KATHOLIEKE UNIVERSITEIT
LEUVEN

Faculty of Business and Economics

Ú¦[b^&cÁ•&@^å˘ |ã}*Á¸ ãc@Á{ [å˘ |æ¦Á]¦[b^&cÁ&[{] |^cã[} [} Áæ&Áà[cc|^} ^&\ Á^•[˘ ¦&^

S¦ã•ÁÔ[[|^} ÊÁY ^} &@æ[ÁY ^ãÊÁØæà¦ã&^Á/æ|æ|æÁÞ[àãa[} Áæ} åÁÜ[^|ÁŠ^˘ •

DEPARTMENT OF DECISION SCIENCES AND INFORMATION MANAGEMENT (KBI)

KBI FFGI

PROJECT SCHEDULING WITH MODULAR PROJECT COMPLETION ON A BOTTLENECK RESOURCE

KRIS COOLEN[†], WENCHAO WEI[†], FABRICE TALLA NOBIBON[†‡], AND ROEL LEUS[†§]

Abstract. In this paper, we model a research-and-development project as consisting of several modules, with each module containing one or more activities. We examine how to schedule the activities of such a project in order to maximize the expected profit when the activities have a probability of failure and when an activity's failure can cause its module and thereby the overall project to fail. A module succeeds when at least one of its constituent activities is successfully executed. All activities are scheduled on a scarce resource that is modeled as a single machine. We describe various policy classes, establish the relationship between the classes, develop exact algorithms to optimize over two different classes (one dynamic program and one branch-and-bound algorithm), and examine the computational performance of the algorithms on two randomly generated instance sets.

Key words. scheduling, uncertainty, research and development, activity failures, modular precedence network

AMS subject classifications. 90B35, 90B36, 90C27, 68M20

1. Introduction. Activities in a practical project are typically subject to many uncertainties; the most frequently studied types of uncertainty are resource breakdowns and duration variability. In research and development (R&D), activities may also fail altogether, for instance because the new technology under study does not perform as anticipated or because a toxicity test is not passed (in case of drug development). We model an R&D project as consisting of several *modules*, with each module containing one or more activities that pursue a homogeneous target, for instance representing repeated trials or technological alternatives. Each activity has a cost, a duration and a probability of success. A module is successful when at least one of its included activities succeeds. The successful completion of the whole project requires the successful completion of all the modules; project success equates with receiving a project payoff (cash inflow). This setup is subsequently referred to as 'modular project completion'. The objective is to schedule the activities in such a way that a maximum expected profit is attained. A solution to this scheduling problem is a *policy*, which is a dynamic decision rule that decides which activities are to be started at which time. We examine the scheduling of the project activities on a single machine, representing a scarce or bottleneck resource. Examples of such scarce resources are specialized equipment, or departments or individuals with specific areas of expertise (see Kavadias and Loch [14] for a similar motivation in a slightly different setting).

The main contributions of this paper are fivefold: (1) we introduce and formulate a generic model for optimally scheduling R&D projects with modular completion; (2) we describe various scheduling policy classes and examine the relationship between the classes; (3) we provide an analysis of a number of properties; (4) we develop exact algorithms to optimize over two different classes (one dynamic program and one branch-and-bound algorithm); and (5) we examine the computational performance of the algorithms on two randomly generated instance sets.

[†]ORSTAT, Faculty of Business and Economics, KULeuven, Leuven, Belgium

[‡]QuantOM, HEC-Management School, University of Liège, Liège, Belgium

[§]Corresponding author. E-mail: Roel.Leus@econ.kuleuven.be. Postal address: ORSTAT, Faculty of Business and Economics, KULeuven, Naamsestraat 69, B-3000 Leuven, Belgium.

The remainder of this text is organized as follows. Section 2 contains some definitions, a formal problem statement and a brief survey of related work. Various classes of scheduling policies are presented in Section 3, and a number of properties are studied in Section 4. Our algorithms are described in Section 5, and their computational performance is tested in Section 6. Finally, Section 7 contains a summary and some conclusions.

2. Problem statement and related work.

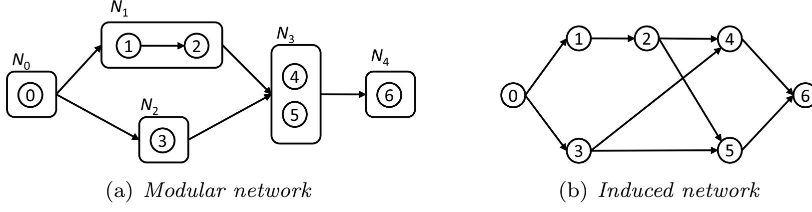
2.1. Definitions. Consider the planning of one project in isolation, consisting of a set $N = \{0, 1, \ldots, n+1\}$ of *jobs* or *activities* (these two terms will be used interchangeably) to be scheduled on a single machine. The job set is partitioned into a set of non-empty *modules* $M = \{0, 1, \ldots, m+1\}$. Let N_i denote the set of jobs belonging to module $i \in M$, then $N = \cup_{i \in M} N_i$ and $N_i \cap N_j = \emptyset$ if $i \neq j$. Activities in the same module pursue a similar target. The (dummy) modules 0 and $m+1$ represent start and end of the project and contain only one (dummy) activity, indexed by 0 and $n+1$ respectively.

Each activity $k \in N \backslash N_{m+1}$ has a probability of technical success (PTS) p_k; we assume that $p_0 = 1$. We consider the outcomes of the different tasks to be independent. Value $q_k = 1 - p_k$ is the probability of failure of activity k. In practice, each activity also has a specific duration, but this is not relevant to this article as we do not discount the cash flows, or more generally because the objective function is not a function of the start or completion times of the individual jobs.

A strict partial order is an irreflexive and transitive binary relation; below we omit the qualifier 'strict' for brevity. Jobs within a module i are subjected to precedence constraints represented by a partial order B_i on N_i. A partial order A on the set of modules M is also part of the input, and an activity in a particular module can only start when all predecessor modules are successful. A module is defined to be successful if at least one of its constituent activities succeeds. The project is said to be successful when all modules are successful. The foregoing definitions lead to an object $(M, A, (N_i, B_i)_{i \in M})$, which will be called the *modular network*. Furthermore, we define the order B^* on set N to relate activities that do not necessarily belong to the same module, as follows: $(k, l) \in B^* \Leftrightarrow (\exists B_i : (k, l) \in B_i) \vee (\exists (i, j) \in A : (k \in N_i) \wedge (l \in N_j))$. The digraph with node set N and arc set B^* is referred to as the *induced network* of the modular network.

Quantity $c_k \geq 0$ represents the cost of processing activity $k \in N \backslash N_{m+1}$; these costs are incurred at the start of each activity. We let $c_0 = 0$. The value $V > 0$ denotes the end-of-project payoff that is received at the execution of the dummy end job $n+1$; this payoff is obtained only when all modules are successful. Our goal is to schedule the activities such as to maximize the expected profit. In the remainder of the article, we refer to this problem as MP1 (short for 'Modular Project scheduling on One machine'). An instance of MP1 corresponds to a tuple $(M, A, (N_i, B_i)_{i \in M}, \mathbf{p}, \mathbf{c}, V)$, with $\mathbf{p}$ and $\mathbf{c}$ two n-vectors whose components are the p_i and c_i, respectively, for $i \notin \{0, n+1\}$.

For an illustration of these definitions, we present an example instance with modular network and induced network given in Fig. 2.1. The project consists of seven activities, $N = \{0, 1, 2, 3, 4, 5, 6\}$, where job 0 is the dummy start job and $n+1 = 6$ represents the dummy end job. The jobs are partitioned into $5 = m+2$ modules, so $M = \{0, 1, 2, 3, 4\}$ with $N_0 = \{0\}$, $N_1 = \{1, 2\}$, $N_2 = \{3\}$, $N_3 = \{4, 5\}$ and $N_4 = \{6\}$. In the example, $B_i = \emptyset$ for $i \in M \backslash \{1\}$ and $B_1 = \{(1, 2)\}$. For a binary relation R on a set S, let $T(R)$ denote its *transitive closure*, defined as the minimal transitive



(a) *Modular network* (b) *Induced network*

Fig. 2.1. *Graphical representation of a modular network of an MP1 instance with five non-dummy jobs partitioned into three non-dummy modules (left) and the corresponding induced network (right)*

relation on S that contains R. The *transitive reduction* $t(R)$ of relation R is the minimal relation on S such that $T(t(R)) = T(R)$. The set A is the transitive closure of the set $\{(0,1),(0,2),(1,3),(2,3),(3,4)\}$. Fig. 2.1(b) shows the induced network of the modular network of Fig. 2.1(a). The partial order B^* is the transitive closure of the relation $\{(0,1),(0,3),(1,2),(2,4),(2,5),(3,4),(3,5),(4,6),(5,6)\}$. Note that in the graphical representation of both the modular and the induced network, only the transitive reduction of the partial order is shown.

We define a *state vector* as an n-component binary vector $\mathbf{x} = (x_1, \ldots, x_n)$ with one component associated with each non-dummy activity i in N, denoting the success $(x_i = 1)$ or failure $(x_i = 0)$ of activity i. A state vector is also called a *scenario*. Let X_i represent the Bernoulli random variable with parameter p_i of success of activity i, and denote by $\mathbf{X} = (X_1, \ldots, X_n)$ the associated vector of random variables. The realization of each X_i is known only at the end of activity i. A *schedule* $\mathbf{s}$ for a project is an ordered subset of its non-dummy activities; by s_t we denote the job in position t in the schedule $\mathbf{s}$. A schedule $\mathbf{s}$ is said to be *feasible* with respect to a scenario $\mathbf{x}$ if for all $i \in M$, for all $l = s_u \in N_i$, and

(F1) for all k with $(k,l) \in B_i$, we have $k = s_t$ for some $t < u$;

(F2) for all j with $(j,i) \in A$, we have $k = s_t \in N_j$ for some $t < u$ with $x_k = 1$.

Remark that a schedule may imply a *selection* of activities: not all elements of N need to be retained. Requirement (F1) expresses that the precedence constraints within a module must be respected (and is independent of the scenario), while requirement (F2) states that the precedence constraints between the modules must not be violated, which does depend on the realization of $\mathbf{X}$. Let $\Sigma_{\mathbf{x}}$ denote the set of all schedules feasible with respect to $\mathbf{x}$ and $\Sigma = \bigcup_{\mathbf{x} \in \mathbb{B}^n} \Sigma_{\mathbf{x}}$. A feasible schedule $\mathbf{s}$ is called *successful* for scenario $\mathbf{x}$ when it results in a successful project, i.e. when

$$\forall\, i \in M \backslash \{0, m+1\}, \exists\, k = s_t \in N_i \text{ with } x_k = 1.$$

The Boolean success function $\sigma(\mathbf{x}, \mathbf{s})$ takes value 1 if $\mathbf{s}$ is successful for $\mathbf{x}$, 0 otherwise. For a scenario $\mathbf{x}$ and schedule $\mathbf{s} \neq \emptyset$ that is feasible for $\mathbf{x}$, define the *profit* as

$$f(\mathbf{x}, \mathbf{s}) = \sigma(\mathbf{x}, \mathbf{s}) \cdot V - \sum_{t=1}^{|\mathbf{s}|} c_{s_t},$$

where $|\mathbf{s}|$ is the length of $\mathbf{s}$. When $\mathbf{s} = \emptyset$ we set $f(\mathbf{x}, \mathbf{s}) = 0$.

For the example project described above, consider the scenario $\mathbf{x}_1 = (0,1,1,0,0)$: activities 2 and 3 succeed, but 1, 4 and 5 fail. The schedule $\mathbf{s}_1 = (1,2,3,4,5)$ is feasible for $\mathbf{x}_1$, but $\sigma(\mathbf{x}_1, \mathbf{s}_1) = 0$: the project fails. Under this scenario, there is no feasible

schedule that can obtain the project payoff. The scenario $\mathbf{x}_2 = (1,0,1,1,0)$, on the other hand, allows for the payoff to be achieved: schedule $\mathbf{s}_2 = (1,3,4)$, for instance, is successful in this case. Note that only part of the activities in N are executed by $\mathbf{s}_2$ and that this schedule would be successful under all scenarios of the format $(1,-,1,1,-)$, where $-$ is either 0 or 1. When all $c_i = 1$, $i = 1,\ldots,5$, and $V = 4$, we have $f(\mathbf{x}_1,\mathbf{s}_1) = -5$ (a negative profit of -5, or loss of 5), while $f(\mathbf{x}_2,\mathbf{s}_2) = 1$.

2.2. Problem statement. A solution to problem MP1 is a *policy*: a decision rule that decides in which sequence to start which activities. Formally, a policy Π is a function $\Pi : \mathbb{B}^n \mapsto \Sigma$, mapping scenarios $\mathbf{x}$ to schedules that are feasible with respect to $\mathbf{x}$ and that satisfy the following constraint:

(NA) if $[\Pi(\mathbf{x})]_u = l$ for an arbitrary job l and position u then also $[\Pi(\mathbf{y})]_u = l$ for all scenarios $\mathbf{y}$ that have $y_k = x_k$ for all jobs $k = [\Pi(\mathbf{x})]_t$, $t < u$.

In the foregoing, we use the notation $[\mathbf{z}]_t$ for the tth component of a vector $\mathbf{z}$. Requirement (NA) is often called *non-anticipativity constraint* and ensures that the decision made at any time t can only be based on information that became available before or at time t (in our case, time can be treated as the position of the jobs). In particular we have $[\Pi(\mathbf{x})]_1 = [\Pi(\mathbf{y})]_1$, $\forall \mathbf{x},\mathbf{y} \in \mathbb{B}^n$. We refer to [23, 27] for more details and further references on the use of policies as functions in stochastic scheduling.

The dynamic character of a policy as a *dynamic decision process* is somewhat concealed by its representation as a function. For this reason, it is sometimes useful to adopt an alternative representation by a *binary decision tree*, which is in line with the literature on sequential testing (see [28], for instance). In such a tree, the non-leaf nodes represent the scheduling of a non-dummy job and are labeled with the index of the job. From a non-leaf node labeled k, two decision branches emanate. The left arc represents a scenario where job k fails ($x_k = 0$) and analogously the right arc implies success of job k ($x_k = 1$). The leaf nodes represent either success or failure (abandonment) of the project. To each leaf node corresponds a unique schedule: the job in position u of this schedule is precisely the label of the u-th node encountered while traversing the unique path from the root to the leaf node. When this schedule is successful, the corresponding leaf node is labeled 'S' (for success). In the other case the project is abandoned and the node is labeled 'F' (failure). For convenience, we make a slight abuse of notation in the remainder of the paper by using the same symbol k for a node and its corresponding job label.

