling disciplines that operate in this way are railway and airline scheduling. The *actual* starting time of activity i is a stochastic variable $S_i(Rf,S)=\max\{s_i(S); \max_{j\in\pi_{TA\cup Rf}(i)}\{S_j(Rf,S)+D_j\}\}$, with $s_0(S)=0$. Following Herroelen and Leus (2002), we adopt as measure of pre-schedule stability the *expected weighted deviation in start times* in the *actual* schedule from those in the pre-schedule. Our aim is to construct a feasible flow f with $Rf\subseteq R(S)$ such that $E[\sum_{i\in N} c_i[S_i(R^f,S)-s_i(S)]]\equiv g(Rf)$ is minimized, where $E[\cdot]$ is the expectation operator and schedule S is an input parameter. $c_i\in\mathbb{N}$ denotes the non-negative cost per unit time overrun on the start time of activity i, which reflects either the difficulty in obtaining the required resources (internal stability) or the importance of on-time performance of the activity to the customer (external stability). We always set $c_0=0$;

minimisation of expected makespan is the special case $c_i=0$, $i\neq n$, and $c_n\neq 0$. It is clear from the definition of $R(S)$ that for any feasible solution f, $f\sim S$.

The following theorem shows that for convenient input data, the allocation problem will not have an empty solution space.

Theorem 2. *For every feasible schedule S there exists at least one feasible flow f such that $f\sim S$.*

3 A branch-and-bound procedure

This section gives an overview of our resource allocation algorithm. Section 3.1 reformulates the problem in the context of constraint satisfaction and presents the basic branching scheme. Section 3.2 provides more details on the search strategy used. Section 3.3 explains how we test for the existence of a feasible flow in the network we have constructed. Constraint propagation techniques speed up our algorithm and are discussed in Section 3.4. The evaluation of the objective function is the subject of Section 3.5. We compare our algorithm with a forbidden set branching scheme and discuss our branching rule in Sections 3.6 and 3.7, respectively.

3.1 Constraint satisfaction problem and branching

A *constraint satisfaction problem* (csp) is defined by a triple (F,B,C) where F is a finite set of variables, B is a function which maps every variable k in F to a set of possible values B_k, called its domain, and C 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 optimisation problem* (csop) is defined as a csp together with an optimisation function g that maps every solution tuple to a numerical value; the csop aims to identify a value assignment with optimal objective function. In this paper, the set of decision variables is the set of flows $F=\{f_{ij} \mid (i,j)\in TA\cup R(S)\}$. For $f_{ij}\in F$, $B_{ij}=[0;+\infty]$ is the domain initially associated with f_{ij}, and Z contains constraint sets (2.2) and (2.3) and the requirement that $f\sim S$, which is implicit from F. Eq. (2.4) is also satisfied because arc $(i,j)\in TA\cup R$ has $e_i(S)\leq s_j(S)\leq e_j(S)$, since input schedule S is feasible.

For $f_{ij}\in F$, B_{ij} can be represented by its lowest entry LB_{ij} and highest entry UB_{ij}: we represent the domains as intervals. 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 detection and removal of *inconsistent* variable assignments, which cannot participate in any solution (Dorndorf et al., 2000). Our branch-and-bound procedure relies on constraint propagation for search space reduction. Remark that, by its very definition, we do not lose any solution tuple by the application of constraint propagation. We restrict ourselves to administration of current domains, and do 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).

We find an optimal resource allocation for a schedule S by considering all subsets $M \subseteq R(S)$ that allow a feasible flow in network $TA \cup M$; one such set corresponds with at least one and mostly multiple feasible f, with $R^f \subseteq M$. We iteratively add arcs from $R(S)$ to M until a feasible flow is attainable (the feasibility test is the subject of Section 3.3). The following observation enables us to restrict our attention to subset minimal M:

Observation 1. *For two feasible flows f_1 and f_2, if $R^{f_1} \subset R^{f_2}$, then $g(R^{f_1}) \leq g(R^{f_2})$.*

A similar remark appears in Stork (2001) (Lemma 5.3.2). Observation 2 enables us to restrict the search to the integer numbers contained in the interval domains of the flows without loss of better solutions (and which is in line with the interpretation we gave to the flow values in Section 2).

Observation 2. *For any feasible flow f_1, we can always find a feasible integer flow f_2 such that $R^{f_1} \subseteq R^{f_2}$.*

Observation 2 follows because f_1 is a maximal flow in the network G^{f_1}, and all capacities and lower bounds are integer. Thus, an integer maximal flow f_2 in the same network exists. f_2 may or may not use all arcs in $TA \cup R^{f_1}$, hence $R^{f_1} \subseteq R^{f_2}$.

At any level p of our search tree, set $TA \cup R(S)$ is partitioned into three disjoint subsets: $TA \cup R(S) = \alpha_p \cup \nu_p \cup \omega_p$, with $\alpha_p = \{(i,j): LB_{ij} > 0\}$ the set of included arcs, $\nu_p = \{(i,j): UB_{ij} = 0\}$ the set of forbidden arcs, and $\omega_p = \{(i,j): LB_{ij} = 0 \text{ and } UB_{ij} > 0\}$ the set of undecided arcs. Bounds LB_{ij} and UB_{ij} are established through constraint propagation (to be discussed in Section 3.4), in conjunction with branching decisions. We add all arcs in $\alpha_p \setminus TA$ to M_p, which results in *partial network* $G_p \equiv G_{M_p}$. If a feasible flow can be obtained in G_p, we fathom the current node

and backtrack, otherwise we need further branching decisions. The branching decision itself entails the selection of an undecided arc $(i,j) \in R(S) \cap \omega_p$: the left branch is to set $LB_{ij}:=1$, so to include (i,j) in the partial network G_p; the right branch is to impose $UB_{ij}:=0$, so to forbid any flow across (i,j) and prohibit inclusion of (i,j) in M by placing the arc into set ν_p. We elaborate on the selection of the branching arc in Section 3.7. Note that such binary branching suffices for our purposes: either an arc is in R, or it is not. The *amount* of flow across an arc is not important, only the question whether the flow is zero or nonzero, given the form of $g(R)$. In effect, by 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.

3.2 Details of the branch-and-bound algorithm

By Jensen's inequality, the deterministic value obtained when activity durations are set equal to their expected values is a lower bound for our objective function (cfr. Fulkerson, 1962, for an application to expected makespan bounding). In order to obtain a lower bound at every node of the search tree, we maintain a set of earliest starting times in G_p based on expected activity durations; these earliest starting times are continuously updated. We refer to this bound as the *critical path lower bound.*

Combinations of the precedence relations defined by $TA \cup M_p$ imply extra transitive relations, captured by $T(A \cup M_p)$. These implicit precedences are incurred anyway, so we can extend set $M_p:=T(A \cup M_p) \setminus \nu_p$ without deteriorating the objective. In our implementation, we continuously update this set rather than reconstruct it from scratch each time it is needed. In a forbidden set branching scheme, the same insight is reflected by the use of a *destruction matrix* (Radermacher, 1985) or alternatives with less memory usage (Stork, 2001) (cfr. Section 3.6).

