doi:10.1016/j.orl.2009.10.006 • www.stefancreemers.be • info@stefancreemers.be

Scheduling Markovian PERT networks with maximum-NPV objective

Stefan Creemers
Roel Leus
Marc Lambrecht

Abstract - **We examine project scheduling with net-present-value objective and exponential activity durations, using a continuous-time Markov decision chain. Based on a judicious partitioning of the state space, we achieve a significant performance improvement compared to the existing algorithms.**

Keywords - **project scheduling, net present value, stochastic activity durations, exponential distribution**

1 Introduction

A project consists of a set of activities (or tasks) $N = \{0, 1, \ldots, n\}$, which are to be processed without interruption. The duration D_i of activity i is a random variable (r.v.); the vector $(D_0, D_1, \ldots, D_n)$ is denoted by $\mathbf{D}$. The set A is a strict order on N, i.e. an irreflexive and transitive relation, which represents technological precedence constraints. The activities 0 and n represent start and end of the project, respectively, and are the (unique) least and greatest element of the partially ordered set (N, A).

Each activity $i \in N$ is associated with a cash flow c_i, which is an integer value that may be positive or negative; this quantity is received or paid at the start of the activity. In order to account for the time value of money, we define r to be the applicable continuous discount rate: the present value of a cash flow c incurred at time t equals ce^{-rt}. We use lowercase vector $\mathbf{d} = (d_0, d_1, \ldots, d_n)$ to represent one particular realization (or sample, or scenario) of $\mathbf{D}$. Alternatively, when each duration D_i is a constant, we use the same notation $\mathbf{d}$. For a given realization $\mathbf{d}$, we can produce a schedule $\mathbf{s}$, i.e., a vector of starting times $(s_0, s_1, \ldots, s_n)$ with $s_i \geq 0$ for all $i \in N$. The schedule $\mathbf{s}$ is *feasible* if $s_i + d_i \leq s_j$ for all $(i, j) \in A$.

In the absence of resource constraints, the minimum-makespan objective requires no real scheduling effort: all activities are started as soon as their predecessors are completed. The literature on this so-called PERT problem is usually concerned with the computation of certain characteristics of the minimum project makespan (earliest project completion) when the activity durations are random variables, mainly with exact computation, approximation and bounding of the distribution function and the expected value [1, 4, 7]. For the case of independent exponential activity durations, Kulkarni and Adlakha [6] describe an exact method for deriving the distribution and moments of the project completion time using continuous-time Markov chains. A Markovian PERT network is a PERT network with independent and exponentially distributed activity durations.

In this article, we focus on project scheduling with NPV (net present value) objective and exponential durations [10, 2]. Both Tilson et al. [10] and Buss and Rosenblatt [2] use the continuous-time Markov chain (CTMC) described in [6] as a starting point for their algorithm. We achieve a significant performance improvement compared to these existing approaches, based on a judicious partitioning of the state space.

2 Problem statement

The execution of a project with stochastic durations can best be seen as a dynamic decision process. A solution is a *policy* Π, which defines *actions* at *decision times*. Decision times are typically $t = 0$ (the start of the project) and the completion times of activities; a tentative next decision time can also be specified by the decision maker. An action can entail the start of a set of activities that is 'feasible', so that a feasible schedule is constructed gradually through time. Next to the input data of the problem instance, a decision at time t may only use information (on activity durations) that has become available before or at time t; this requirement is often referred to as the *non-anticipativity constraint*.

As soon as all activities are completed, the activity durations are known, yielding a realization $\mathbf{d}$ of $\mathbf{D}$. Consequently, every policy Π may alternatively be interpreted [5] as a function $\mathbb{R}_{\geq}^{n+1} \mapsto \mathbb{R}_{\geq}^{n+1}$ that maps given samples $\mathbf{d}$ of activity durations to vectors $\mathbf{s}(\mathbf{d};\Pi)$ of feasible activity starting times (schedules). For a given scenario $\mathbf{d}$ and policy Π, $s_n(\mathbf{d};\Pi)$ denotes the makespan of the schedule. The earlier-mentioned PERT problem aims at characterizing the r.v. $s_n(\mathbf{D};\Pi^{ES})$, where policy Π^{ES} starts all activities as early as possible. NPV, however, is a non-regular measure of performance: starting activities as early as possible is not necessarily optimal.