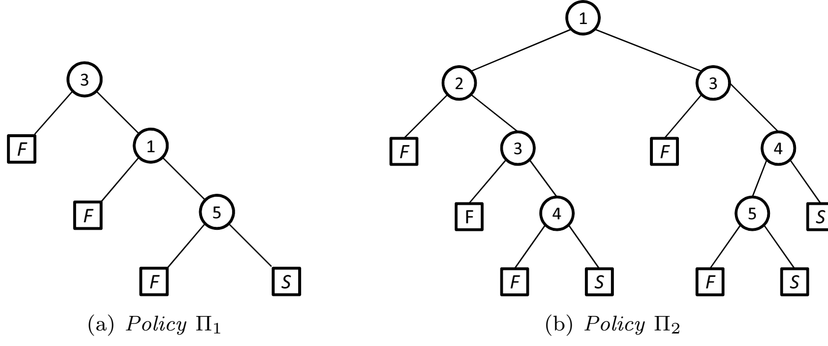
Fig. 2.2 shows two policies for the example project presented earlier. Policy Π_1 schedules only one job of each module and the project is abandoned as soon as a job failure is encountered. Policy Π_2 starts with job 1 and in case of failure, job 2 is executed. Depending on the outcome of job 1, module 3 is treated differently: in case of failure for 1, only job 4 is selected while if $x_1 = 1$ then job 5 is started in case of failure for job 4.

The problem MP1 under study boils down to selecting a policy Π^* within a specific class $\mathcal{C}$ of policies that maximizes the expected profit of the project:

$$\Pi^* = \arg\max_{\Pi\in\mathcal{C}} \mathbb{E}[f(\mathbf{X},\Pi(\mathbf{X}))],$$

with $\mathbb{E}$ the expectation operator. In the remainder of this text, we write $\mathbb{E}[f(\Pi)]$ instead of $\mathbb{E}[f(\mathbf{X},\Pi(\mathbf{X}))]$ to simplify notation. As an illustration, for policy Π_1 described in Fig. 2.2(a) we have

$$\mathbb{E}[f(\Pi_1)] = Vp_3p_1p_5 - c_3 - p_3c_1 - p_3p_1c_5.$$



(a) *Policy* Π_1 (b) *Policy* Π_2

Fig. 2.2. *Two policies for the example project of Fig. 2.1(a)*

In general terms, using the definition of a policy as a mapping, we obtain

$$\mathbb{E}[f(\Pi)] = \sum_{\mathbf{x}\in\mathbb{B}^n} \left(\prod_{i:x_i=1} p_i\right)\left(\prod_{i:x_i=0} q_i\right) f(\mathbf{x},\Pi(\mathbf{x})).$$

Manually, the expected profit of a policy Π is usually more easily computed using the tree representation T of the policy. For an arbitrary node k, define $C(k)$ to be the set of jobs on the path from the root of T to node k excluding node k, and let $C_1(k)$, resp. $C_0(k)$, be the subset of $C(k)$ containing only jobs for which the arc leading to the successor job in the path is a right, resp. left, arc. The collection of all the leaf, resp. non-leaf, nodes of T is denoted as $L(T)$, resp. $NL(T)$. Then

$$\mathbb{E}[f(\Pi)] = \sum_{k\in L(T)} \mathrm{Prob}(k)\left(\overline{V}_k - \sum_{l\in C(k)} c_l\right),$$

where $\mathrm{Prob}(k) = \left(\prod_{l\in C_1(k)} p_l\right)\cdot\left(\prod_{l\in C_0(k)} q_l\right)$ is the probability of reaching node k, and

$$\overline{V}_k = \begin{cases} V & \text{if } k \text{ is labeled } S,\\ 0 & \text{if } k \text{ is labeled } F.\end{cases}$$

Equivalently,

$$\mathbb{E}[f(\Pi)] = \sum_{k\in L(T)} \mathrm{Prob}(k)\cdot\overline{V}_k - \sum_{k\in NL(T)} \mathrm{Prob}(k)\cdot c_k. \tag{2.1}$$

From Eq. (2.1) and by stepwise updating $\mathrm{Prob}(k)$ from root node to leaf nodes, we have the following observation:

Observation 2.1. *Evaluation of an arbitrary policy can be done in time linear in the number of nodes in the decision tree.*

Since an arbitrary policy can be defined by describing its decision tree, this time complexity is probably the best one can hope to obtain. Furthermore, the size of the decision tree of an arbitrary policy may be exponential in the number of jobs. Evaluation of special classes of policies is discussed in Section 4.

2.3. Related work. Extensive literature surveys on the topic of scheduling under uncertainty are provided in [1, 7, 11, 24, 30]. The main topic of interest in these sources is duration uncertainty, sometimes complemented with uncertain resource availabilities. In this text, we incorporate the concept of activity success or failure into the scheduling decisions. De Reyck and Leus [9] develop an algorithm for project scheduling with uncertain activity outcomes, where project success is achieved only if all individual activities succeed. A similar model is tackled by Schmidt and Grossmann [25] and Jain and Grossmann [12], who study the scheduling of failure-prone new-product-development testing tasks when non-sequential testing is admitted. In the foregoing references, however, the possibility of pursuing multiple alternatives to achieve the same result (modular completion) is not included. This concept of modular completion is hinted at in the informal paper [8], but resource constraints are not considered and no solution procedures are proposed. The work of De Reyck et al. [8] is continued by Creemers et al. [5], who also study modular completion but with a focus especially on the impact of activity duration variability on the project's value, while we work with deterministic durations. Creemers et al. also neglect resource constraints, while we are scheduling on a single machine. Malewicz [19] studies parallel machine scheduling where tasks are executed by unreliable machines, and the probability for correct execution of each job by each machine is known. The goal is to find a policy that assigns tasks to machines (possibly in parallel and redundantly) so as to minimize expected completion time; the same task can be executed more than once.

Closely related to the model developed in this paper is the work on sequential testing, in which a series of tests is to be performed to diagnose a system (i.e., to know its state, which usually is either 'working' or 'failing'). A solution in this setting is an inspection strategy, which specifies on the basis of the state of the already inspected components which component is to be inspected next, or halts if it is able to recognize the correct state of the system. Reviews of this body of literature can be found in Boros and Ünlüyurt [2] and Ünlüyurt [28]. The main differences with our scheduling problem are twofold: (1) the inspections will continue as long as the state of the system is not known, while we allow the project to be aborted preliminarily if this is better for the project's value, and (2) most of the work in this area has focused on diagnosing so-called 'k-out-of-n' systems, where the system functions if k or more of its components work. In our model, the success of a project is dependent on the success of its constituent modules, and so a project's success is not merely determined by the number of successful activities.

3. Special classes of policies. Both the representation of a policy as a mapping and as a decision tree allow to conclude that the number of scheduling policies for an MP1 instance is finite. The class of all policies is denoted by $\mathcal{C}_{ALL}$. An optimal policy in $\mathcal{C}_{ALL}$ is called *globally optimal*. In this section we distinguish different policy classes, study their characteristics, and examine the relationship between them. To measure the quality of a policy class for a given MP1 instance, we define the *relative optimality gap* $\gamma(\mathcal{C})$ of a policy class $\mathcal{C}$ as the relative deviation from the global optimum, i.e.

$$\gamma(\mathcal{C}) = \begin{cases} \frac{\pi(\mathcal{C}_{ALL})-\pi(\mathcal{C})}{\pi(\mathcal{C}_{ALL})} & \text{if } \pi(\mathcal{C}_{ALL}) \neq 0, \\ 0 & \text{otherwise,} \end{cases}$$

where $\pi(\mathcal{C})$ denotes the expected profit of a policy that is optimal in policy class $\mathcal{C}$. A relative gap $\gamma(\mathcal{C}) = 0$ implies that an optimal policy of class $\mathcal{C}$ is also a globally

optimal policy. The other extreme, $\gamma(\mathcal{C}) = 1$, occurs when all policies of $\mathcal{C}$ have non-positive expected profit whereas a globally optimal policy has positive expected profit.

3.1. Dominance results. We define the following properties:

(R1) When a job of a module is executed with a failure and all other jobs of the module were previously scheduled without success, the project is abandoned.

(R2) When a job of a module is executed successfully, no other unscheduled job of the same module is scheduled after this job.

(R3) After activity success, the project is never abandoned immediately, i.e. either payoff is obtained or another job is scheduled.

A policy satisfying the properties (R1)–(R3) will be called a *reasonable policy*; all reasonable policies are gathered in set $\mathcal{C}_{REA}$. The example policies in Fig. 2.2 are clearly reasonable. We have the following dominance result:

OBSERVATION 3.1. *There exists a reasonable globally optimal policy for MP1, i.e.* $\gamma(\mathcal{C}_{REA}) = 0$.

Next we describe a subclass of $\mathcal{C}_{REA}$ with a specific structure. To this end, we define a pair of distinct nodes k_1 and k_2 of the decision-tree representation of a reasonable policy to be *equivalent* if they satisfy the properties (E1)–(E3) below.

(E1) The incoming arcs of k_1 and k_2 are right (success) arcs.

(E2) $\forall i \in M : C_1(k_1) \cap N_i \neq \emptyset \Leftrightarrow C_1(k_2) \cap N_i \neq \emptyset$.

(E3) $\forall i \in M : C_1(k_1) \cap N_i = \emptyset \Rightarrow C_0(k_1) \cap N_i = C_0(k_2) \cap N_i$.

A policy $\Pi \in \mathcal{C}_{REA}$ is called a *dominant* policy if the subtrees emerging from every pair of equivalent nodes of its decision-tree representation are identical. The set of dominant policies is denoted by $\mathcal{C}_{DOM}$. The policy in Fig. 2.2(a) is dominant as it contains no equivalent pair of nodes. The policy depicted in Fig. 2.2(b), on the other hand, is not an element of $\mathcal{C}_{DOM}$, which can be seen by considering the equivalent nodes labeled with job 3: the subtrees emerging from these two nodes differ in whether or not to perform job 5 after failure of job 4. The next theorem is a stronger result than Observation 3.1.

THEOREM 3.1. *There exists a dominant policy for MP1 that is globally optimal, i.e.* $\gamma(\mathcal{C}_{DOM}) = 0$.

Proof. Let $\mathcal{I}$ be an instance of MP1 and consider a globally optimal policy Π for instance $\mathcal{I}$ with decision tree T and expected profit π. We can assume Π to be reasonable because of Observation 3.1. If Π is not a dominant policy, we can choose two equivalent nodes k_1 and k_2 for which the subtrees T_1 resp. T_2 with root node k_1 resp. k_2 are not identical. Let $\mathcal{I}'$ be the MP1 instance derived from $\mathcal{I}$ by deleting the modules i for which $C_1(k_1) \cap N_i \neq \emptyset$ as well as the jobs $k \in C_0(k_1)$ inside modules i for which $C_1(k_1) \cap N_i = \emptyset$. Subtrees T_1 resp. T_2 represent policies for instance $\mathcal{I}'$ with expected profits π_1 resp. π_2. Assume $\pi_1 \geq \pi_2$. Replacing T_2 by T_1 in T gives rise to another policy for $\mathcal{I}$ with expected profit $\pi + \text{Prob}(k_2)(\pi_1 - \pi_2) \geq \pi$. This procedure can be repeated until all equivalent nodes of T have identical subtrees emerging from them. ◻

Below, we proceed with the description of a number of subclasses of $\mathcal{C}_{REA}$ that have a compact combinatorial representation, enabling simpler implementation, similar to Stork's [27] treatment of scheduling policies for stochastic resource-constrained project scheduling. Somewhat counter-intuitively, however, we will observe in Section 6 that the subclasses do not allow for faster search procedures in our implementations. Section 3.2 presents elementary policies and module-wise policies are the subject of Section 3.3.

3.2. Elementary policies. In the sequel, we use both the terms 'ordering' and '(order) list' to refer to a complete order on a job set, represented as a permutation $L = (j_1, j_1, \ldots, j_{|L|})$. Denote by $L(t)$ the t-th element of L, so $L(t) = j_t$. The class $\mathcal{C}_E$ of *elementary* policies is inspired by priority rules for deterministic scheduling [16, 17]: each $\Pi \in \mathcal{C}_E$ is characterized by a compatible ordering L of a subset of $N \setminus \{0, n+1\}$. We call a list L *compatible* if either $L = \emptyset$ or when conditions C(1)–C(4) below hold.

(C1) For each non-dummy module i, there is at least one job $k \in N_i$ in list L.

(C2) If a job l of module i belongs to L then all jobs k with $(k, l) \in B_i$ appear in the list before job l.

(C3) If a job l of module i belongs to L then for each module j with $(j, i) \in A$ there must be at least one job of module j in the list before job l.

(C4) If a job of module i belongs to L then all jobs that appear earlier in the list are in modules j for which $(i, j) \notin A$.

Remark that condition (C3) is redundant because it is implied by (C1) and (C4). The condition is listed nevertheless because it will be needed for the definition of a 'compatible partial list' (see Section 5.2), in which condition (C1) does not necessarily hold.

For a given scenario $\mathbf{x}$, the elementary policy $\Pi(\,\cdot\,; L)$ parameterized by compatible list L generates a unique schedule $\Pi(\mathbf{x}; L)$ as described by Algorithm 1. Policy Π_1 in Fig. 2.2(a) is elementary with $L = (3, 1, 5)$, while policy Π_2 in Fig. 2.2(b) is not elementary due to its different treatment of jobs 4 and 5 according to the outcome of job 1. We observe that elementary policies are reasonable policies. A yet stronger result is the following:

THEOREM 3.2. *Elementary policies are dominant, i.e.* $\mathcal{C}_E \subset \mathcal{C}_{DOM}$.

Proof. Consider an elementary policy $\Pi(\,\cdot\,; L)$ for an arbitrary MP1 instance $\mathcal{I}$. We know that an elementary policy is reasonable. Choose two arbitrary equivalent nodes $k_1 = L(t_1)$ and $k_2 = L(t_2)$ (we slightly abuse notation by identifying a node k and its corresponding job label). If $t_1 = t_2 = t$, both subtrees must be identical as they are both completely determined by the sublist of L obtained by deleting the first $t - 1$ elements of L. Next, we assume that $t_1 < t_2$; we will show that this situation never occurs. Denote by i the module containing k_1. Note that, according to (R2), $C_1(k_1) \cap N_i = \emptyset$. We will derive a contradiction for every possible occurring case. If we assume $k_1 \in C(k_2)$, then either $k_1 \in C_1(k_2)$ or $k_1 \in C_0(k_2)$. If $k_1 \in C_1(k_2)$ then condition (E2) implies $C_1(k_1) \cap N_i \neq \emptyset$ and a contradiction is found. If $k_1 \in C_0(k_2)$, on the other hand, then condition (E3) would imply $k_1 \in C_0(k_1)$, which is impossible.