Stork (2001) presents a single machine relaxation bound for stochastic project scheduling. This bound considers sets of precedence unrelated activities that are pair-wise incompatible because of resource constraints, and computes a lower bound on expected project makespan as the smallest expected head plus smallest expected tail added to the sum of the expected durations of the activities. In our problem, the sequencing problem for such sets of activities has already been completely solved, and either directly or transitively, a precedence constraint $i \rightarrow j$ will be included for all pairs of incompatible activities (i,j) with $e_i(S) \leq s_j(S)$. We can include all those pairs into M_0 from the outset (in our implementation, we add them to A). We refer to this extra measure as the *single machine rule.* For the

example project in Figure 1, we see that activities 3 and 4 jointly consume more than the available 3 resource units. The schedule in Figure 3, referred to as S^*, solves this conflict by positioning activity 4 before activity 3. We can therefore add element (4,3) to M_0 or A.

When too many arcs have been forbidden, the partial network can no longer be completed to generate a feasible flow. Fast detection of these situations allows termination of exploration of the current branch of the search tree. For this purpose, we resort to a second network: the *remainder network* $G_p^r = G_{R(S)\setminus\nu_r}$. As long as G_p^r allows a feasible flow, respecting the branching decisions higher in the search tree, it is possible to select a set $R' \subseteq R(S)\setminus\nu_p$ that allows a feasible flow in G_p and corresponds with all branching decisions. Otherwise, we prune the branch and backtrack. From Theorem 2, G_0^r always allows a feasible flow. When G_p^r verifies the existence of at least one feasible solution down the search tree, we apply constraint propagation to further tighten the domains of the decision variables, to avoid branching into infeasible areas as well as making branching decisions that are already implicit. A discussion of this propagation is the subject of Section 3.4.

3.3 Testing for the existence of a feasible flow

In this section, we discuss a simple way to test for the existence of a feasible flow in a given network, using maximal flow computations in a transformed network. Möhring (1985) studies a related transformation that, in our terminology, allows to determine the minimal required value of a (cfr. the proof of Theorem 1). Naegler and Schoenherr (1989), Schwindt (2001) and Neumann et al. (2002) discuss similar transformations.

For network G_M, $M \subseteq R(S)$, we construct a new network G'_M as follows. We switch from bounds on node flow to bounds on arc flow by duplicating each node $i \in N\setminus\{0,n\}$ into two nodes i_s and i_t and adding arc (i_t,i_s) to the network, with upper bound on flow (i_t,i_s) equal to its lower bound, both equal to u_i (cfr. Ford and Fulkerson, 1962); nodes 0 and n are renamed 0_s and n_t, respectively. All arcs entering i in G_M now lead to i_t, all arcs leaving i now emanate from i_s. Figure 4(a) shows the transformed network G'_0 of G_0 for the project shown in Figure 1. We choose $S=S^*$ and take $M_0=\{(4,3)\}$, as suggested in Section 3.2. Node i_t can be interpreted as the start of activity i (reception of resources), node i_s as its completion (passing on the resources). We augment network G'_M with source node s, sink node t and arc (t,s). Every arc (i_t,i_s) in G'_M is replaced by arcs (i_t,t) and (s,i_s); the resulting network is referred to as G''_X. 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 $+\infty$. All flow lower bounds are set to 0. Figure 4(b) shows the network G''_0 obtained from the network G'_0 of Figure 4(a).

Denote by $\mu(M)$ the maximal s-t flow value in G''_M, and let h be a corresponding maximal flow. It is clear that h satisfies the following two conditions:

$$h(i_s,\{j_t \mid j\in N\}) \le u_i \qquad \forall i\in N\setminus\{n\} \qquad (3.1)$$

$$h(\{j_s \mid j\in N\},i_t) \le u_i \qquad \forall i\in N\setminus\{0\} \qquad (3.2)$$

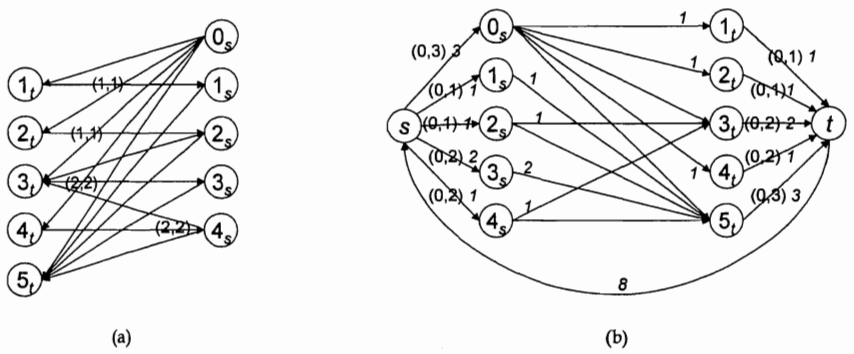

Figure 4. G'_0 and G''_0 for the example problem. Lower and upper bounds on arc flow are as indicated between brackets, otherwise $(0,+\infty)$. Flow values in (b) are indicated in italic, otherwise 0.

If we define $\mu_{max}:= a+r(N)$, we see that $\mu(M)\le \mu_{max}$, and equality $\mu(M)=\mu_{max}$ holds if and only if a maximal s-t flow in G''_M saturates all source and sink arcs, so that conditions (3.1) and (3.2) are satisfied as equality. The following lemma holds:

Lemma 1. *For $M\subseteq R(S)$, a feasible flow f exists in G_M with $R\subseteq M$ if and only if $\mu(M)=\mu_{max}$.*

The maximal flow in network G''_0 of Figure 4(b) amounts to $8<\mu_{max}=9$ (arc flows are in the figure), so we conclude that a feasible flow is not attainable in G_0. We apply Lemma 1 during the course of the branch-and-bound algorithm to test for the existence of a feasible flow in both the partial network G_p and the remainder network G'_p. For this purpose, we use the extended networks G''_p and G''^{r}_p. We impose LB_{ij} and UB_{ij} as lower and upper bounds on $f(i_s,j_t)$ in either network, if the arc is present, rather then $(0,+\infty)$: these bounds have been tightened based on the branching decisions and constraint propagation and hold for any feasible flow.

At level 0 of the search tree, $f(i_s,j_t)$ in both networks is initialised at LB_{ij} for every (i,j) present, 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). This algorithm is strongly polynomial and is

implemented in a breadth first approach such that the labels are immediately available for use in the branching rule (cfr. Section 3.7).