In this text we investigate the determination of an optimal scheduling policy for the expected-NPV objective. In the special case where the durations have constant values $\mathbf{d}$, the objective function corresponding with a schedule $\mathbf{s}$ is the following:

$$\max \qquad g(\mathbf{s},\mathbf{d}) = \sum_{i=0}^{n} c_i e^{-rs_i}.$$

Our goal in this article is to select a policy Π^* within a specific class that maximizes $E[g(\mathbf{s}(\mathbf{D};\Pi),\mathbf{D})]$, with $E[\cdot]$ the expectation operator with respect to $\mathbf{D}$. The generality of this problem statement suggests that optimization over the class of all policies will probably turn out to be computationally intractable. We therefore restrict our attention to a subclass that has a simple combinatorial representation and where decision points are limited in number: our solution space consists of all policies that start activities only at the end of other activities (activity 0 is started at time 0).

3 The algorithm

We assume the durations of the activities $i \in N \setminus \{0,n\}$ to be mutually independent exponentially distributed r.v.s with mean $\frac{1}{\mu_i}, \mu_i > 0$. When the objective function is the expected makespan, the early-start policy Π^{ES} is always optimal, but this is no longer the case for

NPV. Section 3.1 briefly presents the state space of our search procedure, Section 3.2 discusses how we partition this state space in order to facilitate memory management, and the stochastic dynamic-programming (SDP) algorithm that produces an optimal policy is the subject of Section 3.3.

3.1 State space

At any time instant t, each activity's *status* is either *idle* (= unstarted), *active* (= in the process of being executed) or *finished*; we write $\Omega_i(t) = 0$, 1 or 2, respectively, for $i \in N$. The state of the system is defined by the status of the individual activities and is represented by vector $\Omega(t) = (\Omega_0(t), \Omega_1(t), \ldots, \Omega_n(t))$. State transitions take place each time an activity finishes and are determined by the policy at hand. The project's starting and finishing conditions are $\forall i \in N : \Omega_i(0) = 0$ and $\forall i \in N : \Omega_i(t) = 2$, $\forall t \geq \omega$, respectively, where ω indicates the project completion time. The problem of finding an optimal scheduling policy corresponds to optimizing a discounted criterion in a continuous-time Markov decision chain (CTMDC) on the state space Q, with Q containing all states of the system that can be visited by the transitions (which are called *feasible* states); the decision set is described in Section 3.3.

An upper bound on $|Q|$ is 3^n. Enumerating all these 3^n states is not recommendable, because typically the majority of the states do not satisfy the precedence constraints. Tilson et al. [10] develop a simple yet efficient algorithm to produce a set of possible states; this set contains Q but may be strictly larger. Additionally, to the best of our knowledge, all related studies in the literature reserve memory space to store the entire state space of the CTMDC; Buss and Rosenblatt [2] point out that some method of decomposition to reduce these memory requirements would allow for considerable efficiency enhancements. In what follows, we present an algorithm that considerably improves upon the storage and computational requirements of earlier algorithms by means of efficient creation of Q and decomposition of the network of state transitions.

3.2 Uniformly directed cuts

Our algorithm consists of two main steps. The first step is discussed in this section, and consists of the generation of all inclusion-maximal antichains of A (sets of activities that can be executed in parallel); Kulkarni and Adlakha [6] refer to these sets as *uniformly directed cuts* or *UDCs*, and we will maintain this term (although we work with activity-on-the-node instead of activity-on-the-arc representation). In the second step of the algorithm we apply a backward SDP-recursion to determine optimal decisions; this recursion is the subject of Section 3.3.