Algorithm 1 Schedule generation by elementary policy $\Pi(\,\cdot\,; L)$ for scenario $\mathbf{x}$

```
1: s = ∅
2: while L ≠ ∅ do
3:    Remove first job k from L and append it to the end of s; let i be such that
      k ∈ N_i
4:    if (x_k = 0) ∧ (no other job of N_i appears in L) then
5:       Return s
6:    else if x_k = 1 then
7:       Delete all other jobs of N_i from L
8:    end if
9: end while
10: Return s
```

Finally, if $k_1 \notin C(k_2)$ we can choose $t < t_1$ with $L(t) \in C_1(k_2) \cap N_i$. Again by (E2), $C_1(k_1) \cap N_i \neq \emptyset$, which completes the proof. □

Define an *n:n-system* ('n-out-of-n-system') or *single-activity-module project* as an instance of MP1 where each module contains exactly one activity. For n:n-systems, every job needs to be executed successfully in order to win the project payoff. A 1:*n-system* ('1-out-of-n-system') or *single-module project*, on the other hand, contains only one non-dummy module holding all non-dummy jobs and the project succeeds at the completion of the first successful job. Note that for a general 1:n-system, B_1 need not be empty. A result similar to the theorem below is shown in [8] for the setting without resource constraints.

THEOREM 3.3. *Elementary policies are globally optimal for* n:n*-systems and* 1:n*-systems.*

Proof. First consider an n:n-system. Let Π^* be any globally optimal policy and put $L = \Pi^*(\mathbf{1})$, with $\mathbf{1}$ the scenario where all jobs are successful. We must show that $\mathbb{E}[f(\Pi(\mathbf{X}; L))] = \mathbb{E}[f(\Pi^*)]$. Consider an arbitrary state vector $\mathbf{x}$ and let u be the smallest number such that activity $[\Pi^*(\mathbf{x})]_u$ fails. Because of the non-anticipativity constraint, $[\Pi^*(\mathbf{x})]_t = [\Pi^*(\mathbf{1})]_t$ for all $t < u$. Because $[\Pi^*(\mathbf{1})]_t = [\Pi(\mathbf{x}; L)]_t$ for all $t < u$, it follows that policies Π^* and $\Pi(\,\cdot\,; L)$ behave the same for scenario $\mathbf{x}$ for all positions smaller than u. Since the elementary policy will abandon the project at time u, we have $f(\Pi(\mathbf{x}; L)) \geq f(\Pi^*(\mathbf{x}))$. If no such position u exists then non-anticipativity implies that Π^* and $\Pi(\,\cdot\,; L)$ behave exactly the same for scenario $\mathbf{x}$ and for all positions and $f(\Pi(\mathbf{x}; L)) = f(\Pi^*(\mathbf{x}))$. Consequently, $\mathbb{E}[f(\Pi(\mathbf{X}; L))] \geq \mathbb{E}[f(\Pi^*)]$. The inverse inequality follows from the optimality of Π^*.

Now consider a 1:n-system. The proof proceeds analogously to the n:n case. Again, Π^* is any globally optimal policy but now we show $\mathbb{E}[f(\Pi(\mathbf{X}; L))] = \mathbb{E}[f(\Pi^*)]$ for $L = \Pi^*(\mathbf{0})$, where $\mathbf{0}$ is the scenario in which all jobs fail. For an arbitrary state vector $\mathbf{x}$, let u be the smallest number such that activity $[\Pi^*(\mathbf{x})]_u$ is successful. By non-anticipativity, the schedules corresponding to $\mathbf{x}$ on the one hand and to $\mathbf{0}$ on the other hand are the same for all positions smaller than u. When no such u exists, these schedules are identical. Since the elementary policy will not execute any other job after the first successful job (it reaps the payoff), it follows that $f(\Pi(\mathbf{x}; L)) \geq f(\Pi^*(\mathbf{x}))$, which completes the proof. □

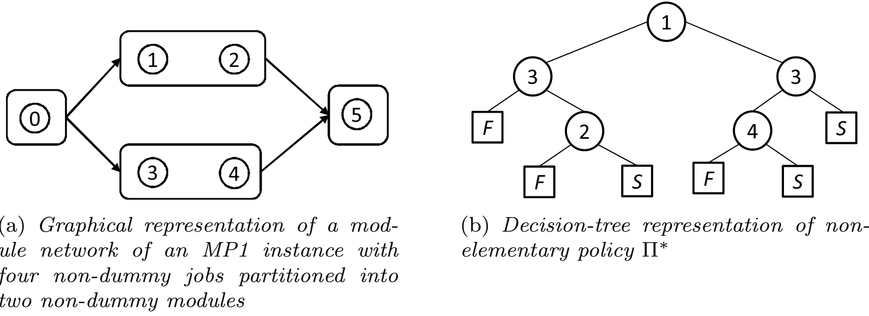
A result similar to Theorem 3.3 does not hold for arbitrary MP1 instances:

OBSERVATION 3.2. *Elementary policies are not globally optimal, i.e. there exist instances for which* $\gamma(\mathcal{C}_E) > 0$.

To verify the observation, consider the project network in Fig. 3.1(a), consisting of two parallel modules, each module containing two jobs that are not precedence-related. In Appendix A we derive a number of conditions on the parameters of this instance under which the non-elementary policy Π^* as described by Fig. 3.1(b) yields a higher expected total profit than any elementary policy. This is the case, for instance, for the values $p_1 = p_2 = p_3 = p_4 = \frac{1}{2}$, $c_1 = c_3 = 1$, $c_2 = c_4 = 3$ and $V = 13$. Full verification of the correctness of the counterexample is rather lengthy and relegated to the appendix. We even have a stronger observation:

OBSERVATION 3.3. *Elementary policies can be arbitrarily bad, i.e. there exist instances for which* $\gamma(\mathcal{C}_E) = 1$.

In Appendix B we provide an example instance that has an objective value of 1.18 for the globally optimal policy, while the empty policy is the best elementary policy (with zero objective value). Our verification of this latter result is slightly less satisfactory than for Observation 3.2, however, because it was assisted by a computer



(a) *Graphical representation of a module network of an MP1 instance with four non-dummy jobs partitioned into two non-dummy modules*

(b) *Decision-tree representation of non-elementary policy* Π^*

FIG. 3.1. *Counterexample for the claim that elementary policies would be globally optimal*

implementation of our algorithms (see Section 5) rather than by pure reasoning (as in Appendix A). We have not been able to find a counterexample of the same size as for Observation 3.2.

An MP1 instance with $A = \{(0, j) \,|\, j = 1, \ldots, m+1\} \cup \{(i, m+1) \,|\, i = 1, \ldots, m\}$ and $B_i = \emptyset$ for all $i \in M$ is called *precedence-unrelated.* Based on [3, 20] the special cases of precedence-unrelated n:n-systems and precedence-unrelated 1:n-systems are polynomially solvable. More concretely we have following two theorems:

THEOREM 3.4. *For a precedence-unrelated* n:n*-system and a job list* L *with* $\frac{c_{L(k)}}{q_{L(k)}} \leq \frac{c_{L(k+1)}}{q_{L(k+1)}}$ *for all* $k = 1, \ldots, n-1$*, elementary policy* $\Pi(\,\cdot\,; L)$ *is globally optimal unless its expected profit is less then zero, in which case it is optimal to directly abandon the project.*

THEOREM 3.5. *Consider a precedence-unrelated* 1:n*-system. Assume the non-dummy jobs* k *are indexed in non-decreasing ratio* c_k/p_k *and let* k^* *be the smallest job index such that* $c_{k^*}/p_{k^*} \geq V$*; if no such index exists put* $k^* = n+1$*. Elementary policy* $\Pi(\,\cdot\,; L)$ *is globally optimal with* $L = (1, 2, \ldots, k^*-1)$ *if* $k^* > 1$ *and* $L = \emptyset$ *if* $k^* = 1$*.*

For an n:n-system with a *series-parallel* precedence graph, a polynomial algorithm exists for solving MP1 [21]. A series-parallel graph (SPG) is defined recursively as a series or parallel composition of two SPGs; it can be verified in polynomial time whether a graph is a SPG or not [29].

3.3. Module-wise policies. A *module-wise policy* or *M-policy* is a reasonable policy that only produces schedules in which all jobs belonging to the same module are executed consecutively, so no 'jumping' between modules occurs. We denote by $\mathcal{C}_M$ the class of M-policies. Formally, we have $\Pi \in \mathcal{C}_M$ if $\Pi \in \mathcal{C}_{REA}$ and for all $\mathbf{x} \in \mathbb{B}^n$, $i \in M$, and $k \in N_i$ with $x_k = 0$ and $k = [\Pi(\mathbf{x})]_t$ for some $t < |\Pi(\mathbf{x})|$, we have $[\Pi(\mathbf{x})]_{t+1} \in N_i$.

We define *module-sequence policies* or *MS-policies* as M-policies that adhere to the same linear extension on the module order A for each possible realization. This class is represented by symbol $\mathcal{C}_{MS}$. Formally, $\Pi \in \mathcal{C}_{MS}$ if $\Pi \in \mathcal{C}_M$ and there exists a strict total order $\prec$ on A such that for all $\mathbf{x} \in \mathbb{B}^n$, $i, j \in M$, $k \in N_i$ and $l \in N_j$, with $k = [\Pi(\mathbf{x})]_s$ and $l = [\Pi(\mathbf{x})]_t$, we have $s < t \Leftrightarrow i \prec j$. Finally, the class of elementary MS-policies or EMS-policies is denoted by $\mathcal{C}_{EMS}$.

The hierarchy between the policy classes put forward in this paper is depicted in Fig. 3.2. An arrow from one class to another means that the first is a subclass of the second.

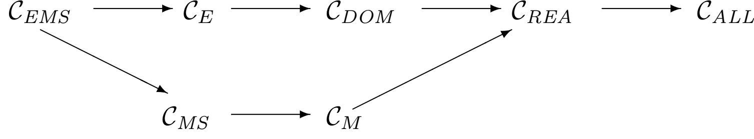
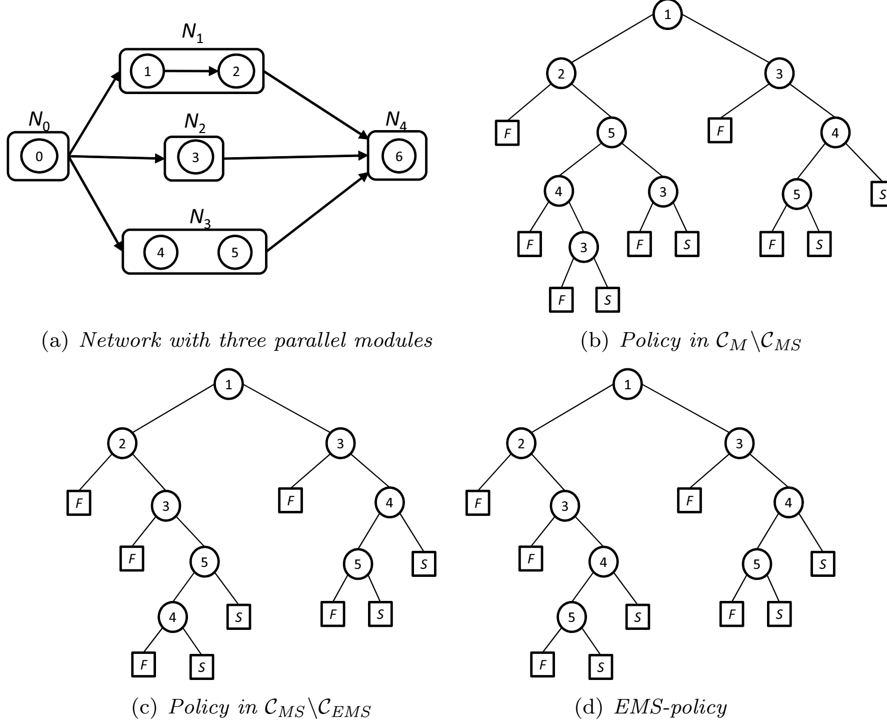


FIG. 3.2. *Hierarchy of policy classes $\mathcal{C}_{ALL}$, $\mathcal{C}_{REA}$, $\mathcal{C}_{DOM}$, $\mathcal{C}_E$, $\mathcal{C}_M$, $\mathcal{C}_{MS}$ and $\mathcal{C}_{EMS}$*



(a) *Network with three parallel modules* (b) *Policy in $\mathcal{C}_M \backslash \mathcal{C}_{MS}$*

(c) *Policy in $\mathcal{C}_{MS} \backslash \mathcal{C}_{EMS}$* (d) *EMS-policy*

FIG. 3.3. *Illustration of the difference between classes $\mathcal{C}_M$, $\mathcal{C}_{MS}$ and $\mathcal{C}_{EMS}$*

Fig. 3.3(b) shows a decision-tree representation of an M-policy for MP1 instances corresponding to the network of Fig. 3.3(a). This policy is not an MS-policy because depending on success or failure of activity 1, modules 2 and 3 are treated in different order. The policy of Fig. 3.3(c) is a member of C_{MS} with $1 \prec 2 \prec 3$, but it is not elementary because the execution order of jobs 4 and 5 of module 3 is dependent on the scenario for module 1. Fig. 3.3(d) depicts an EMS-policy defined by job list $(1, 2, 3, 4, 5)$.