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. This supports the use of the shortest augmenting path algorithm, which can be implemented with very little overhead. When 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 $\max\{0\ ;\ f_{ij} - UB_{ij} + \Delta\}$. When a lower bound LB_{ij} is tightened by Δ, we try to reconstitute a feasible flow by looking for a series of augmenting cycles that contain (i_s,j_t) as a forward arc and jointly set $f_{ij} \geq LB_{ij} + \Delta$. As an example, we explain how the addition of arc (4,1) in G_0 because of extra constraint $LB_{41}=1$ as branching decision at level 1 of the search tree is handled: we add arc (4,1) in G_0'' and find augmenting cycle $s\text{-}4_s\text{-}1_t\text{-}0_s\text{-}4_t\text{-}t\text{-}s$, such that $\mu(M_1)= 9$, which indicates a feasible flow.

3.4 Constraint propagation

Define a csp to be consistent for a constraint c if $\forall k\in F$, $\forall q_k\in B_k$: $\exists(\tilde{q}_1,\ldots,\tilde{q}_{k-1},\tilde{q}_{k+1},\ldots,\tilde{q}_{|F|})\in(\times_{l\in F,l\neq k}B_l)$: c holds for solution ($k=q_k$ and $\forall l\in F\setminus\{k\}$: $l=\tilde{q}_l$). We propagate constraints to achieve desired consistency in a manner comparable with Davis (1987), which is related to algorithm AC-3 to obtain arc-consistency in binary constraint satisfaction problems (Mackworth, 1977). In this section, we cover the topics of flow and schedule consistency, and provide some details of the manner in which constraint propagation is implemented in our algorithm. Since constraint propagation is performed after updating G_p', infeasibilities will be discovered beforehand, and constraint propagation is used *only* for (flow) bound tightening (unlike it classical use), leading to a smaller search tree and aiding in strengthening the objective bound.

3.4.1 *Flow consistency*

We define our csp to be *outflow-consistent* if it is consistent for Eqs. (2.2). *Inflow-consistency* is achieved by consistency for Eqs. (2.3). We tighten the upper and lower arc flow bounds as follows. Consider constraints (2.2) for a particular i-value. We can achieve consistency by tightening our bounds in the following way (cfr. also Brearley et al., 1975):

$$LB_{ij}:=\max\{LB_{ij}\ ;\ u_i-\sum_{(i,k)\in TA\cup R(S)}UB_{ik}\} \qquad \forall j\in N\setminus\{0\} \qquad (3.3)$$

$$UB_{ij}:=\min\{UB_{ij}\ ;\ u_i-\sum_{(i,k)\in TA\cup R(S)}LB_{ik}\} \qquad \forall j\in N\setminus\{0\} \qquad (3.4)$$

Consistency is achieved if we iterate (3.3) and (3.4) as long as any of the updates changes the argument of any other. It is clear that no feasible values are deleted from the domains and that after tightening the bounds, the csp is consistent for the constraint under consideration. For inflow consistency, we obtain similar equations. For the example project, again with $S=S^*$, we can immediately set $LB_{02}=1$ and $LB_{04}=2$ from inflow-consistency in activities 2 and 4, respectively. This in turn gives $UB_{01}=UB_{03}=UB_{05}=0$ from outflow-consistency in activity 0. Outflow-consistency in activity 3 sets $LB_{35}=2$, and it then follows from inflow-consistency in activity 5 that $UB_{45}=1$ (which was 2, originally). We also notice that $LB_{41}=1$ because activity 1 cannot obtain its resource unit elsewhere, and this leads to $UB_{43}=1$. Given these new bounds, we extend α_0 with (4,1) (and other arcs), and beget $\mu(M_0)=9$ (cfr. the example in Section 3.3), such that G_0 contains a feasible flow and we do not need to branch at all. Such a flow is depicted in Figure 2(a). Since, by the single machine rule, activity 4 will either directly or transitively pass on flow to activity 3, there is no need for using (1,3), and the flow in Figure 2(b) is dominated; notice, however, that $LB_{43}=0$, because flow from 4 to 3 across activity 1 is dominated, but not impossible.

3.4.2 *Schedule consistency*

Define $I_t:=\{(i,j)\in TA\cup R(S) \mid j\in N_t\}$, all arcs entering N_t-jobs, and $P_t:=\{(i,j)\in TA\cup R(S) \mid e_i<t\leq s_j\}$, the arcs that are 'in parallel' with N_t. As an example, for the schedule of Figure 3, $N_3=\{3\}$, $I_3=(\{0,1,2,4\},3)$ and $P_3=(\{0,1,2,4\},5)$.

Lemma 2. *$I_t\cup P_t$ is a network cut in the graph $G_{R(S)}$, $\forall t\in \delta(S)\setminus\{0\}$.*

Corollary. *If feasible flow f is compatible with feasible schedule S, it holds that*

$$r(N_t) + f(P_t) = a \qquad \forall t\in \delta(S)\setminus\{0\} \qquad (3.5)$$

Figure 5 shows the graph $G_{R(S)}$ and the three cuts defined by $I_1\cup P_1$, $I_2\cup P_2$, $I_3\cup P_3$, for the example schedule in Figure 3. As can be verified, the flow across each of these cuts equals 3 for both resource flows depicted in Figure 2.

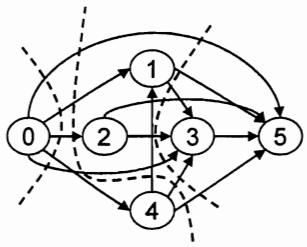

Figure 5. Network cuts corresponding with the example schedule (dashed lines).

We define our csp to be *schedule-consistent* at period t° if it is consistent for Eq. (3.5) with $t=t^\circ$. It is clear that schedule-consistency for period 1 corresponds with outflow-consistency for activity 0. In a similar way as before, we obtain the following consistency-updates:

$$LB_{ij}:=\max\{LB_{ij}\,;\, a-r(N_t)-\sum_{(k,l)\in P_t,\neq(i,j)} UB_{kl}\} \qquad \forall(i,j)\in P_t \tag{3.6}$$

$$UB_{ij}:=\min\{UB_{ij}\,;\, a-r(N_t)-\sum_{(k,l)\in P_t,\neq(i,j)} LB_{kl}\} \qquad \forall(i,j)\in P_t \tag{3.7}$$

We see that $r(N_t)=a$ permits to eliminate all arc flows in P_t from the outset.

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

Artigues (2000) remarks that we can initialise UB_{ij} as $\min\{u_i,u_j\}$. At level 0 of the search tree of the branch-and-bound algorithm, we tighten the domains of F by making them flow and schedule consistent. If we branch on f_{ij} at level p, the left branch is to impose $LB_{ij}:=1$, the right branch is to set $UB_{ij}:=0$. We propagate this bound and make the domains consistent again. As a result, after constraint propagation at level p, the set of domains is consistent for the set of constraints consisting of the branching decisions up until p combined with Eqs. (2.2), (2.3) and (3.5).