Let $\mathcal{U} = \{U_1, U_2, \ldots, U_{|\mathcal{U}|}\}$ denote the set of UDCs. Formally, $\mathcal{U}$ is the maximum-size subset of the power set 2^N whose elements U satisfy the following conditions:

(1) $\forall \{i, j\} \subset U : (i, j) \notin A \wedge (j, i) \notin A$,

(2) $\nexists u \in N \setminus U : U \cup \{u\}$ satisfies condition (1).

We associate with each $U \in \mathcal{U}$ a *rank* number $r(U) = |\{i \in N : \exists j \in U | (i, j) \in A\}|$, which counts the number of predecessor activities; the elements U are indexed in non-decreasing

rank $r(U)$ (with arbitrary tiebreaker). Throughout Sections 3.2 and 3.3 we use the project network depicted in Figure 1 as an illustration. All the corresponding elements of $\mathcal{U}$ are represented in Figure 2; UDCs with the same rank have the same horizontal position (they are in the same 'column').

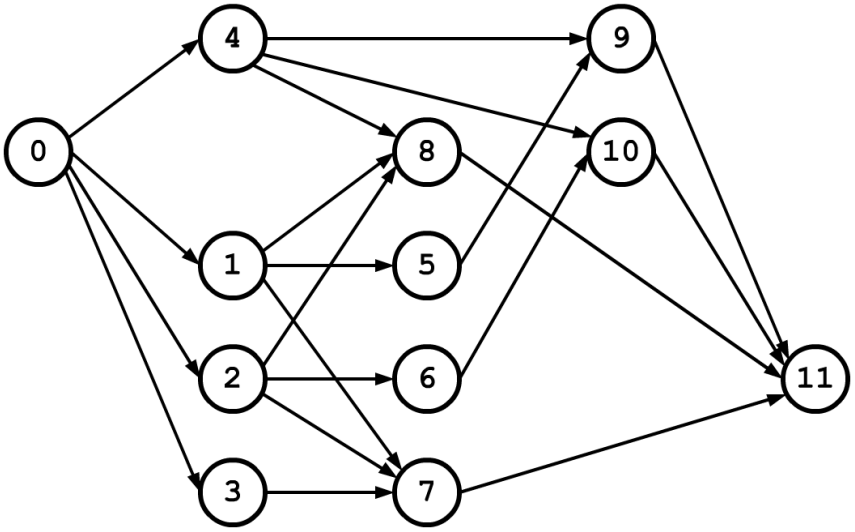


Figure 1: Example project network.

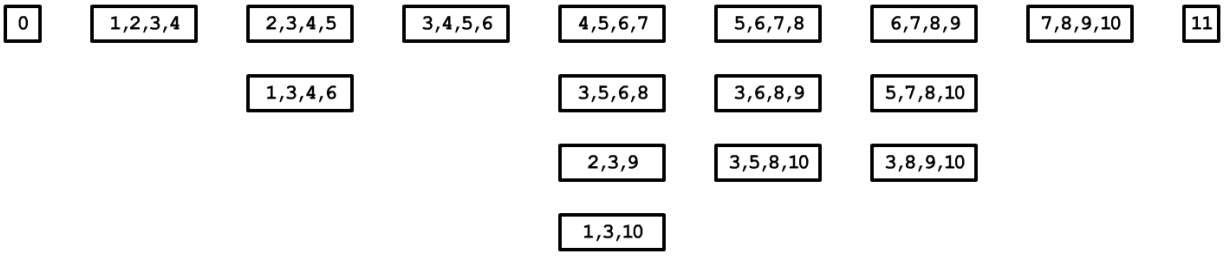
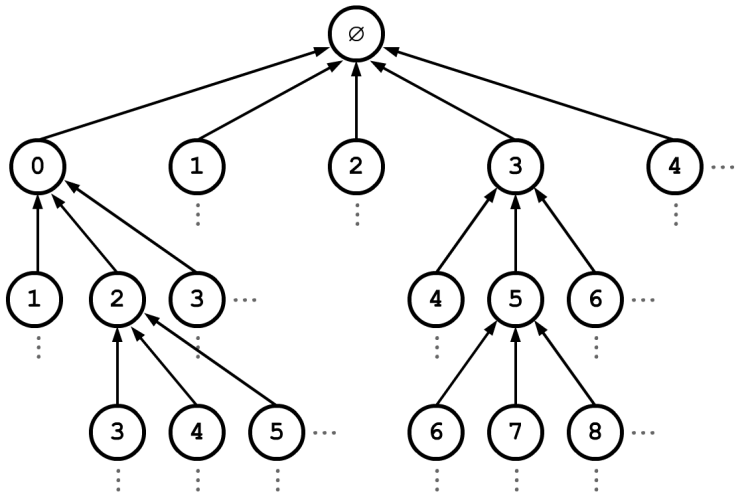