THEOREM 3.6. *Dominant module-wise policies are elementary module-sequence policies and vice versa, i.e. $\mathcal{C}_{EMS} = \mathcal{C}_M \cap \mathcal{C}_{DOM}$.*

Proof. Clearly, $\mathcal{C}_{EMS} \subseteq \mathcal{C}_M \cap \mathcal{C}_{DOM}$. For a dominant M-policy Π, it suffices to find a list L of the form $(L_1, \ldots, L_m)$ with $L_i \subset N_{\sigma(i)}$ for some permutation σ of the modules, such that $\Pi = \Pi(\cdot\,; L)$. Because Π is an M-policy, all jobs of the schedule $\Pi(\mathbf{0})$ belong to the same module i_1. Set $L_1 = \Pi(\mathbf{0})$ and for ease of notation assume $L_1 = (1, \ldots, l)$. Let $\mathcal{I}_1$ be the MP1 instance derived from $\mathcal{I}$ by deleting (all jobs of) module i_1. For every $k \in L_1$, let Π^k be the policy of $\mathcal{I}_1$ such that $\Pi((e_k, \mathbf{x})) = ((1, \ldots, k), \Pi^k(\mathbf{x})), \forall \mathbf{x} \in \mathbb{B}^{n-l}$, with e_k a vector of $\mathbb{B}^l$ consisting of all zeros, except for component k. Since Π is dominant, all policies Π^k are identical and

again dominant. Denote this policy by Π_1. The same procedure can be repeatedly applied to produce a list $L_j \subset N_{i_j}$, an instance $\mathcal{I}_j$ and a policy Π_j for $j = 2, \ldots, m$. Finally, set $L = (L_1, \ldots, L_m)$ and $\sigma = (i_1, \ldots, i_m)$. By construction, Π equals the elementary policy $\Pi(\,\cdot\,; L)$. ∎

4. Properties. A problem of interest related to MP1 is that of searching for an optimal schedule when the outcome of the activities (failure or success) is known in advance. An instance of this problem corresponds to a tuple $(M, A, (N_i, B_i)_{i\in M}, \mathbf{x}, \mathbf{c}, V)$ in which $\mathbf{x}$ is a given state vector, contrary to an MP1 instance, where a vector $\mathbf{p}$ containing the success probabilities is given. The objective is to find a schedule $\mathbf{s}^*$ feasible with respect to $\mathbf{x}$ with maximal profit $f(\mathbf{x}, \mathbf{s}^*)$. We refer to this problem as DMP1 (short for 'Deterministic Modular Project scheduling on One machine'). The following result is quite straightforward:

THEOREM 4.1. *DMP1 can be solved in polynomial time.*

Proof. If there is a module in which all activities fail, $\mathbf{s}^* = \emptyset$ is optimal. Otherwise, for each non-dummy module i, define N_i' as the set of jobs $l \in N_i$ with $x_l = 1$ and $x_k = 0$ for all jobs $k \in N_i$ with $(k, l) \in B_i$. For each job $l \in N_i'$ define $\alpha_l := c_l + \sum_{k:(k,l)\in B_i} c_k$. Choose for each non-dummy module i a job $l^*(i)$ such that $l^*(i) = \arg\min_{l\in N_i'} \alpha_l$. Let $\mathbf{s}$ be a schedule containing all the $l^*(i)$ such that $(i, j) \in A$ implies $l^*(i)$ before $l^*(j)$ in $\mathbf{s}$, and insert all jobs k for which $(k, l^*(i)) \in B_i$ immediately before the corresponding $l^*(i)$ in an order that respects B_i. If $f(\mathbf{x}, \mathbf{s}) > 0$, $\mathbf{s}^* = \mathbf{s}$ is optimal, otherwise $\mathbf{s}^* = \emptyset$ is optimal. The described procedure produces an optimal schedule in polynomial time. ∎

For the general MP1 problem, on the other hand, the following complexity result holds:

THEOREM 4.2. *MP1 is NP-hard, even for the special cases of n:n-systems and 1:n-systems.*

Proof. For n:n-systems, the proof can be found in [9] and relies on a reduction from $1|prec|\sum w_i C_i$, with the implicit assumption that the payoff is sufficiently high so that all activities are executed. For 1:n-systems, when the payoff $V > \max_k c_k/p_k$, we can restrict ourselves to policies Π that schedule all the jobs in the scenario where all jobs fail, i.e. $|\Pi(\mathbf{0})| = n$. NP-hardness then follows from a reduction from MP1 for n:n-systems, see [8]. ∎

Another important issue is the hardness of computation of the profit of an elementary policy for an arbitrary MP1 instance, which is settled by the following theorem:

THEOREM 4.3. *Given an arbitrary MP1 instance $\mathcal{I}$ and an arbitrary job list L compatible with $\mathcal{I}$, the expected profit $\mathbb{E}[f(\Pi(\,\cdot\,; L))]$ for the elementary policy $\Pi(\,\cdot\,; L) \in \mathcal{C}_E$ can be computed in time linear in n.*

Proof. In general, the expected profit of a policy Π can be obtained as

$$\mathbb{E}[f(\Pi)]) = S \cdot V - \sum_{k=1}^{n} R(k) c_k, \tag{4.1}$$

with S the probability of project success and $R(k)$ the probability that job k is paid for when policy Π is applied. The probabilities $R(k)$ can be calculated recursively when the policy is elementary. Obviously, $R(k) = 0$ if $k \notin L$. For job k in position t of the list L, denote with i_t the module of job $L(t)$, i.e. $k = L(t) \in N_{i_t}$, $t \in \{1, \ldots, T\}$ with $T = |L|$. In this case we have $R(k) = (1 - \pi_{i_t}(t))(1 - Q(t))$, with $Q(t)$ the probability that the project fails before the t-th job in L is processed and $\pi_{i_t}(t)$ the

probability that success is achieved for module i_t before job $L(t)$ is executed, under the condition that the project is not abandoned before stage t. We initialize $Q(1) = 0$ and $\pi_j(1) = 0$ for all $j = 1, \dots, m$, since no non-dummy activities have been started yet before $L(1)$. We extend the definition of $Q(t)$ and $\pi_j(t)$ to $t = T + 1$, which corresponds to adding the dummy end job to the end of the list, i.e. $L(T+1) = n+1$. Because the project is successful if and only if every module is successful, we have $S = \prod_{j=1}^{m} \pi_j(T+1) = 1 - Q(T+1)$. For $1 \le j \le m$ and $2 \le t \le T+1$ we obtain the recursions

$$\pi_j(t) = \begin{cases} \pi_j(t-1) + (1 - \pi_j(t-1))\, p_{L(t-1)} & \text{if } j = i_{t-1}, \\ \pi_j(t-1) & \text{otherwise,} \end{cases} \tag{4.2}$$

$$Q(t) = \begin{cases} Q(t-1) + (1 - Q(t-1))\,(1 - \pi_{i_{t-1}}(t)) & \text{if } N_{i_{t-1}} \cap \bigcup_{s \ge t} L(s) = \emptyset, \\ Q(t-1) & \text{otherwise.} \end{cases} \tag{4.3}$$

For a given stage t, we need $Q(t)$ and $\pi_j(t)$ for only one module j to know the value of $R(L(t))$. Moreover, to obtain $\pi_j(t)$ for all modules j, we only need to adapt $\pi_j(t-1)$ for one module j. Combined with the observation that a recursive step takes only constant computation time, the theorem follows. $\square$

We conclude that the evaluation of elementary policies can be done efficiently. For general policies, by contrast, evaluation time may be exponential in the number of jobs because the number of nodes in the decision tree may be exponential in n and a compact representation is not always at hand (see Observation 2.1). Finally, we note that the values $\pi_j(t)$ can alternatively also be computed directly, as follows:

$$\pi_j(t) = 1 - \prod_{L(s) \in N_j,\, s < t} q_{L(s)}.$$

As an example, consider job list $L = (1, 3, 2, 4)$ for the network of Fig. 3.1(a). Remark that the equality $p_k + q_k p_l = 1 - q_k q_l$ holds for every pair of jobs k, l. From Table 4.1 we find $R(1) = 1$, $R(3) = 1$, $R(2) = q_1$, $R(4) = q_3(1 - q_1 q_2)$ and $S = (1 - q_1 q_2)(1 - q_3 q_4)$, leading to

$$\mathbb{E}[f(\Pi(\,\cdot\,; L))] = (1 - q_1 q_2)(1 - q_3 q_4)V - c_1 - c_3 - q_1 c_2 - q_3(1 - q_1 q_2) c_4.$$

One easily verifies that the same result is obtained via the decision-tree representation and Eq. (2.1).

Some special cases can be implemented in an alternative way:

1. Consider an n:n-system and let $L = (1, \dots, n)$ be the list determining the elementary policy $\Pi(\,\cdot\,; L)$. According to Eqn. (2.1), the expected profit $\mathbb{E}[f(\Pi(\,\cdot\,; L))]$

TABLE 4.1
Computation of values Q and π

stage t	1	2	3	4	5
$L(t)$	1	3	2	4	5
$\pi_1(t)$	0	p_1	p_1	$p_1 + q_1 p_2$	$p_1 + q_1 p_2$
$\pi_2(t)$	0	0	p_3	p_3	$p_3 + q_3 p_4$
$Q(t)$	0	0	0	$q_1 q_2$	$q_1 q_2 + (1 - q_1 q_2) q_3 q_4$

equals

$$V\prod_{k=1}^{n} p_k - c_1 - \sum_{k=2}^{n}\left(\prod_{l=1}^{k-1} p_l\right) c_k.$$

We have $O(n)$ additive operations, but the number of multiplications is $O(n^2)$. Nevertheless, we can compute the numbers $a_l := \prod_{k=1}^{l} p_k, l = 1, \ldots, n$, only using $O(n)$ multiplications because $a_l = a_{l-1}p_l$ $(l = 2, \ldots, n)$. A similar reasoning can be followed for the evaluation of elementary policies for 1:n-systems.

2. Consider an arbitrary EMS-policy Π with corresponding job list L and assume a total module order $\prec$ such that $1 \prec \cdots \prec m$. For each module $j \in \{1, \ldots, m\}$, let $\mathcal{I}_j$ be the instance of MP1 consisting of only module j and define a list L_j as the sublist of L consisting of jobs of module j only. Policy $\Pi(\,\cdot\,; L_j)$ is a well defined elementary policy for instance $\mathcal{I}_j$; remark that $\mathcal{I}_j$ is a 1:$|L_j|$-system. Now recursively define a_j to be the expected profit of $\Pi(\,\cdot\,; L_j)$ with payoff equal to a_{j+1} and initialize $a_{m+1} \equiv V$. Because of the structure of the EMS-policy Π, we have $\mathbb{E}[f(\Pi)] = a_1$ and in order to obtain a_1 we need to compute the expected profit of the elementary policies $\Pi(\,\cdot\,; L_j)$ for $j = m, m-1, \ldots, 1$. Because the time complexity of evaluating an elementary policy for a 1:$|L_j|$-system is $O(|L_j|)$ as pointed out above, the time complexity of evaluating the EMS-policy Π is $O(\sum |L_j|) = O(n)$ as $|L_j| \leq |N_j|$ and $\sum |N_j| = n$.

The final theorem of this section shows that the class of elementary policies always contains an optimal module-sequence policy.

THEOREM 4.4. *There exists an EMS-policy that is optimal in the class $\mathcal{C}_E$ of elementary policies. Consequently,* $\max\{\mathbb{E}[f(\Pi)] \mid \Pi \in \mathcal{C}_E\} = \max\{\mathbb{E}[f(\Pi)] \mid \Pi \in \mathcal{C}_{EMS}\}$.

Proof. Consider an elementary policy $\Pi(\,\cdot\,; L)$ and assume $L = (L_i, \widetilde{L}, k_i, \widehat{L})$ with $L_i \subset N_i$, $\widetilde{L} \cap N_i = \emptyset$ and $k_i \in N_i$. We show that the expected profit is not decreasing when the block L_i is moved just before job k_i, i.e. we prove $\mathbb{E}[f(\Pi(\,\cdot\,; L'))] \geq \mathbb{E}[f(\Pi(\,\cdot\,; L))]$ with $L' = (\widetilde{L}, L_i, k_i, \widehat{L})$. The expected profit can be computed using Eq. (4.1). If the elementary policy defined by job list L resp. L' is applied, we denote by S resp. S' the probability of project success, and by $R(k)$ resp. $R'(k)$ the probability of paying for job k. These probabilities can be computed with the recursive formulas (4.2)–(4.3), which results in $S' = S$, $R'(k) = (1-\alpha)R(k)$ for $k \in L_i$ and $R'(k) = R(k)$ otherwise. The constant α is the probability that the project fails due to the processing of the jobs in block $\widetilde{L}$, i.e.

$$\alpha = 1 - \prod_{\substack{j: N_j \cap \widehat{L} = \emptyset, \\ j \neq i}} \left(1 - \prod_{k \in \widetilde{L} \cap N_j} q_k\right).$$

Consequently, $R'(k) \leq R(k)$ for all k and the theorem follows. □

5. Algorithms. Two exact algorithms to solve problem MP1 are presented in this section. A dynamic-programming algorithm that finds a globally optimal policy is discussed in the first part of the section. The second part is devoted to a branch-and-bound algorithm that finds an optimal elementary policy. The section ends with a short description of two heuristic solution procedures.

5.1. Dynamic programming. We describe a backward stochastic dynamic-programming (stochastic DP, SDP) recursion to determine a globally optimal policy for MP1. The algorithm is loosely inspired by the DP in Kulkarni and Adlakha [18], which is an exact method for deriving the distribution and moments of the earliest project completion time of Markovian PERT networks. At any decision moment t, the *status* of an activity is either *idle* or *redundant*. An activity is idle when it has not yet been started and success has not been achieved yet for its module. An activity is redundant when it has been processed or when the corresponding module was successfully completed. Denote by Y the set of idle activities and by R the redundant activities. At any decision moment t, these sets constitute a partition of the job set: $N = Y(t) \cup R(t)$ and $Y(t) \cap R(t) = \emptyset$. The *state* of the system corresponds with one choice for the status for each activity. Consequently, the state of the system is completely determined by Y, because $R = N \backslash Y$.

A state $Y \in 2^N$ is called *feasible* when membership of Y implies membership of Y for all successor activities according to B^*. We let $\mathcal{S}$ represent the set of feasible states; $\mathcal{S}$ will also be called the *state space*. Formally, we have $Y \in \mathcal{S}$ if and only if $k \in Y$ implies $l \in Y$ for all $(k,l) \in B^*$. As a consequence, the dummy end job is always a member of Y.

For an MP1 instance $\mathcal{I}$ and a feasible system state Y, a smaller MP1 instance $\mathcal{I}(Y)$ is defined by removing all non-dummy redundant jobs in $N \backslash Y$ from the modular network of $\mathcal{I}$, removing empty modules, and also removing the corresponding elements from A and B_i $(i = 1, \dots, m)$. The value function $G : \mathcal{S} \mapsto \mathbb{R}_+$ of the SDP recursion maps a feasible state Y onto the expected profit of an optimal policy for MP1 instance $\mathcal{I}(Y)$. We always have $G(Y) \geq 0$ because the empty policy is included in the policy class. The maximum expected profit of the initial MP1 instance $\mathcal{I}$ equals $G(N \backslash \{0\})$, since $\mathcal{I}(N \backslash \{0\}) = \mathcal{I}$. As initial boundary condition, we set $G(\{n+1\}) = V$. For $|Y| > 1$, define the set of eligible activities $E(Y)$ as the subset of Y with all preceding jobs being redundant, i.e. $l \in E(Y)$ if and only if $l \in Y$, and $k \notin Y$ for all $(k,l) \in B^*$. The SDP relies on the following recurrence relation:

$$G(Y) = \max_{k \in E(Y)} \Big\{ 0 \,,\, p_k G(Y_{p,k}) + q_k H(Y_{q,k}) - c_k \Big\}, \quad Y \in \mathcal{S} \setminus \{\{n+1\}\} \tag{5.1}$$

$$G(\{n+1\}) = V \tag{5.2}$$

with $Y_{p,k} = Y \backslash N_{i_k}$, $Y_{q,k} = Y \backslash \{k\}$ and

$$H(Y_{q,k}) = \begin{cases} 0 & \text{if } Y \cap N_{i_k} = \{k\}, \\ G(Y_{q,k}) & \text{otherwise.} \end{cases}$$

An optimal policy Π^* can be extracted from the SDP by registering the jobs where the maxima are reached in (5.1). Concretely, if $\mathbf{x}$ is a realization, $[\Pi^*(\mathbf{x})]_0$ is a job where the maximum is reached in (5.1) for state $Y^0 = N \backslash \{0\}$. Denote this job by k_1 and its module by i_1. If $x_{k_1} = 1$ we move to state $Y_p^1 = Y^0 \backslash N_{i_1}$ and $[\Pi^*(\mathbf{x})]_1$ is a job where the recurrence relation reaches its maximum for state Y_p^1. If $x_{k_1} = 0$ and $|N_{i_1}| > 1$ we move to state $Y_q^1 = Y^0 \backslash \{k_1\}$ and $[\Pi^*(\mathbf{x})]_1$ is a job where the recurrence relation reaches its maximum for state Y_q^1. If $|N_{i_1}| = 1$ the schedule ends ($|\Pi^*(\mathbf{x})| = 1$). Proceeding in this way, we can construct schedule $\Pi^*(\mathbf{x})$. Note that the project is abandoned when the maximum over all eligible jobs is negative, which coincides with a value function of zero in (5.1).