We use a queuing structure, where a constraint is added to the queue when one of its arguments is changed, and removed when propagated. At level 0, the update queue is initialised to include all available updates. When executed, an 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. Leus and Herroelen (2001) show that an update of a bound $b(x)$ of variable x because of an update of $b'(y)$ of variable y can be omitted from this re-addition if $b(x)$ itself was an input to the last update of $b'(y)$. This means that if we tighten UB_{ij} by outflow consistency in i, we do not add the LB-outflow-consistency test for i to the queue, only the LB-inflow-consistency test for j. From this last set of tests, we might eliminate the update of LB_{ij}, but we actually only use one boolean indicator variable per set of updates.

Constraints are chosen from the queue in FIFO order. Following Davis (1987), we have also implemented a fixed sequential order, with comparable computational results. At level $p>0$, we make the csp consistent in the knowledge that it was consistent at level $p-1$: if we 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 initialised with only the updates having LB_{ij} (left branch) or UB_{ij} (right branch) as argument.

3.5 Evaluation of the objective function

Evaluation of the objective value $g(R')$ for a given flow f amounts to the PERT problem, which cannot be efficiently solved (Hagstrom, 1988; Möhring, 2000). For this reason one usually approximates the expectation of the objective function of a given policy by means of simulation (Igelmund and Radermacher, 1983b; Möhring and Radermacher, 1989; Stork, 2001). In our algorithm, if we have obtained a feasible M_p, we do not compute $g(R')$ for a feasible flow f on $TA \cup M_p$, but rather $g(M_p)$; logically we have $g(R') \leq g(M_p)$. We show that this does not change the results of the algorithm in Section 3.6 (Observation 5).

Examination of code execution has shown that some 95% is absorbed by the evaluation of the objective by means of simulation (more details are provided in Section 4.2). This part of the algorithm is clearly the bottleneck with respect 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. We can conclude that for comparison of different optimal search algorithms, the number of different solutions that need to be evaluated, is a good indicator of overall running time.

Stork (2001) 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. In our computational experiments, we strive for a constant value of the standard deviation of the percentage deviation of simulated versus 'true' makespan (the last one obtained from a high number of runs) over the dataset - it turns out that the number of simulation runs corresponding with the same standard deviation decreases with the number of activities. This approach has the advantage of reducing (relative) simulation effort for larger problem instances.

3.6 Minimal forbidden sets and minimal solutions

Branch-and-bound algorithms have been proposed for various classes of policies for the stochastic resource-constrained project scheduling problem (Igelmund and Radermacher, 1983b; Stork, 2001). For *ES*-policies, these are based on the forbidden set branching scheme, which proceeds as follows. The (subset) minimal forbidden sets (*mfs*s) are arranged in a predetermined order. Each node v in the search tree is associated with a *mfs* F and branching on v systematically resolves F by creating a child node u_{ij} of v for each ordered pair (i,j), $i,j \in F$, $i \neq j$. Each leaf v of the search tree represents a policy that is defined by resolving each *mfs* according to the decisions made on the path from v to the root of the tree. It is not mentioned in what order the branching alternatives in a specific node are to be considered, we presume an ordering based on activity indices. An obvious dominance rule can be applied (as referred to in Section 3.2): if we have added (i,j) to resolve a *mfs* higher in the search tree, and $\{i,j\} \subset F'$ with F' the next *mfs* to be resolved, then choice (i,j) to resolve F' dominates all other child nodes of v. It is easy to see how this branching scheme can be applied to solve the resource allocation problem studied in this paper: a *mfs* can now only be resolved by adding pairs $(i,j) \in R(S)$, the other branches are not considered. The project in Figure 1, for instance, has *mfs*s {1,2,4} and {3,4}, yielding *immediately* arcs (4,1) and (4,3) as only possible solution if $S=S^*$, thus making a strong case for this scheme - note, however, that extension of binary branching with constraint propagation also eliminates any need for branching.

For a feasible flow f, consider the following definition. An arc $(i,j) \in R(S)$ is *minimal* with respect to f, if apart from arc (i,j) itself, no path from i to j exists in $G(N,T(A \cup R_f))$. We call a minimal arc $(i^\circ,j^\circ) \in R(S)$ *redundant* with respect to f, if $f_{i^\circ j^\circ} \neq 0$ and a feasible flow exists on $G(N,T(A \cup R_f) \setminus (i^\circ,j^\circ))$. A feasible flow is called *minimal* if it does not contain redundant minimal arcs. From Observation 1, we have

Observation 3. *A feasible flow that is not minimal is dominated.*

We ask the reader to note that the set of solutions that we wish to examine is the set of possible (subset) *minimal* selections of arcs from $R(S)$ such that for each *mfs* F, an arc $(i,j) \in R(S)$ is selected with $i,j \in F$ (or alternatively, a feasible flow exists). On the other hand, a selection γ of arcs from $R(S)$ is defined to be *sufficient* if a minimal feasible flow f exists such that γ is the set of the minimal arcs of f that are not in TA. We notice that there is a one-to-

one relationship between sufficient selections and transitive closures $T(A \cup R)$ of minimal feasible flows f. We obtain the following result.

Observation 4. *There is a many-to-one relationship between minimal selections and sufficient selections.*

This can be seen since the sufficient selection is always subset of at least one minimal selection. The sufficient selection may not explicitly resolve all *mfs*s, but implicitly, all *mfs*s have been dealt with, since a feasible flow exists on its transitive closure. There may exist multiple ways in which to identify a 'subset minimal hitting set' of the set of arcs in the transitive closure, that explicitly undoes each of the remaining *mfs*s. The following lemma demonstrates that our algorithm spans the entire search space of minimal selections, a fact which is evident for *mfs*-branching schemes, but perhaps less intuitive for binary branching.

Lemma 3. *The transitive closure of each minimal flow is examined in at least one leaf node in the search tree (without bounding).*

With respect to a leaf node at level p of the search tree, an arc $(i,j) \in M_p$ is *minimal*, if apart from arc (i,j) itself, no path from i to j exists in $G(N,T(A \cup M_p))$. We call a minimal arc $(i°,j°) \in M_p$ *redundant* in the leaf node, if a feasible flow exists on $G(N,T(A \cup M_p) \setminus (i°,j°))$. A leaf node is called *minimal* if it does not contain redundant minimal arcs. We notice that for every minimal arc (i,j) in a leaf node, $LB_{ij} \geq 1$ (if the single machine rule adds to A, rather than M_0): the arc has been added to M by a branching decision or constraint propagation, and not as part of the transitive closure of other arcs in M. In other words, each minimal arc of the node is also a minimal arc of f, which leads to

Observation 5. *For any leaf node at level p of the search tree and any feasible flow f compatible with all branching decisions at all higher levels of the node,* $T(A \cup M_p) = T(A \cup R)$.