THEOREM 5.1. *Recurrence relation (5.1)–(5.2) finds a globally optimal policy.*

Proof. Let $\mathcal{I}$ be an arbitrary MP1 instance an let Π be a reasonable policy with decision-tree representation T and expected profit π. Denote the root node of T by k and the module containing job k by i. If job k is successful, we consider the instance $\mathcal{I}_p$ obtained by removing module i together with adjacent inter-modular precedence constraints from the modular network of instance $\mathcal{I}$. The subtree T_p of T emerging from the success edge of node k of T is a decision-tree representation of a policy Π_p for instance $\mathcal{I}_p$ with expected profit π_p. If job k fails, we consider the instance $\mathcal{I}_q$ obtained by removing job k from module i. Let T_q be the subtree emerging from the failure edge of node k of T. If k is the only job in module i then T_q consists of only one (leaf) node, labeled F. Otherwise, T_q is a tree representation of a policy Π_q for instance I_q with expected profit π_q. The expected profit of Π can be expressed as $\pi = p_k\pi_p + q_k\bar{\pi}_q - c_k$ with $\bar{\pi}_q = 0$ if $N_i = \{k\}$ and $\bar{\pi}_q = \pi_q$ otherwise. By the principle of optimality, it follows that policy Π is optimal for instance $\mathcal{I}$ if the policies Π_p and Π_q are optimal for the smaller instances $\mathcal{I}_p$ and $\mathcal{I}_q$, respectively. □

THEOREM 5.2. *The optimal policy generated by recurrence relation (5.1)–(5.2) is a dominant policy.*

Proof. Let Π^* be a policy generated by the SDP as explained above. In order to show that Π^* is dominant, we choose two equivalent nodes k_1 and k_2 of the decision tree T^* of Π^*. The decisions made from these nodes are identical because they correspond to one and the same state in the recurrence relation. The state $Y(k)$ corresponding to a node k of T^* is given by the complement of $C_0(k) \cup \{N_i \mid N_i \cap C_1(k) \neq \emptyset\}$. From (E2)–(E3) it follows that $Y(k_1) = Y(k_2)$. □

For efficient implementation of the DP recursion, we construct a total order relation $\leq_{\mathcal{S}}$ on the state space $\mathcal{S}$ that determines the order in which the states are evaluated. The relation is such that the value-function values on the right-hand side of (5.1) that are input to the computation of $G(Y)$ have already been computed beforehand. We observe that the corresponding states always have a cardinality strictly less than $|Y|$. This implies that any ordering $\leq_{\mathcal{S}}$ respecting $Y_1 \leq_{\mathcal{S}} Y_2 \Leftrightarrow |Y_1| \leq |Y_2|$ satisfies our needs. To find a suitable order, the approach taken in [6] is followed by partitioning the state space according to the so-called rank of inclusion-maximal antichains of the induced network. An inclusion-maximal antichain is also referred to as a *uniformly directed cut* (UDC). Denote the set of UDCs by $\mathcal{U}$. The UDCs of the MP1 instance depicted in Fig. 2.1 are $U_0 = \{0\}$, $U_1 = \{1,3\}$, $U_2 = \{2,3\}$, $U_3 = \{4,5\}$ and $U_4 = \{6\}$. Let U be a UDC and denote by $N(U)$ the subset of N containing the successor jobs of U, i.e.

$$N(U) = \{l \in N \mid \exists\, k \in U : (k,l) \in B^*\}.$$

The *rank* r of a UDC counts the number of predecessor activities in the induced network, i.e. $r(U) = |N \backslash (N(U) \cup U)|$. For the example we have $N(U_0) = \{1,2,3,4,5,6\}$, $N(U_1) = \{2,4,5,6\}$, $N(U_2) = \{4,5,6\}$, $N(U_3) = \{6\}$, $N(U_4) = \emptyset$ and $r(U_0) = 0$, $r(U_1) = 1$, $r(U_2) = 2$, $r(U_3) = 4$, $r(U_4) = 6$.

A set of subsets of N is associated with each $U \in \mathcal{U}$ and is denoted by $\sigma(U)$. A set $Y \subset N$ belongs to $\sigma(U)$ if we can write $Y = N(U) \cup U'$ with U' a non-empty subset of U such that $E(Y) \subset U$. Note that $N(U) \cup U$ always belongs to $\sigma(U)$ since $E(N(U) \cup U) = U$. In the following theorem, we summarize results obtained in [6]:

THEOREM 5.3. *Let U, U' be UDCs and let $Y \in \sigma(U)$, $Y' \in \sigma(U')$. We have*

(i) *$\{\sigma(U) \mid U \in \mathcal{U}\}$ is a partition of $\mathcal{S}$.*

(ii) *Assume Y is in the left-hand side, and Y' is in the right-hand side of* (5.1). *If $U' \neq U$ then $r(U') > r(U)$.*

(iii) *If* $r(U') > r(U)$ *then* $|Y'| < |Y|$.

From Theorem 5.3 we infer an appropriate total order on the state space $\mathcal{S}$ by enumerating the parts $\sigma(U)$ of the partition in non-increasing rank order and the states in a given set $\sigma(U)$ in non-decreasing cardinality order. For the example, this leads to the following steps in computing the recurrence relation:

1. $\sigma(U_4) = \{Y_0\}$ with $Y_0 = \{6\}$; $G(Y_0) = V$.
2. $\sigma(U_3) = \{Y_1, Y_2, Y_3\}$ with $Y_1 = \{4, 6\}$, $Y_2 = \{5, 6\}$, $Y_3 = \{4, 5, 6\}$;
   $G(Y_1) = \max\{0, p_4 G(Y_0) - c_4\}$,
   $G(Y_2) = \max\{0, p_5 G(Y_0) - c_5\}$,
   $G(Y_3) = \max\{0, p_4 G(Y_0) + q_4 G(Y_2) - c_4, p_5 G(Y_0) + q_5 G(Y_1) - c_5\}$.
3. $\sigma(U_2) = \{Y_4, Y_5, Y_6\}$ with $Y_4 = \{3, 4, 5, 6\}$, $Y_5 = \{2, 4, 5, 6\}$, and $Y_6 = \{2, 3, 4, 5, 6\}$;
   $G(Y_4) = \max\{0, p_3 G(Y_3) - c_3\}$,
   $G(Y_5) = \max\{0, p_2 G(Y_3) - c_2\}$,
   $G(Y_6) = \max\{0, p_2 G(Y_4) - c_2, p_3 G(Y_5) - c_3\}$.
4. $\sigma(U_1) = \{Y_7, Y_8\}$ with $Y_7 = \{1, 2, 4, 5, 6\}$, $Y_8 = \{1, 2, 3, 4, 5, 6\}$;
   $G(Y_7) = \max\{0, p_1 G(Y_3) + q_1 G(Y_5) - c_1\}$,
   $G(Y_8) = \max\{0, p_1 G(Y_4) + q_1 G(Y_6) - c_1, p_3 G(Y_7) - c_3\}$.

Note that $Y_6 \notin \sigma(U_1)$ since $E(Y_6) \not\subset U_1$. The maximum expected profit is $G(Y_8)$.

5.2. Branch and bound. In this section, we develop an algorithm to optimize over the class $\mathcal{C}_{EMS}$ of elementary module-sequence policies. Recall that an optimal policy in this class is also optimal in the superclass $\mathcal{C}_E$ (Theorem 4.4). Adapting the DP described in the previous section does not seem to be straightforward. To see this, reconsider the argument made in the proof of Theorem 5.1: a policy Π of an instance $\mathcal{I}$ can be decomposed in two policies Π_p resp. Π_q of smaller instances $\mathcal{I}_p$ resp. $\mathcal{I}_q$. Unfortunately, policy Π is not necessarily elementary even if policies Π_p and Π_q are elementary, making the recurrence relation (5.1) invalid for elementary policies. To further illustrate this, consider the instance and policy Π^* depicted in Fig. 3.1. This policy is not elementary despite the fact that Π^*_p and Π^*_q are elementary (and module-sequence) with corresponding order lists $(3, 4)$ and $(3, 2)$ respectively. The fact that an elementary policy is representable by a job list makes the solution space $\mathcal{C}_E$ very suitable for implicit enumeration by means of a branch-and-bound (B&B) algorithm, however, and this section is devoted to the development of such an algorithm.

5.2.1. Branching strategy. We devise a branching tree that enumerates the elements of class $\mathcal{C}_{EMS}$. Fig. 5.1 shows the branching tree of an MP1 instance with the modular network of Fig. 2.1(a). A node η in the tree is labeled with a job $k(\eta)$ and represents a partial order list $PL(\eta)$, determined by the node labels of the unique path in the search tree starting from the root node and ending in the node labeled $k(\eta)$. The root node at level 0 is labeled with the dummy start job 0 and every leaf node is labeled with the dummy end job $n + 1$. Denote by $M_u(\eta)$ resp. $M_s(\eta)$ the set of unstarted resp. already started modules, i.e. $M_u(\eta) = \{i \in \{1, \ldots, m\} \mid N_i \cap PL(\eta) = \emptyset\}$ and $M_s(\eta) = \{1, \ldots, m\} \backslash M_u(\eta)$. Consider a node η labeled with a job k inside a module i. A node η' labeled with a job k' inside a module i' is a child node of node η if conditions (P1)–(P3) below hold.

(P1) $k' \notin PL(\eta)$.
(P2) $(PL(\eta), k')$ is a compatible partial list (see Section 3.2).
(P3) $i' \notin M_s(\eta) \backslash \{i\}$.

The collection of all the leaf nodes corresponds to the collection of all compatible job lists for which jobs belonging to the same module are consecutive, and thus to class

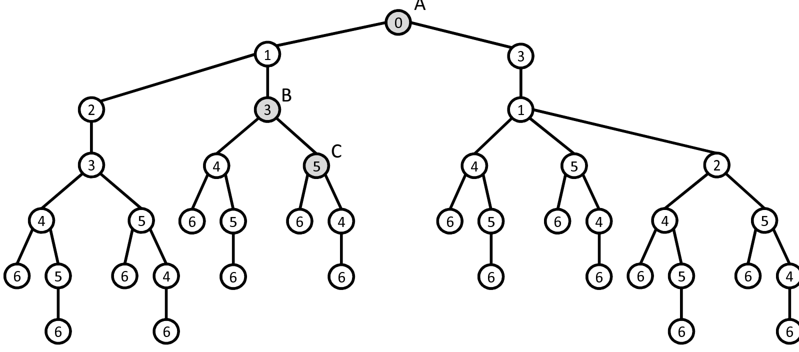


Fig. 5.1. *Branching tree for an MP1 instance with modular network as depicted in Fig. 2.1(a)*

$\mathcal{C}_{EMS}$. Note that removal of condition (P3) defines a branching tree for class $\mathcal{C}_E$, where the consecutive execution of jobs from the same module is not longer enforced. For the example network, this would mean that node B of the branching tree depicted in Fig. 5.1 would receive a third child, labeled with job 2.

5.2.2. Upper bound. In our upper-bound computation we assume that the project is executed, i.e. we exclude the empty policy from the solution space. A negative upper bound at a given node is automatically pruned by the zero global lower bound (see Section 5.2.3). Consequently, at least one job in each module needs to processed (condition (C1)), this to render the project potentially successful. Our upper bound $\bar{z}(\eta)$ at node η consists of a positive part $\overline{EP}(\eta)$ that overestimates the expected payoff and a negative part $\underline{EC}(\eta)$ that underestimates the expected cost, and exploits the knowledge of the thus-far constructed partial list $PL(\eta)$.

The expected payoff increases when more jobs are included in the list. Consequently, an overestimation of the expected payoff results from scheduling all jobs in all modules, unless adding a job to the list leads to an invalid order list. Denote by $M_1(\eta)$ the set of modules in which no new activities can be started due to condition (P3), i.e. $M_1(\eta) = M_s(\eta)\backslash\{i\}$, and gather the remaining modules in $M_2(\eta) = \{1, \ldots, m\}\backslash M_1(\eta)$. The first term of the upper bound equals

$$\overline{EP}(\eta) = \left[\prod_{i\in M_1(\eta)} \Big(1 - \prod_{k\in N_i\cap PL(\eta)} q_k\Big)\right] \cdot \left[\prod_{i\in M_2(\eta)} \Big(1 - \prod_{k\in N_i} q_k\Big)\right] \cdot V. \tag{5.3}$$

Note that $M_1(\eta)$ also contains modules containing a job for which addition to the partial order list leads to a violation of condition (C4). The bound described by Eq. (5.3) would remain valid if we optimized over class $\mathcal{C}_E$, but the set of forbidden modules $M_1(\eta)$ would be smaller and given by $M_1(\eta) = \{i \in \{1, \ldots, m\} \mid N_j \cap PL(\eta) \neq \emptyset$ for some $(i, j) \in A\}$. Consequently, the bound is stronger for class $\mathcal{C}_{EMS}$ than for class $\mathcal{C}_E$.

The expected cost increases when more jobs are in the list, so an underestimation includes only a single job for each module that has no job in the partial list. The probability of paying for such a job decreases when more modules are scheduled before it, because they all need to be successful. To write down a closed formula, define N_i' as the subset of N_i holding the first jobs of a module: $N_i' = \{l \in N_i \mid \nexists k \in N_i :$

$(k,l) \in B_i\}$. The second term of the upper bound can be written as

$$\underline{EC}(\eta) = \sum_{i \in M_s(\eta)} \left[\sum_{k \in N_i \cap PL(\eta)} R(k) c_k \right] + \sum_{i \in M_u(\eta)} \left[\left(\prod_{j:(i,j) \notin A} P(j) \right) \min_{k \in N_i'} c_k \right] \tag{5.4}$$

with

$$P(j) = \begin{cases} \min_{k \in N_j'} p_k & \text{if } j \in M_u(\eta), \\ 1 - \prod_{k \in N_j \cap PL(\eta)} q_k & \text{if } j \in M_s(\eta). \end{cases} \tag{5.5}$$

The quantity $R(k)$ is the probability of paying for job k when $PL(\eta)$ is applied (in line with the definition in Section 4) under the condition that no further activities in the module of job k are processed apart from those in $PL(\eta)$. These values can be computed using the recursion discussed in the proof of Theorem 4.3 applied to the instance obtained by removing jobs that are not in $PL(\eta)$.

The upper bound at the root node η_0 is a global upper bound and is given by

$$\bar{z}(\eta_0) = \prod_{i=1}^{n} \left(1 - \prod_{k \in N_i} q_k \right) V - \sum_{i=1}^{m} \left(\prod_{j:(i,j) \notin A} \min_{k \in N_j'} p_k \right) \min_{l \in N_i'} c_l. \tag{5.6}$$

The computation of the bound is illustrated for the nodes A, B and C of the branching tree depicted in Fig. 5.1. The global upper bound at the root node A is

$$\bar{z}(A) = (1 - q_1 q_2) p_3 (1 - q_4 q_5) V - (p_3 c_1 + p_1 c_3 + p_1 p_3 \min\{c_4, c_5\}).$$

For node B at level 2, we have a partial list $PL(B) = (0, 1, 3)$ and module sets $M_s(B) = \{1, 2\}$, $M_u(B) = \{3\}$, $M_1(B) = \{1\}$ and $M_2(B) = \{2, 3\}$. Since $R(1) = 1$ and $R(3) = p_1$, the local upper bound in B is

$$\bar{z}(B) = (1 - q_1) p_3 (1 - q_4 q_5) V - (c_1 + p_1 c_3 + p_1 p_3 \min\{c_4, c_5\}).$$

Remark that the bound would weaken if we optimized over $\mathcal{C}_E$: then $M_1(B) = \emptyset$ and $M_2(B) = \{1, 2, 3\}$, and the factor $1 - q_1$ in $\overline{EP}(B)$ would change to $1 - q_1 q_2$. Finally, in node C at level 3, we have $PL(C) = (0, 1, 3, 5)$, $M_s(C) = \{1, 2, 3\}$, $M_u(C) = \emptyset$, $M_1(C) = \{1, 2\}$, $M_2(C) = \{3\}$, $R(1) = 1$, $R(3) = p_1$ and $R(5) = 1 - (q_1 + p_1 q_3) = p_1 p_3$, resulting in a local upper bound

$$\bar{z}(C) = p_1 p_3 (1 - q_4 q_5) V - (c_1 + p_1 c_3 + p_1 p_3 c_5).$$

5.2.3. Global lower bound. Since the empty policy belongs to $\mathcal{C}_{EMS}$, an initial global lower bound is $\underline{z} = 0$. Heuristic procedures can be used to strengthen this bound (see Section 5.3). In each leaf node η, the local upper bound described in the previous subsection is also exactly the expected profit of the elementary policy defined by the (full) order list $PL(\eta)$ and hence constitutes a global lower bound ($\underline{z} = \bar{z}(\eta)$).

5.3. Heuristic procedures. Inspired by the results of Section 3.2, we propose a heuristic algorithm for general $n{:}n$-systems (Algorithm 2). The algorithm produces an elementary policy $\Pi(\,\cdot\,; L)$ that can serve as an approximate solution to the global optimum and functions as a subroutine in Algorithm 3, which provides an elementary policy for general MP1 instances.

Algorithm 2 Heuristic for $n{:}n$-systems

1: $L = \emptyset$; E = set of non-dummy jobs without non-trivial predecessors
2: **while** $E \neq \emptyset$ **do**
3: Choose a job k^* from E such that $\forall k \in E \backslash \{k^*\}$ we have $c_{k^*}/q_{k^*} \leq c_k/q_k$
4: Add job k^* to the end of L
5: Update $E = E \backslash \{k^*\} \cup$ {jobs in $N \backslash (E \cup L \cup \{n+1\})$ with all predecessors in L}
6: **end while**
7: Return L

Algorithm 3 Heuristic for general instances

1: **for** each module i **do**
2: Let $N_i' = \{k \in N_i \mid \nexists l \in N_i : (l, k) \in B_i\}$
3: Let $k_i = \arg\min\{c_l/p_l \mid l \in N_i'\}$
4: **end for**
5: Apply Algorithm 2 to the $m{:}m$-system obtained by removing all jobs from module i except for job k_i

6. Computational experiments. We have implemented the algorithms in C++ using Microsoft Visual Studio 2010. The experiments were run on a Dell Desktop E8500 Optiplex 760 with an Intel Core 2 Duo processor with clock rate of 3.16 GHz and 3.21 GB of RAM, equipped with Windows XP Professional Version 2002 Service Pack 3. All CPU times are expressed in seconds.

We have generated two data sets, which are available on-line[1]. The first set exclusively consists of $n{:}n$-systems, whereas the second contains general MP1 instances, possibly holding more than one activity per module. The $n{:}n$-instances are created similarly to [9], using the random network generator RanGen [10] to build directed graphs for a given value of n and of the density of the network, which is measured by the *order strength OS*. The order strength is defined as the number of precedence-related activity pairs divided by the maximum possible number of such pairs, which is $\binom{n}{2} = n(n-1)/2$. We create 10 instances for each of the combinations of values for the parameters n and OS, with $n \in \{10k \mid k = 1, 2, \dots\}$ and $OS \in \{0.4, 0.6, 0.8\}$. The second data set contains instances for the same combinations of values for n and for the OS of the induced network as for the $n{:}n$-instances, although the OS value is now only approximative, and not necessarily identical for all the instances. Full details on the data generation are provided in Appendix C.

Below, we include a discussion of some implementation issues for the DP algorithm (Section 6.1), followed by a presentation of the computational results (Section 6.2).

6.1. Implementation issues for the DP algorithm. The SDP recursion uses a bitwise representation for a set of activities. A subset of n jobs is represented by an array of size $\lceil n/32 \rceil$ containing 32-bit integers. A job k is an element of the subset if and only if the k-th bit of the integer array is a 1-bit. This representation allows to efficiently manage memory. Moreover, binary set operations such as the union, intersection and complement of two sets and adding an element to or removing an element from a set can also be implemented more efficiently.

To generate the UDCs, we observe that the set $\mathcal{U}$ of inclusion-maximal antichains of the induced network coincides with the set of maximal independent sets of the

[1] Available at http://www.econ.kuleuven.be/public/NDBAC96/MP1_instances.htm

undirected graph with node set N that contains all edges $\{k, l\}$ with either $(k, l) \in B^*$ or $(l, k) \in B^*$. The algorithm presented in [13] is implemented to generate all maximal independent sets in lexicographic order, with only polynomial delay between the output of two successive independent sets. During the generation process, the rank of the UDC is computed and UDCs are grouped based on their rank.

The main issue in the implementation of the SDP recursion is the memory management: memory and not computation time will turn out to be the bottleneck. For instances with $n \leq 32$ it is possible to represent the value function with an array of size 2^n, in which each index of the array is one of the 2^n possible states. In practice, however, this implementation is only suitable for solving very small instances. Storing an array of size 2^{30}, for example, would require too much RAM even for an average recent computer. Fortunately, $|\mathcal{S}|$ is often substantially less than 2^n, especially for dense precedence networks (both intermodular as well as intramodular), which creates opportunities for handling the memory more efficiently. Note that $|\mathcal{S}|$ can be computed exactly using the state-space partition in UDCs (Theorem 5.3), i.e. $|\mathcal{S}| = \sum_{U \in \mathcal{U}} |\sigma(U)|$. We use a *hash table* [4, 15, 26] to tackle these memory issues, which is a data structure enabling fast storage and lookup of the data elements. Elements of a hash table are pairs consisting of a *key* and a *value*. In our SDP, a key-value pair takes the form $(Y, G(Y))$ with $Y \in \mathcal{S}$ a feasible state and $G(Y)$ the value function at state Y. A *hash function* maps a key to one of the possible table entries and determines the location to store and retrieve the associated value. The function $g(Y) := h(Y) \bmod L$ maps a key Y to an integer hash value in the range $[0, L-1]$, with L the length of the hash table. The integer $h(Y)$ undergoes a modulo-L operation to guarantee a valid index of the hash table. Below, we will use the term hash function both for g and for h. When two different keys are mapped to the same table entry, a *collision* occurs, which can be resolved in several ways. More details on the data structures used are provided in Appendix D.

A key Y is represented by an integer array $Y = (y_0, \ldots, y_l)$ of size $l + 1 = \lceil n/32 \rceil$. A simple function of the form $\sum_{i=0}^{l} y_i$ results in equal hash values for different permutations of the components of Y, so a multiplication with a position-dependent power of a large prime number c is usually performed, i.e. $h_0(Y) := \sum_{i=0}^{l} y_i c^i$. This is a common hash function mainly used for string keys [15]. Fig. 6.1 shows the performance of h_0 on an n:n-instance with $n = 60$ and $OS = 0.4$ (filename *s_n60_os4_1*), dependent on the table size L (scaled by the total number of elements in the list: the horizontal axis shows $L/|\mathcal{S}|$). The state space for the example instance is quite large, approximatively $|\mathcal{S}| = 10\,924\,600$. The left plot shows the average number of calls needed to retrieve an element from the hash table as a function of the table size, the right plot depicts the CPU time. Typically, L is chosen as a large prime P or as a power of two. By choosing the latter, i.e. $L = 2^k$, the slightly more expensive modulo operator in $g(Y) = h(Y) \bmod L$ can be avoided since the result of the modulo operator coincides with the first k bits of $h(Y)$. Both options are explored in Fig. 6.1. We find that a prime number for the table size (diamond marks, light-grey curve) is far better than 2^k (square, dark grey), both on CPU time as well as with respect to the average number of calls. One reason for this is probably that h_0 by itself is actually a rather poor hash function, and that simply dropping the $(32 - k)$ most significant bits results in too many collisions. The modulo operator with a prime number, on the other hand, if not too close to a power of two, will improve the randomness and is in fact a well-known hash function for integer keys, known under the name *division method* [4]. The prime number should be as far as possible from a power of two to

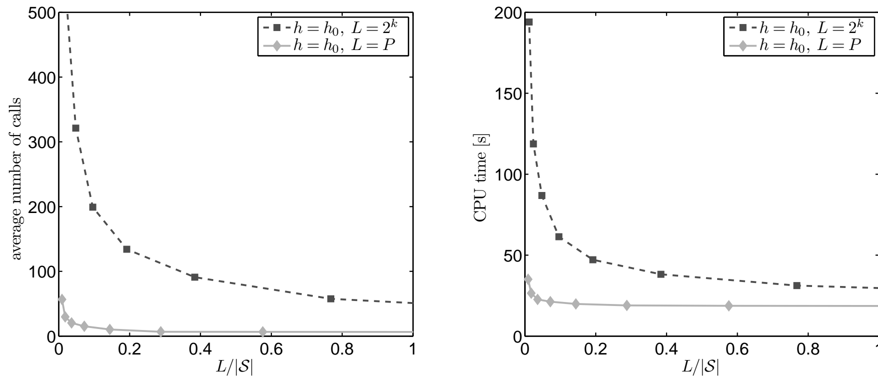


FIG. 6.1. *Influence of the table size on the performance of the hash table*

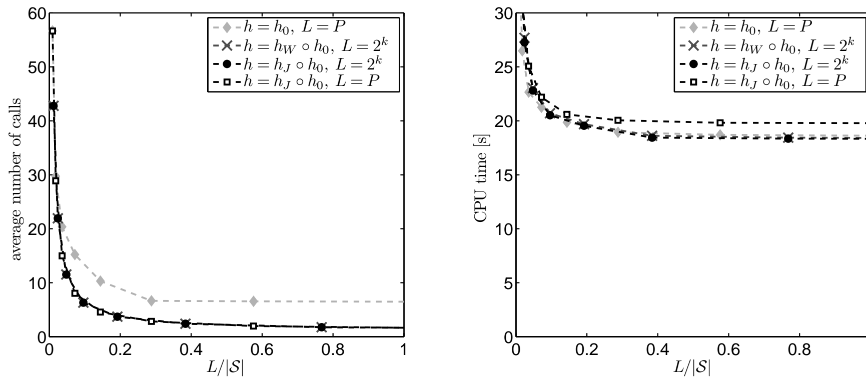


FIG. 6.2. *Effect of the composition of h_0 with an integer hash function*

avoid clustering [4, 15]. In our implementation, we take a prime number close to the middle of the intervals $(2^k, 2^{k+1})$.

The quality of the hash function h_0 can be significantly improved when it is followed by a good integer hash function. Integer hash functions expect an integer key and manipulate the bits of the key to produce a randomized integer where the probability of a 1-bit or a 0-bit is equally likely for all the 32 bits of the integer. Good integer hash functions strive for a change in as many bits as possible when two different keys only differ in a single or a few bits. From the literature, the best-performing hash functions are Wang's "32-bit Mix Function" h_W and Jenkins' 32-bit integer hash function h_J [31]. The left graph of Fig. 6.2 shows that the quality of hash function h_0 (with prime table sizes) improves after applying h_W or h_J (with power-of-two table sizes), or h_J (with prime table sizes). The average number of calls to retrieve an element can be reduced from about 6.5 to 1.7 for a sufficiently large table size. The right graph indicates that, for prime table sizes, the efficiency gain for h_0 by composing with function h_J is fully undone by the increase in complexity of the hash function. For power-of-two sizes, however (where we avoid the expensive modulo operator), the composed functions are better and and even have a minor advantage in CPU time compared to prime table sizes. Function h_J is slightly better than h_W, both in quality (number of calls) and in efficiency, but the difference is very small.

TABLE 6.1
Comparison of average CPU times and number of solved instances (out of 10) between DP and B&B for the data set containing only n:n-instances. The CPU time is averaged only for the solved instances.