This justifies our approach for objective function evaluation as explained in Section 3.5, since $g(M_p)=g(R)$ for every feasible flow f compatible with the decisions corresponding with a leaf node. If the node is not minimal, neither is f, so we have

Observation 6. *Non-minimal leaf nodes can be discarded without loss of better solutions.*

3.7 Branching rules

In this section, we discuss the branching rule implemented in the binary branching scheme, and we compare binary branching and *mfs*-branching from a theoretical viewpoint. Computational comparisons are provided in Section 4.

3.7.1 *A heuristic branching rule*

In order to obtain an increase in flow in G_p'', the branching arc itself or one of the other arcs that are added to M_p, needs to create a new augmenting path from s to t. Define $S=\{i\in N\setminus\{n\} \mid i_s \text{ labelled}\}$, $T_1=\{j\in N\setminus\{0\} \mid j_t \text{ is unlabelled}\}$, $T_2=\{j\in T_1 \mid t \text{ can be reached from } j_t \text{ via an augmenting path}\}$, and $T_3=\{j\in T_2 \mid \text{flow on } (j_t,t) \text{ in } G_p'' \text{ is strictly lower than } u_j\}$. T_3 limits the augmenting path in the definition of T_2 to a single edge. The set of arcs considered for branching is set (S,T_3) if it is not empty, otherwise (S,T_2) if it is not empty, otherwise (S,T_1) (which is never void).

We limit the set of candidate arcs to include only the arcs that have nonzero flow in the remainder network; this set is never empty. In effect, we mimic the remainder flow with the partial flow: we acknowledge the flow-carrying arcs in the left branch, and afterwards destroy feasibility of the remainder flow in the right branch. Choice between eligible arcs is based on highest sum of flow in G_p' on the arc itself plus the other arcs that are added to M_p (by precedence, not constraint propagation). This sum is an estimate of the increase in flow in G_p that is achieved by the addition of the arc. A tiebreaker rule selects an arc with lowest difference between head and tail index. Multiple other evaluation criteria have been considered but turned out to lead to less efficient results. This criterion intends to branch on the minimal arcs of the remainder flow first (although this is not guaranteed).

It is difficult to apply a dominance rule based on Observation 3 earlier than just before solution evaluation, but since G_p' always contains a feasible flow, we can apply the test to the remainder network. If it were possible to maintain flow in G_p' not only feasible but also *minimal*, we could eliminate all non-minimal leaf nodes by mimicking this remainder flow by the partial network - addition of only the *minimal* arcs would suffice. Examination of this possibility is material for further research. The testing of leaf nodes for minimality before evaluation of the objective has not been implemented either, and needs further research for efficient implementation.

3.7.2 *Comparison with minimal forbidden set based branching*

An advantage of the binary branching scheme is that knowledge of the *mfs*-structure of the problem to be solved, is not indispensable: the algorithm functions correctly as long as the branching arcs are always selected from $R(S)\setminus(M_p\cup v_p)$. The choice of the branching arc has to be made heuristically: it follows from Stork (2001) that the question whether a single arc $(i,j)\in R(S)$ resolves any *mfs*, is *NP*-complete. We have developed a version of the binary branching scheme that successively branches on *mfs*s, when the complete *mfs*-structure of the problem to be solved is derived beforehand. Once all resolution alternatives for a *mfs* have been exhausted (all arcs are in v_p), the *mfs* may still be undone by transitive precedences because of decisions further down the search tree, so when this point is reached, we skip the *mfs* and consider the next. When the last *mfs* in L has been dealt with and still $\mu(M_p)<\mu_{max}$, the node can be fathomed although G_p^r may still admit a feasible flow. This algorithm was clearly outperformed by the heuristic branching rule discussed in Section 3.7.1 (although the differences were not very large).

A disadvantage of *mfs*-branching schemes is that branching is often not done on arcs that will be minimal in leaf nodes resulting from the node in which they are added; as explained in Section 3.6, addition of a sufficient selection of arcs suffices. A second disadvantage is set out in the following example.

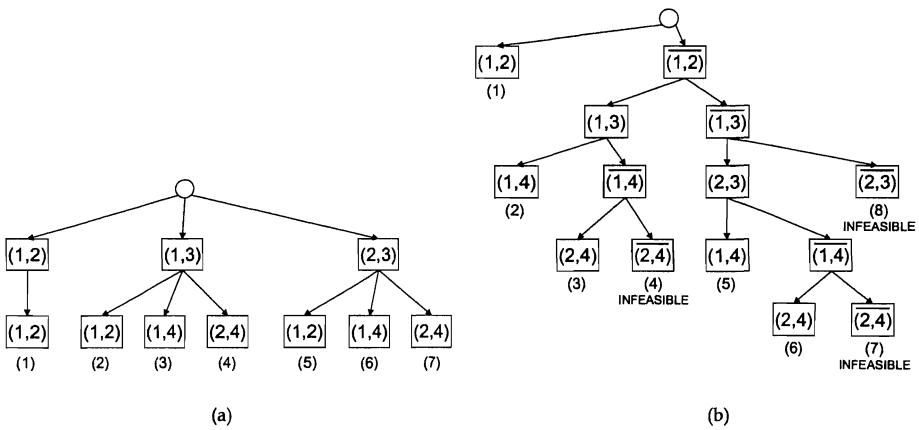

Figure 6. The search trees for (a) the *mfs*-branching scheme and (b) the binary branching scheme. The barred right branches in (b) indicate exclusion of the corresponding arc from carrying flow.

An example project has 4 non-dummy activities, with $r_1=r_2=2$, $r_3=r_4=1$, and $a=4$. The two *mfs*s are $m_1=\{1,2,3\}$ and $m_2=\{1,2,4\}$, order $L=(m_1,m_2)$, $e_1(S)\leq s_2(S)<e_2(S)\leq s_3(S)<e_3(S)\leq s_4(S)$, and the 4 activities are not precedence-related. Figure 6 pictures the search trees of the binary branching scheme *and* the *mfs*-branching scheme, both when branching arcs are chosen according to L. The binary branching scheme has 8 leaf nodes, 3 of which are recognized as infeasible and thus need no evaluation of the objective, and the remaining 5 correspond with the minimal nodes in the *mfs*-branching scheme. The *mfs*-branching scheme performs 7 evaluations of the objective, 2 of which for non-minimal nodes (labelled '(2)' and '(5)') (at most, if no bounding applies). We see that the *mfs*s are resolved one by one in the same order in L, but on backtracking from the selection of an arc to resolve a *mfs*, apart from adding a new arc, the binary branching scheme also labels the previous arc as 'forbidden'. This means that the arc will *not* be used at lower levels in the search tree for resolution of another *mfs*, which *may* occur in the associated *mfs*-branching scheme we start from, in which we can therefore recognize dominated nodes (the arc that was added first need not be present anymore, if it does not resolve other *mfs*s at the same time). This does not exclude non-minimal nodes from being visited by binary branching: consider the case where {1,2,3} is first resolved by (1,3) rather than (1,2) in Figure 6: one non-minimal leaf node would result in (b) (and still 2 in (a)).