	$OS = 0.8$				$OS = 0.6$				$OS = 0.4$			
	DP		B&B		DP		B&B		DP		B&B	
n	CPU	#	CPU	#	CPU	#	CPU	#	CPU	#	CPU	#
10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10
20	0.00	10	0.00	10	0.00	10	0.01	10	0.00	10	0.04	10
30	0.00	10	0.08	10	0.00	10	2.23	10	0.02	10	35.51	10
40	0.00	10	2.74	10	0.01	10	349.52	10	0.23	10	314.03	3
50	0.00	10	77.82	10	0.04	10	1.04	3	2.39	10	1.23	2
60	0.00	10	506.28	9	0.21	10	0.17	2	30.73	10	161.03	5
70	0.01	10	5.70	4	1.03	10	0.06	2		0	0.81	1
80	0.02	10	0.05	1	3.99	10	30.80	3		0	15.98	1
90	0.05	10		0	15.35	10		0		0		0
100	0.09	10		0	56.62	8		0		0		0
110	0.21	10		0		0		0		0		0
120	0.46	10		0		0		0		0		0

We have obtained similar findings for the other instances. Based on this analysis, we have opted for hash function $h_J \circ h_0$ as final implementation for solving the instances of the two data sets. Fig. 6.2 suggests that a table size at least half the size of the state space is sufficient, because the improvement for larger sizes is only minor. The size of the hash table is set as the largest power of two smaller than $|\mathcal{S}|$, and is thus between $|\mathcal{S}|/2$ and $|\mathcal{S}|$. Note that the average number of calls is monotonously decreasing when L goes up (the quality of the hash table improves with increasing L), but the same need not be true for the CPU time: the efficiency of hash tables can decrease when L is overly large. The largest table size that we have been able to implement is $L = 2^{28}$, which is about 25 times the size of the state space of the test instance. Compared to a good choice for the table size, which is $L = 2^{23}$, the average number of calls drops to 1.1, but the CPU time increases by more than a second. A possible explanation for this might be the increase in cache misses, since blocks of memory will contain more redundant lines holding unused table entries [22].

6.2. Computational results. Table 6.1 reports the average CPU time of the algorithms for the first data set, which contains only n:n-systems. The number of instances solved until guaranteed optimality (out of 10) is displayed together with the average CPU time. The DP solves instances with 120 jobs and high OS (0.8) in less than a half second. When the order strength decreases, the state space grows (see Table 6.2(a)) and the instances become harder to solve. For $OS = 0.6$, we can solve instances with up to 100 jobs and more than 23 million states in less than one minute on average. Larger instances cannot be solved because the computer runs out of memory. If OS is further lowered to 0.4, the DP solves instances with $n = 60$ in a half minute. For larger networks, we again encounter memory problems. We conclude that the instances become harder when the density of the networks decreases, and that overall the DP solves quite large instances (with a very large state space) in little time.

For the first data set, the B&B algorithm finds optimal elementary policies with the same objective values as the DP (Theorem 3.3). A time limit of 30 minutes is imposed when running the B&B algorithm. The results are also summarized in

Table 6.2
Additional indicators of the computational effort of the two algorithms for each instance group specified by n and OS of the first data set (n:n-instances)

(a) Average size of the state space in the DP (average only over the solved instances)

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	22	40	91
20	88	316	1557
30	254	2044	16284
40	689	9650	176200
50	1460	41983	1421452
60	3657	162351	14074728
70	8092	644306	
80	15569	2176367	
90	31474	7335005	
100	61867	23517689	
110	131360		
120	280685		

(b) Average number of visited nodes in the B&B tree (average over all 10 instances per cell)

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	11	18	40
20	57	417	1914
30	2005	27834	511554
40	34514	3337916	11596365
50	558017	6215383	5608505
60	2483434	4673974	5258784
70	2621293	7446097	5292828
80	2506070	3315457	5248734

Table 6.3
Comparison of average CPU times and number of solved instances (out of 10) between DP and B&B for the second data set (general MP1 instances). The CPU time is averaged only for the solved instances.

	$OS = 0.8$				$OS = 0.6$				$OS = 0.4$			
	DP		B&B		DP		B&B		DP		B&B	
n	CPU	#	CPU	#	CPU	#	CPU	#	CPU	#	CPU	#
10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10	0.00	10
20	0.00	10	0.03	10	0.00	10	0.71	10	0.00	10	24.04	10
30	0.00	10	18.22	10	0.00	10	126.97	5	0.01	10	254.31	3
40	0.00	10	627.54	5	0.02	10	10.35	2	1.23	10		0
50	0.00	10		0	0.04	10		0	5.95	9		0
60	0.02	10		0	1.54	10		0	23.6	8		0
70	0.06	10		0	3.27	10		0	41.9	6		0
80	0.07	10		0	16.65	10		0	403.9	4		0
90	0.19	10		0	24.6	9		0	2041.1	2		0
100	0.21	10		0	21.1	4		0		0		0
110	1.09	10		0	160.3	4		0		0		0
120	1.48	10		0	363.6	4		0		0		0

Table 6.1. We conclude that the B&B is far slower than the DP; each unsolved instance is interrupted due to the time limit, there are no memory problems. This also means more instances will likely be solved by simply increasing the time limit. For instances with $OS = 0.8$, the algorithm solves to optimality 64 instances (out of 80) with at most 80 activities within the time limit. For order strengths of 0.6 and 0.4, this number reduces to 50 and 42, respectively. Table 6.2(b) shows the average number of nodes (rounded to the nearest integer) explored by the B&B algorithm. This number grows very rapidly up to the value of n for which the number of instances solved is less than 50%, after which the number of nodes stabilizes. This might be explained by the fact that for most instances, the time limit is reached and the number of nodes investigated so far by the B&B algorithm is significantly smaller than the number of nodes that would be investigated if the algorithm was run to completion.

Table 6.4

Additional indicators of the computational effort of the two algorithms for each instance group specified by n and OS of the second data set (general MP1 instances)

(a) Average size of the state space in the DP (average only over the solved instances)

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	21	44	72.4
20	80	325	1093.8
30	263	2905	8940.5
40	790	13452	622098
50	2587	19032	2486256
60	3955	736142	1774351
70	32972	1481591	3922089
80	18416	5792564	17851239
90	91720	7234182	13352239
100	116229	5457474	
110	494835	19013690	
120	595798	13552638	

(b) Average number of visited nodes in the B&B tree (average over all 10 instances per cell)

n	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	42	120	200
20	2461	53015	745391
30	854607	38286295	61974755
40	34113658	36044811	

In Table 6.3, the run times and the number of solved instances are shown for the second data set, which contains general MP1 instances. The DP is again by far more efficient than the B&B. Compared to the first data set, the difficulty of the instances for a fixed value of n and OS is significantly more heterogenous (see also Table 6.4). One possible cause might be that the order strength of the induced network does not accurately capture the complexity of an instance anymore. The intermodular precedence constraints may only have a small influence on the order strength of the induced network, but they can have an important impact on the size of the state space and thus on the difficulty of the instance. If multiple modules contain only few precedence constraints, the state space grows dramatically.

The average relative optimality gap of the elementary policies found by the B&B algorithm is 1.28% for the 85 instances that are solved until optimality within the time limit. Only 11 instances out of 85 have an optimality gap different from 0%, and exactly one of those 11 instances reaches an optimality gap of 100% (this is the instance reproduced in Appendix B).

7. Summary and conclusions. This article has studied a model for scheduling R&D projects such as to maximize the expected profit when activities have a possibility of failure. The model extends earlier work by introducing a two-layered network structure, where activities are grouped in modules such that individual activity default does not necessarily imply an overall project failure. Activities have deterministic durations, a fixed processing cost, and are scheduled on a single machine. In this setting, a solution is a policy, which can be naturally described by a binary decision tree.

Multiple policy classes are proposed and the relationship between the classes is examined. Elementary policies are determined by a list of activities and have a compact representation, this in contrast to more general classes of policies. We show that elementary policies are globally optimal for a number of specific network structures, but not in general. We also prove that it is sufficient to execute the jobs module by module when the solution space is restricted to elementary policies. Although the general scheduling problem is NP-hard, some special cases are shown

to be polynomially solvable.

A backward stochastic dynamic-programming recursion is developed to produce globally optimal policies. The algorithm is quite efficient and can solve large instances with more than 100 jobs, with the performance dependent on the density of the network. The bottleneck of the algorithm turns out to be memory rather than computation time. Because an adaptation of the dynamic-programming recursion to elementary policies is not straightforward, a branch-and-bound algorithm is proposed to find an optimal elementary policy. Notwithstanding the smaller solution space, in our implementations the optimization over elementary policies is significantly more time-consuming than the dynamic program, making the computation time the bottleneck of this algorithm. On average, the quality of an optimal elementary policy seems to be quite close to a globally optimal policy. There are instances nevertheless where a project would not be executed (has a zero objective value) when execution is restricted to elementary policies, whereas a globally optimal policy with a strictly positive objective value exists.

For future research, a valid research question is whether a more efficient solution procedure is achievable for elementary policies. As a more fundamental extension, we distinguish especially the incorporation into the problem statement of the time value of money by means of discounting. In this case, adhering to the simple resource structure that is studied in this paper (a single machine) may no longer be advisable, and other scheduling policies that allow for parallel processing of activities will probably lead to better results.

Appendix A. Verification of Observation 3.2. We examine the hypothesis that for the MP1 instance presented in Section 3 the non-elementary policy Π^* as described by Fig. 3.1(b) would yield a higher expected total profit than any elementary policy. For our analysis, we divide all 56 elementary policies[2] that select at least one job into four classes according to the first job to be executed. Policies in class 1 execute job 1 in the first place, and similarly for classes 2, 3 and 4. Below, we derive a number of conditions on the parameters c, V and p for policy Π^* to be strictly better than the elementary policies. One additional requirement is that the policy have a strictly positive expected profit, i.e. Π^* is strictly better than the trivial (elementary) policy of abandoning the project immediately (represented by the empty list).

Comparison with class 1. We choose $p_1 = p_2 = p_3 = p_4 = \frac{1}{2}$ and we also impose

$$c_3 < c_2 \tag{A.1}$$

and

$$c_3 < c_4. \tag{A.2}$$

We will represent an elementary policy defined by job list $(abcd)$ as Π_{abcd}. Below, we establish conditions under which Π^* is better than each of the 14 policies in class 1.

(i) Inequality $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1234})]$ holds if $-c_3 - p_3c_2 + p_2p_3V > -c_2 - p_2c_3 + p_2p_3V + p_2q_3p_4V - q_3p_2c_4$, which leads to

$$c_2 + \frac{1}{2}c_4 > c_3 + \frac{1}{4}V. \tag{A.3}$$

[2] $56 = \binom{2}{1}\binom{2}{1}2 + \binom{4}{3}3! + \binom{4}{4}4!$

Table A.1
Comparison between policies Π_{1432} *and* Π^*

$\mathbb{E}[f(\Pi_{1432})]$	$\mathbb{E}[f(\Pi^*)]$	inequalities
$-c_1 - \frac{1}{8}(4c_4 + 2c_2 - V)$	$-c_1$	(A.5)
$\frac{1}{16}(V - 4c_3 - 2c_2)$	$\frac{1}{8}(V - 4c_3 - 2c_2)$	(A.6)
$-\frac{1}{4}(2c_4 + c_3) + \frac{3}{8}V$	$-\frac{1}{4}(2c_3 + c_4) + \frac{3}{8}V$	(A.2)

(ii) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1243})]$ due to our findings under 1 and the fact that $\mathbb{E}[f(\Pi_{1243})] < \mathbb{E}[f(\Pi_{1234})]$. The latter inequality is due to Eq. (A.2).

(iii) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1324})]$ if $0 > -c_2 - p_2c_4 + p_2p_4V$, which corresponds to

$$V < 4c_2 + 2c_4. \tag{A.4}$$

(iv) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{1342})]$ if $0 > -c_4 - p_4c_2 + p_2p_4V$, or

$$V < 4c_4 + 2c_2. \tag{A.5}$$

(v) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{134})]$ when $-c_3 - p_3c_2 + p_2p_3V > 0$, or

$$V > 4c_3 + 2c_2. \tag{A.6}$$

(vi) We compute $\mathbb{E}[f(\Pi_{1432})] = \left(\frac{9}{16}\right) V - \left(c_1 + \frac{3}{8}c_2 + \frac{1}{2}c_3 + c_4\right)$ and $\mathbb{E}[f(\Pi^*)] = \left(\frac{1}{2}\right) V - \left(c_1 + \frac{1}{4}c_2 + c_3 + \frac{1}{4}c_4\right)$. In Table A.1, both policies' profits are written as a sum of three terms and each term in the first column is strictly smaller than the corresponding term in the second column due to the equation referred to in the third column. We conclude that $\mathbb{E}[f(\Pi_{1432})] < \mathbb{E}[f(\Pi^*)]$.

(vii) From (A.1) and 6, we have $\mathbb{E}[f(\Pi_{1423})] < \mathbb{E}[f(\Pi_{1432})] < \mathbb{E}[f(\Pi^*)]$.

(viii) $\mathbb{E}[f(\Pi^*)] > \mathbb{E}[f(\Pi_{132})]$ if $p_4V - c_4 > 0$, or equivalently if

$$V > 2c_4. \tag{A.7}$$

(ix) From (A.2) and 5, we have $\mathbb{E}[f(\Pi_{143})] < \mathbb{E}[f(\Pi_{134})] < \mathbb{E}[f(\Pi^*)]$.

(x) From (A.1) and 8, we have $\mathbb{E}[f(\Pi_{123})] < \mathbb{E}[f(\Pi_{132})] < \mathbb{E}[f(\Pi^*)]$.

(xi) From (A.2) and 8, we have $\mathbb{E}[f(\Pi_{142})] < \mathbb{E}[f(\Pi_{132})] < \mathbb{E}[f(\Pi^*)]$.

(xii) From (A.2) and 10, we have $\mathbb{E}[f(\Pi_{124})] < \mathbb{E}[f(\Pi_{123})] < \mathbb{E}[f(\Pi^*)]$.

(xiii) From (A.6) and 8, we have $\mathbb{E}[f(\Pi_{13})] < \mathbb{E}[f(\Pi_{132})] < \mathbb{E}[f(\Pi^*)]$.

(xiv) From (A.2) and 13, we have $\mathbb{E}[f(\Pi_{14})] < \mathbb{E}[f(\Pi_{13})] < \mathbb{E}[f(\Pi^*)]$.

From the above we conclude that policy Π^* has an expected profit that is greater than the expected profit of any elementary policy of class 1 when inequalities (A.1)–(A.7) hold and all success probabilities are equal to 50%.

Comparison with classes 2, 3 and 4. For a policy of class 3, we look at the policy of class 1 obtained by interchanging job 3 by job 1 and job 2 by job 4 in the associated decision trees. This results in a one-to-one correspondence between the policies of class 1 and class 3 with equal corresponding expected profits if we further impose

$$c_1 = c_3, \tag{A.8}$$

$$c_2 = c_4. \tag{A.9}$$

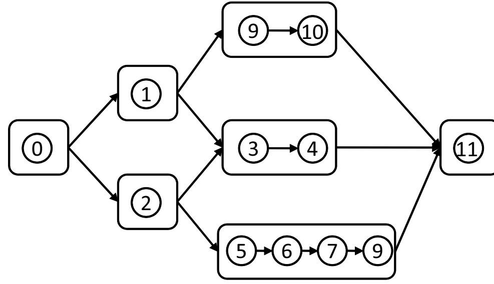


FIG. B.1. *Modular network of MP1 instance with* $n = 10$ *and* $m = 5$

For any elementary policy of class 2, we can look at a corresponding policy of class 1 by an interchange of jobs 1 and 2 in the associated decision trees. This policy is clearly better than the corresponding policy of class 2 because $c_1 < c_2$ follows from (A.8) and (A.1). We can develop a similar argument for class 4 by an interchange of jobs 4 and 3 and the result follows from (A.2).