The disadvantage of the binary branching scheme is that it distinguishes between precedence relations with and without flow, which is not always necessary. An illustration is provided in the following example. Consider a project with three non-dummy activities 1,2 and 3, the only precedence relation being $(2,3)\in A$, $r_1=a=3$ and $r_2=r_3=2$, and $s_1=0$, $s_2=e_1=1$, and $s_3=e_2=2$. The *mfs*s are {1,2} and {1,3}. A feasible flow exists both when $f_{13}=0$ and when $f_{13}=1$, and the binary branching scheme will study these 2 possibilities separately, if no constraint propagation is applied (otherwise, $LB_{12}=2$ immediately and no branching is required) and (1,3) is branched on first.

4 Computational experiments

We have implemented the algorithms in C++, using the Microsoft Visual C++ 6.0 programming environment, on a Dell XPS B800r personal computer with Pentium III processor. Section 4.1 explains the general experimental set-up. Different branching schemes are compared in Section 4.2. Section 4.3 provides details on algorithm speedup. Objective function comparisons with an allocation heuristic are the subject of Section 4.4.

4.1 Experimental set-up

The scheduling problems are 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 (Demeulemeester et al., 2002). We specify values for the order strength *OS* (values 0.2 and 0.5), the resource factor *RF* (0.7 and 0.9) 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. (2002). For various values of n, we generate 25 problem instances for each of the 2^3 parameter settings, resulting in 200 instances in total. The reader may note that not the entire domain of the problem parameters is covered, contrary to benchmark datasets such as *PSPLIB* (available at http://www.bwl.uni-kiel.de/Prod/psplib/index.html; see also Kolisch and Sprecher (1997)). Nevertheless, our choices are logical: if *OS* is large, many precedence constraints are present from the outset, and $M=\varnothing$ will regularly already admit a feasible flow. If *RF* is low, only a small number of activities have 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 made most important choices.

As mentioned before, any schedule may be the input for the resource allocation algorithm. In our experiments, we work with the schedule resulting from branch-and-bound based on deterministic baseline durations (Demeulemeester and Herroelen, 1992, 1997). The scheduling algorithm is truncated after 1 minute of CPU-time. We assume that only duration increases occur compared with the baseline plan; Gutierrez and Kouvelis (1991) provide a motivation based on expected worker behaviour under Parkinson's Law. The duration of activity i ($i \neq 0,n$) is disrupted with probability p_i, a rational number. When this occurs, its actual duration D_i exceeds its baseline duration d_i, its disruption length $L_i = D_i - d_i$ being a random variable. The probability p_i that activity i is prolonged in this way, is drawn from a uniform distribution on the interval [0;0.7]. We assume exponential activity disruption lengths, with average length if disrupted equal to the baseline duration.

After an examination of the standard deviation of the percentage deviation of simulated makespan versus the 'true' value obtained from 2000 simulations, we opt for a standard deviation of some 3%, since any lower, the standard deviation as a function of simulation count 'flattens out'. This results in 450 iterations (n=21), 350 iterations (n=31) and

300 iterations (n=41). The final evaluation of the performance of an allocation after termination of the algorithm is carried out by means of 2000 simulations.

4.2 Branching schemes

Table 1a presents computational results for the binary branching scheme. A time limit of 150 seconds on CPU-time is imposed; we report the computational results averaged over *all* instances, solved to guaranteed optimality or not, such that we in effect examine the performance of a truncated branch-and-bound heuristic. The results pertain to the branch-and-bound algorithm with *AMR* allocation (cfr. Section 4.4) as initial solution, critical path lower bound, single machine rule and schedule and flow consistency. The average number of nodes that can be visited within the time limit is around 100,000 (n=31) to 85,000 (n=41). For n=21, 92.72% of the running time of the algorithm is absorbed by simulation of the objective, which rises to 96.03% and 95.90% for n=31 and n=41, respectively.

Table 1b provides details for an implementation of the *mfs*-branching scheme. Enumeration of the *mfs*s is performed as described in Stork and Uetz (2000) and Stork (2001), with the particular implementation advantages for a single resource type, and the reduction tests; they are represented as a vector list, allowing fast access. When a *mfs* consists of two activities, it is already dealt with by the single machine rule, and is not listed. Table 2 provides some details on the *mfs*-structure of the problems in the datasets. The *mfs*s are ordered as in Stork (2001), based on the effect on the initial lower bound and on the number of branching alternatives. The objective evaluation function is borrowed from the binary branching code. Apart from changes resulting from the 'fitting' of the *ES*-policy onto a schedule, a difference with the implementation of Stork (2001) is the test whether a *mfs* is implicitly resolved: we continuously administer the set of implicit arcs in the same way as $T(A \cup M_p)$ is recorded in the binary branching scheme. This choice may induce (minor) differences in running time, but it does not influence the number of evaluations, which is a primary overall time efficiency measure. No simulation is performed for lower bound computation at intermediate levels of the search tree. The computational results obtained seem compatible with the ones in Stork (2001) for general *ES*-policies, taking into account the differences in problem characteristics, *mfs*-structure and computing system. For n=31, all 157 problems that were solved optimally by *mfs*-branching were also solved to optimality by binary branching. For this subset of the dataset, binary branching required 0.78 seconds on average, 567 nodes and 253 evaluations. The same holds for n=41, with 0.72 seconds, 435 nodes and 202 evaluations.

Table 1a. Results for the binary branching scheme, for all problems (only those solved to optimality).

	$n = 21$	$n = 31$	$n = 41$
avg. nr. nodes	1596	14178 (6118)	20097 (5853)
avg. CPU (sec)	1.49	20.77 (8.66)	34.55 (9.78)
avg. nr. evaluations	608	6585 (2809)	9171 (2689)
# optimal	100%	91.5%	82.5%

Table 1b. Results for *mfs*- branching scheme, for all problems (only those solved to optimality).

	$n = 21$	$n = 31$	$n = 41$
avg. nr. nodes	6652 (2893)	15355 (3134)	17778 (2356)
avg. CPU (sec)	11.46 (4.91)	37.75 (6.85)	64.66 (5.56)
avg. nr. evaluations	5496 (2354)	12620 (2448)	13671 (1467)
# optimal	95.5%	78.5%	61%

Table 2. Details on the *mfs*-structure of the datasets.

	$n = 21$	$n = 31$	$n = 41$
avg. # *mfss* ; only >2 activ.	146 ; 137	1542 ; 1523	11288 ; 11323
avg. # alt. 1 *mfs* >2 activ.	3.93	5.14	6.42
avg. time *mfss* info. (sec)	0	0.01	0.07

4.3 Improving efficiency

For problems with 30 non-dummy activities, we present successive improvements in the efficiency of the binary branching algorithm in Table 3. Results pertain again to the algorithm truncated after 150 seconds of CPU-time. The table indicates the average percentage of number of nodes visited, CPU-time and number of objective function evaluations when compared with the final version of the algorithm. For computation of the values in the table, for the rare cases where the reference setting (5) obtained 0, we substituted the unit of measurement, namely 1 (node or evaluation) or 0.01 (sec). The table also indicates the percentage of problems for which the optimum was guaranteed within the time limit.

Table 3. Successive improvements in the binary branching algorithm, for all problems (only those solved to optimality).

	avg. nr. nodes	avg. CPU time (sec)	number of evaluations	# optimal
(1) = B&B, *CPLB* and *AMR*	725.89% (780.36%)	137.02% (140.7%)	124.3% (126.56%)	90.5%
(2) = (1) + *single machine rule*	548.94% (584%)	118.3% (120.01%)	115.88% (117.3%)	90.5%
(3) = (1) + *schedule consistency*	595.32% (636.59%)	129.54% (132.55%)	118.79% (120.49%)	91%
(4) = (1) + *flow consistency*	432.37% (463.2%)	133.44% (136.7%)	121.32% (123.19%)	90.5%
(5) = (2) + (3) + (4)	100%	100%	100%	91.5%

We notice that the single machine rule speeds up the algorithm considerably. The combination of schedule and flow consistency and single machine rule is most efficient overall. Imposing schedule consistency alone is more efficient with regard to CPU time, whereas flow consistency alone is more efficient when it comes to number of nodes; we conclude that pursuing flow consistency is more time-intensive, but strongly reduces the search space. There is a trade-off between computation time and tightness of the domains: considering the large difference in number of nodes of the search tree, only a less than proportionate gain in average CPU-time is obtained. Nevertheless, the constraint propagation effort is more than offset by the benefits: consistency leads to less infeasible branches (forbidden arcs are recognized sooner), shorter branches (required arcs are identified sooner and hence more arcs become implicit), and the domains are tighter in the maximal flow computations, such that these take less computation time.

4.4 Objective function comparisons

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. The allocation routine can easily be uncoupled from the schedule generation; the stand-alone algorithm, denoted by *AMR*, is outlined in Appendix B. For the quality of the allocation, we compare the allocation of the binary branching algorithm (again truncated after 150 seconds) with the *AMR* result. The results are summarized in Table 4. CPU-time for *AMR* is negligible. The branch-and-bound algorithm performs significantly better than the simple allocation heuristic, but evidently requires more computational effort. Inclusion of the problems for which optimality was not guaranteed, increases the deviations, which indicates

that the hard problems have a wider variety of objective function values (and thus benefit more from optimisation).

Table 4. Deviation in objective function value of allocation heuristic *AMR* for increasing problem size (20, 30 and 40 non-dummy activities), for all problems (only those solved to optimality).

	$n = 21$	$n = 31$	$n = 41$
avg. dev. *AMR*	5.03%	6.21% (5.81%)	5.61% (5.57%)

5 Conclusions and suggestions for further research

This paper proposes a model for resource allocation for projects with variable activity durations. The allocation is required to be compatible with a deterministic pre-schedule, and the objective is to guarantee stability in activity starting times compared with the pre-schedule. We restrict our attention to the case of a single resource type, since this environment maps into a single resource network. Constraint propagation is applied during the search to accelerate the algorithm.

When day-to-day changes in job assignments are possible in a project-based organisation, resource allocation is not required to remain constant. This situation is encountered especially in single project settings, where the resources are entirely dedicated to one project. In such environments, stability is typically not very important either and makespan will be the primary objective. The search for an optimal earliest start policy, compatible or not with a baseline schedule, is of little value in such cases, since (among other policies) pre-selective policies also comply with environmental requirements, and are known to dominate the class of earliest start policies for the makespan objective.

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 stability. In view of the complexity of sequential scheduling and resource allocation, the joint approach does not appear a workable alternative for the moment.

With regard to constraint propagation, an option that is regularly mentioned in the literature (Dorndorf et al. 2000) is to obtain a certain degree of consistency, but not to pursue completion of the constraint propagation at every step, because this might be too time consuming. Another possibility is to let the consistency concept at hand vary throughout the search. These ideas require further research.

Appendix A

Proof of Theorem 1.

Suppose that f is a feasible flow and let $X=R^f$. $G(N,A\cup X)$ is acyclic because of feasibility of f. For any set of activities $\nu \subset N$ that is precedence-unrelated in $G(N,A\cup X)$, construct set $C=\nu\cup(\cup_{i\in\nu}\sigma_{T(A\cup X)}(i))$. Since $0\in N\setminus C$ and $n\in C$, $(N\setminus C,C)$ is a cut in network $G(N,A\cup X)$. We have $(C,i)=\varnothing$, $\forall i\in\nu$, such that $(N,\nu)=(N\setminus C,\nu)$, because any (j,i), $j\in C$, would imply that ν has precedence-related activities. For every $i\in C$, all $j\in\sigma_{T(A\cup X)}(i)$ are also in C, so $(C,N\setminus C)$ is empty. We see that $r(\nu)=f(N,\nu)=f(N\setminus C,\nu)\le f(N\setminus C,C)=f(N\setminus C,C)-f(C,N\setminus C)=a$ and so ν is not a forbidden set. Consequently, all forbidden sets have been implicitly resolved by f and X defines an *ES*-policy, which proves the first part of the theorem.

Suppose now that X defines a feasible *ES*-policy. It remains to be shown that a feasible flow f exists with $R^f\subseteq T(A\cup X)\setminus TA$. G^f will then automatically be acyclic because of feasibility of the policy. This part of the proof is based on Möhring (1985) (theorem numbered 1.25), who proves that (in our terminology) the minimum required capacity $a_{\min}$ to guarantee a feasible flow equals the maximum resource usage by any antichain of the partially ordered set, based on the min-flow max-cut theorem (cfr. Lawler, 1976) (an antichain being a set of precedence unrelated activities), for network $G(N,A\cup X)$ in which u_i is considered as a lower bound on flow through node i, and no transitive arcs are considered. Such a flow can always be rearranged, by addition of transitive flow-carrying arcs, to one in which u_i is both lower and upper bound on all nodes. Feasibility of the *ES*-policy guarantees that no forbidden set remain an antichain, or in other words, we have $a_{\min}\le a$, which proves the second part of the theorem. □

As hinted at by Neumann et al. (2002), the second part of this proof can also be obtained directly from the min-flow max-cut theorem, inspired especially by the accompanying remarks in Lawler (1976).

Proof of Theorem 2.