Conclusion. Policy Π^* is a non-elementary policy for the example instance with an expected profit strictly better than any elementary policy if we can find values for c_i $(i = 1, 2, 3, 4)$ and V satisfying (A.1)–(A.9) and with positive expected profit (we have chosen $p_i = \frac{1}{2}$ for all i). The choice $c_1 = c_3 = 1$ and $c_2 = c_4 = 3$ satisfies Eqs. (A.1), (A.2), (A.8) and (A.9). Eqs. (A.3)–(A.7) impose $10 < V < 14$. If we select $V = 13$, for example, then $\mathbb{E}[f(\Pi^*)] = 3 > 0$. Optimal elementary policies, on the other hand, achieve an expected profit of $\mathbb{E}[\Pi(\cdot; L)] = 47/16 = 2.9375$, which is the case, for instance, for the elementary policy defined by job list $L = (1, 2, 3, 4)$.

Appendix B. Verification of Observation 3.3. We verify the observation that elementary policies can be arbitrarily bad. To this end, we present an instance for which the global optimum has a positive objective value while the optimal elementary policy is the empty policy, with zero objective. The instance is part of our second data set, consisting of general MP1 instances. The instance consists of $n = 10$ jobs divided over $m = 5$ modules. Fig. B.1 shows the modular network. The cost vector is $\mathbf{c} = (5, 4, 8, 0, 1, 30, 43, 24, 33, 50)$, the PTS vector is $\mathbf{p} = (0.819, 0.963, 0.912, 0.816, 0.840, 0.965, 0.891, 0.870, 0.876, 0.995)$ and the payoff V equals 63. The globally optimal policy is non-elementary, with expected profit equal to 1.18. Every non-empty compatible job list leads to an elementary policy with a negative expected profit.

Appendix C. Data generation. In the generated data sets, the costs c_k of an activity k are independent samples of a discrete uniform distribution on the set $\{0, 1, \ldots, 50\}$, and success probabilities p_k are independent samples of a continuous uniform distribution on the interval [0.8; 1]. For the first data set with n:n-systems, the end-of-project payoff V is an integer randomly selected from the interval $[0.5a; 2a]$, with a the value of the payoff such that the elementary policy $\pi(\cdot; L)$ produced by Algorithm 2 has zero expected profit, i.e. $a = (1/a_n) \sum_{k=1}^{n} a_{k-1} c_{L(k)}$, with $a_0 = 1$ and $a_k = \prod_{l=1}^{k} p_{L(l)}$, $k = 1, \ldots, n$.

For the second data set, for given n and OS, we generate five instances with $m = \lceil n/4 \rceil$ non-dummy modules and another five instances with $m = \lceil n/2 \rceil$ non-dummy modules. The module network consisting of the $m + 2$ modules is generated with RanGen; a discussion of the determination of the order strength OS' of the

module network is postponed to the next paragraph. Each module receives at least one activity, and the remaining $n-m$ activities are randomly allocated to the m modules. This results in a modular network with $B_i = \emptyset$ for all $i \in M$. If the order strength of the induced network is greater than or equal to OS, we stop. Otherwise, let $B'_i = \{(k,l) \mid k,l \in N_i,\ k < l\}$ and repetitively choose an intermodular precedence constraint (k,l) from $B' = \cup_{i \in M} B'_i$ at random and add it, together with any corresponding transitive elements, to the modular network until the order strength of the induced network exceeds or equals OS. Note that at each step of this process, set B' is updated by removing the newly added elements. The costs and probabilities are generated in the same way as for the first data set. The payoff V is chosen randomly in $[0.5b;\, 2b]$, where b is the value of the payoff such that the elementary policy $\pi(\,\cdot\,; L)$ produced by Algorithm 3 has zero expected profit.

The value of OS' is chosen such that a generated instance contains about half of the total number of possible intermodular precedence constraints on average. To this end, let t be the numerator of the order strength OS', i.e. $t = OS'\binom{m}{2}$, and let $\bar{n} = n/m$ be the average number of jobs per module. In case all modules contained exactly $\bar{n}$ activities, the order strength of the induced network without any intermodular precedence constraints would equal $t\bar{n}^2/\binom{n}{2}$. Furthermore, there are $m\binom{\bar{n}}{2}$ possible elements in B' such that aiming for half of the elements of B' leads to an instance with an order strength OS of its induced network equal to $\big(OS'\binom{m}{2}\bar{n}^2 + m\binom{\bar{n}}{2}/2\big)/\binom{n}{2}$. Solving this equation for OS' finally leads to the formula

$$OS' = \frac{m(n-1)OS - (n-m)/2}{n(m-1)}.$$

The actual number of jobs in a module can sometimes significantly differ from the average $\bar{n}$. Moreover, RanGen also approximates the order strength OS' of the generated networks. These uncertainties lead to a data set in which the total number of intermodular constraints of the instances is nicely spread between zero and the maximum value, where the intermodule precedence network is a chain.

Appendix D. Implementation of the hash table for the SDP. We implement the hash table by means of two different arrays A_1 and A_2. The key-value pairs are stored in array A_1 from left to right, complemented with an integer for dealing with collisions. This integer refers to the previous element of A_1 with the same hash value. More concretely, an element $A_1(i) = (A_1(i)(0), A_1(i)(1), A_1(i)(2))$ is a triple $(Y, G(Y), k)$ for $i = 0, \dots, |\mathcal{S}| - 1$, with k the largest integer smaller than i such that the key Y' of $A_1(k)$ maps to the same table entry, i.e. $g(Y) = g(Y')$. If no such integer exists, k equals -1. In this way, the keys of A_1 that map to the same table entry constitute a linked list. The linked list evolves from right to left in A_1 and halts at an element $A_1(i)$ with $A_1(i)(2) = -1$. The array A_2 is of size L and $A_2(i)$ is the index of array A_1 that holds the head of the linked list containing the elements hashed onto table entry i for $i = 0, \dots, L-1$. If this list is empty then we set $A_2(i) = -1$. When k elements are already present, a new element is added to the hash table following Algorithm 4. Note that elements are added to the front instead of to the end of the linked list, which offers two advantages compared to adding to the end. First, a new element can be included very quickly without running through the entire linked list. Second, to retrieve an element, the linked list is scanned from head to end; see Algorithm 5 for a description of the retrieval of a value function of a key. Since recently added elements are more likely to appear in further calculations, adding to the front of the list will also retrieve elements faster.

Algorithm 4 Insert new element $(Y, G(Y))$ in the hash table; assume k items are already stored

1: $i \leftarrow g(Y)$
2: $A_1(k) \leftarrow (Y, G(Y), A_2(i))$
3: $A_2(i) \leftarrow k$

Algorithm 5 Retrieve the value function of state Y

1: $i \leftarrow A_2(g(Y))$
2: **if** $(A_1(i)(0) == Y)$ **then**
3:     **return** $A_1(i)(1)$
4: **else**
5:     $i \leftarrow A_1(i)(2)$
6: **end if**
7: **if** $(i == -1)$ **then**
8:     item is not in list
9: **else**
10:     Go to line 2
11: **end if**

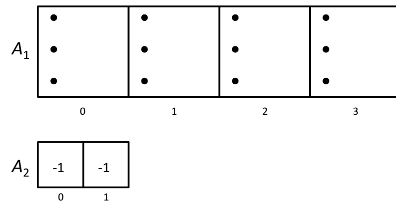
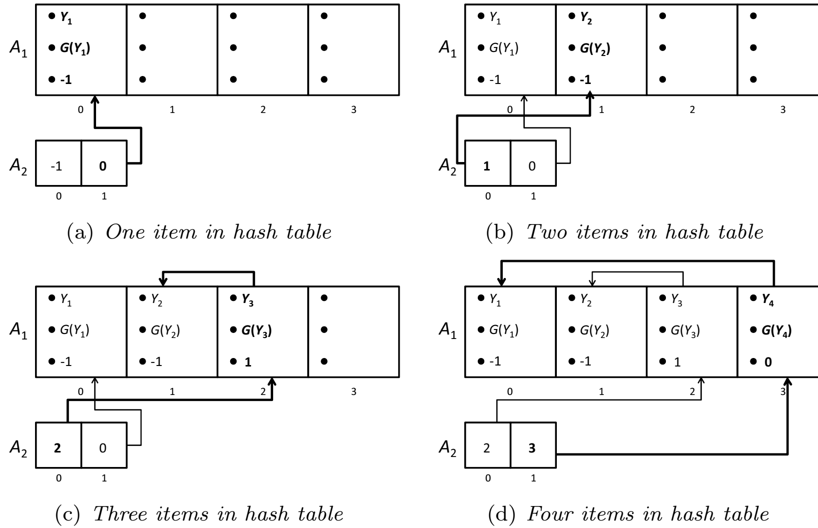


FIG. C.1. *Contents of arrays A_1 and A_2 when the hash table is empty*



(a) *One item in hash table* (b) *Two items in hash table*

(c) *Three items in hash table* (d) *Four items in hash table*

FIG. C.2. *Contents of arrays A_1 and A_2 when new item i is added to the table, $i = 1, 2, 3, 4$*

To illustrate the foregoing, consider an example involving the storage of four key-value pairs $(Y_i, G(Y_i)), i = 1, 2, 3, 4$. Assume $L = 2$ and $g(Y_1) = g(Y_4) = 1$, $g(Y_2) = g(Y_3) = 0$. Fig. C.1 depicts an empty hash table. Fig. C.2 illustrates the status of the lists A_1 and A_2 each time an item is added. At the end we obtain two linked lists, namely $3 \rightarrow 2$ and $4 \rightarrow 1$, corresponding to bucket entry 0 and 1.

Acknowledgements. This work was supported by research project G.0508.09 of the Research Foundation - Flanders (FWO) (Belgium). We are grateful to Philipp Melchiors (TU München) for some insightful suggestions.

REFERENCES

[1] H. Aytug, M.A. Lawley, K.N. McKay, S. Mohan, and R.M. Uzsoy. Executing production schedules in the face of uncertainties: A review and some future directions. *European Journal of Operational Research*, 165(1):86–110, 2005.

[2] E. Boros and T. Ünlüyurt. Diagnosing double regular systems. Technical Report RRR 30-97, RUTCOR - Rutgers Center for Operations Research, Rutgers University, US, 1997.

[3] R.W. Butterworth. Some reliability fault-testing models. *Operations Research*, 20(2):335–343, 1972.

[4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge MA and McGraw-Hill, New York, 1990.

[5] S. Creemers, R. Leus, and B. De Reyck. Project scheduling with alternative technologies: Incorporating varying activity duration variability. In *Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management (IEEM 2010)*, Macau, December 7-10 2010.

[6] S. Creemers, R. Leus, and M. Lambrecht. Scheduling Markovian PERT networks to maximize the net present value. *Operations Research Letters*, 38:51–56, 2010.

[7] A.J. Davenport and J.C. Beck. A survey of techniques for scheduling with uncertainty. Unpublished manuscript, available at website `http://www.eil.utoronto.ca/chris/chris.papers.html`, 2000.

[8] B. De Reyck, Y. Grushka-Cockayne, and R. Leus. A new challenge in project scheduling: The incorporation of activity failures. *Review of Business and Economics*, LII(3):411–434, 2007.

[9] B. De Reyck and R. Leus. R&D-project scheduling when activities may fail. *IIE Transactions*, 40(4):367–384, 2008.

[10] E. Demeulemeester, M. Vanhoucke, and W. Herroelen. A random generator for activity-on-the-node networks. *Journal of Scheduling*, 6:13–34, 2003.

[11] W. Herroelen and R. Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research*, 165(2):289–306, 2005.

[12] V. Jain and I.E. Grossmann. Resource-constrained scheduling of tests in new product development. *Industrial and Engineering Chemistry Research*, 38:3013–3026, 1999.

[13] D.S. Johnson, M. Yannakakis, and C.H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, 1988.

[14] S. Kavadias and C.H. Loch. Optimal project sequencing with recourse at a scarce resource. *Production and Operations Management*, 12(4):433–444, 2003.

[15] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, second edition, 1998.

[16] R. Kolisch. Efficient priority rules for the resource-constrained project scheduling problem. *Journal of Operations Management*, 14:172–192, 1996.

[17] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90:320–333, 1996.

[18] V. Kulkarni and V. Adlakha. Markov and Markov-regenerative PERT networks. *Operations Research*, 34:769–781, 1986.

[19] G. Malewicz. Parallel scheduling of complex dags under uncertainty. In *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms*, pages 66–75, 2005.

[20] L.G. Mitten. An analytic solution to the least cost testing sequence problem. *The Journal of Industrial Engineering*, page 17, January-Februari 1960.

[21] C.L. Monma and J.B. Sidney. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Research*, 4(3):215–224, 1979.

[22] S. Oliveira and D. Stewart. *Writing Scientific Software: A Guide to Good Style*. Cambridge University Press, 2006.

[23] F.J. Radermacher. Cost-dependent essential systems of ES-strategies for stochastic scheduling problems. *Methods of Operations Research*, 42:17–31, 1981.

[24] I. Sabuncuoglu and S. Goren. Hedging production schedules against uncertainty in manufacturing environment with a review of robustness and stability research. *International Journal of Computer Integrated Manufacturing*, 22(2):138–157, 2009.

[25] C.W. Schmidt and I.E. Grossmann. Optimization models for the scheduling of testing tasks in new product development. *Industrial and Engineering Chemistry Research*, 35:3498–3510, 1996.

[26] T.A. Standish. *Data Structures, Algorithms & Software Principles in C*. Addison-Wesley, 1995.

[27] F. Stork. *Stochastic resource-constrained project scheduling*. PhD thesis, TU Berlin, Germany, 2001.

[28] T. Ünlüyurt. Sequential testing of complex systems: A review. *Discrete Applied Mathematics*, 142:189–205, 2004.

[29] J. Valdes, R.E. Tarjan, and E.L. Lawler. The recognition of series parallel digraphs. *SIAM Journal on Computing*, 11(2):298–313, 1982.

[30] G.E. Vieira, J.W. Herrmann, and E. Lin. Rescheduling manufacturing systems: A framework of strategies, policies, and methods. *Journal of Scheduling*, 6(1):39–62, 2003.

[31] T. Wang. Integer hash function. Technical report, HP Enterprise Java Lab, 2007. Available at website `http://www.concentric.net/~ttwang/tech/inthash.htm`.