Network $G(N,A\cup R(\boldsymbol{S}))$ is acyclic, since every arc (i,j) has $e_i(\boldsymbol{S})\le s_j(\boldsymbol{S})\le e_j(\boldsymbol{S})$. For any antichain ν of the resulting partial order, it holds that $\max_{i\in\nu}s_i(\boldsymbol{S})<\min_{i\in\nu}e_i(\boldsymbol{S})$, or in other words, $\nu\subseteq N_{t^e}$, with decision point $t^e\in\delta(\boldsymbol{S})$ determined as $t^e=\min_{i\in\nu}e_i(\boldsymbol{S})$. Since feasibility of $\boldsymbol{S}$ implies that $r(N_{t^e})\le a$, the *ES*-policy defined by the partially ordered set is feasible. By Theorem 1, a feasible flow f exists with $TA\cup R^f\subseteq T(A\cup R(\boldsymbol{S}))$. We have that $T(A\cup R(\boldsymbol{S}))=TA\cup R(\boldsymbol{S})$ because $(i,j)\in T(A\cup R(\boldsymbol{S}))$ if a path exists from i to j in $G(N,A\cup R(\boldsymbol{S}))$, which may use

arcs from A and $R(S)$. If it uses only A-arcs, then $(i,j)\in TA$, otherwise $(i,j)\in R(S)$ by the definition of this last set. Hence we see that f is compatible with S because $R^f \subseteq R(S)$. □

Proof of Lemma 1.

(if) Call h a flow that realizes $\mu(M)$. For every $(i,j)\in TA\cup M$, set $f_{ij}:=h(i_s,j_t)$. The constructed f only uses arcs in $TA\cup M$ and from equality in (3.1) and (3.2) follows equality in (2.2) and (2.3). Also, as $TA\cup R(S)$ leads to an acyclic graph and $M\subseteq R(S)$, condition (2.4) is met.

(only if) Analogously, h-values can be derived from feasible f that satisfy (3.1) and (3.2) as equality. □

Proof of Lemma 2.

Define $C=N_t\cup\{i\in N \mid \exists j\in N:(j,i)\in P_t\}$. Clearly, $C\subseteq N$, and it holds that $C\cup\sigma_{TA\cup R(S)}(C)=C$, because for any $i\in C$ and $j\in\sigma_{TA\cup R(S)}(i)$ we have $(0,j)\in P_t$. From the definitions of I_t and P_t, we can see that $I_t\cup P_t=(N\setminus C,C)$, the single source node is in $N\setminus C$ and the single sink node is in C. A set of arcs that satisfies these conditions is a network cut in graph $G_{R(S)}=G(N,TA\cup R(S))$. □

Proof of Lemma 3.

We provide a proof ex absurdo. Suppose that a minimal feasible flow f exists whose transitive closure is not obtained in any leaf node of the search tree. If we consider the tree, and start at the root node, we can follow it downwards, selecting each time the branch that allows f: there is always exactly one choice, since the branching options at one level are mutually exclusive and jointly exhaustive. Once a leaf node is reached, not enough arcs have been added to M_p (by branching and transitive closure, and possibly also constraint propagation) to include all flow-carrying arcs of f. As we are in a leaf node of the search tree, a feasible flow exists on $T(A\cup M_p)\setminus v_p$, with $T(A\cup M_p)\subset T(A\cup R^f)$ and the inclusion is strict. Since only minimal arcs determine such transitive extensions, at least one minimal arc of f is redundant, so f cannot be a minimal flow and we arrive at a contradiction. □

Appendix B

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. f_{0n} is initialised with value a, all other flows are set to 0. The algorithm iteratively reroutes flow quantities

until a feasible overall flow is obtained. Condition (*) is not mentioned in the reference but seems logical.

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

Acknowledgments

The research of the first author was supported by the Fund for Scientific Research - Flanders (Belgium) (F.W.O.).

References

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.

Akturk, M.S. and Gorgulu, E. (1999). Match-up scheduling under a machine breakdown. *European Journal of Operational Research*, **112**, 81-97.

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.

Aytug, H., Lawley, M.A., McKay, K., Mohan, S. and Uzsoy, R. (2002). Executing production schedules in the face of uncertainties: a review and some future directions. To appear in: *European Journal of Operational Research.*

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

Brearley, A.L., Mitra, G. and Williams, H.P. (1975). An analysis of mathematical programs prior to applying the simplex method. *Mathematical Programming*, **8**, 54-83.

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.

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. (2002). A Random generator for activity-on-the-node networks. *Journal of Scheduling*, to appear.

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.

Fulkerson, D.R. (1962). Expected critical path lengths in PERT networks. *Operations Research*, **10**, 808-817.

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 construction of stable project baseline schedules. *Research Report 0220, Department of Applied Economics, Katholieke Universiteit Leuven, Belgium.*

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

Igelmund, G. and Radermacher, F.J. (1983b). Algorithmic approaches to preselective strategies for stochastic scheduling problems. *Networks*, **13**, 29-48.

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

Kolisch, R. and Sprecher, A. (1997). PSPLIB - A project scheduling library. *European Journal of Operational Research*, **96**, 205-216.

Lawler, (1976). *Combinatorial optimization: networks and matroids.* Holt, Rinehart and Winston.

Leus, R. and Herroelen, W. (2001). Models for robust resource allocation in project scheduling. *Research report 0128, Department of Applied Economics, Katholieke Universiteit Leuven, Belgium.*

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

Mehta, S.V. and Uzsoy, R.M. (1998). Predictable scheduling of a job shop subject to breakdowns. *IEEE Transactions on Robotics and Automation*, **14**, 3, 365-378.

Möhring, R.H. (1985). Algorithmic aspects of comparability graphs and interval graphs. *In:* Rival, I. (ed.). *Graphs and Orders*, 41-101. Reidel Publishing Company.

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

Möhring, R.H. and Radermacher, F.J. (1989). The order-theoretic approach to scheduling: the stochastic case. *Chapter III.4 in:* Slowinski and Weglarz.

Naegler, G. and Schoenherr, S. (1989). Resource allocation in a network model - the Leinet system. *Chapter II.8 in*: Slowinski and Weglarz.

Neumann, K., Schwindt, C. and Zimmermann, J. (2002). *Project scheduling with time windows and scarce resources.* Lecture Notes in Economics and Mathematical Systems, Springer.

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

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

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

Schwindt, C. (2001). Project scheduling with sequence-dependent changeover times. *Paper presented at the EURO 2001 meeting in Rotterdam, July 9-11.*

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

Stork, F. (2001). *Stochastic resource-constrained project scheduling.* Ph.D. Thesis, TU Berlin.

Stork, F. and Uetz. M. (2000). On the representation of resource constraints in project scheduling. *Technical report 693/2000, Department of Mathematics, Technische Universität Berlin, Germany.*

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

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

Wu, S.D., Storer, H.S. and Chang, P.-C. (1993). One-machine rescheduling heuristics with efficiency and stability as criteria. *Computers and Operations Research,* **20**(1), 1-14.