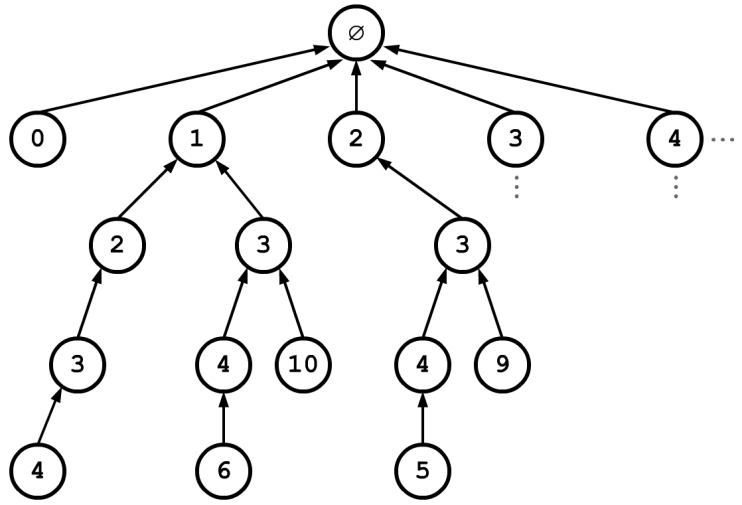
Figure 2: All UDCs of the example project network.

In a similar way as in [9] for the generation of all minimal forbidden sets of an instance of a resource-constrained scheduling problem, we apply a simple backtracking algorithm to enumerate all UDCs. The subsets of N are enumerated in a tree T where each node w of T, except the root node, is associated with exactly one activity $i \in N$. If node w is associated with activity i then w has a child node for each activity $i = j + 1, \ldots, n$. In this way, there is a one-to-one correspondence between the set of nodes of T and the power set 2^N; each node w of the tree corresponds with a subset $W \subseteq N$ of activities, for which the activities are collected by traversing the tree from w to the root node. In order to obtain only the UDCs and no other subsets, the tree T is pruned during this generic process: a node w is

discarded as soon as its associated activity i is comparable to an activity associated with at least one of the ancestor nodes. The resulting tree is referred to as $\mathcal{T}(A)$.



(a) T



(b) $\mathcal{T}(A)$

Figure 3: The trees T and $\mathcal{T}(A)$ for the example project.

The total number of UDCs can clearly be exponential in the number of activities, and more efficient approaches for the enumeration may exist (cfr. [8]), but we content ourselves with this simple algorithm for two reasons: (1) this enumeration is not the bottleneck in the running time of the search procedure for optimal scheduling policies, and (2) during the recursion that is applied to enumerate and prune the tree, additional information on movement between UDCs (see infra) can be conveniently gathered.

Each UDC U is associated with a set $\sigma(U)$ of states in which one or more activities in U are active and the remaining activities in $N\backslash U$ are either idle or finished: $\sigma(U)$ contains

all states $\mathbf{v} \in Q$ with $\sigma^{-1}(\mathbf{v}) = U$; the function $\sigma^{-1} : Q \mapsto \mathcal{U}$ is a surjection and is defined in the following way. Let $\mathbf{v} = (v_0, v_1, \ldots, v_n)$ be an arbitrary state. We construct the set $\zeta(\mathbf{v}) = \{i \in N : v_i = 1 \vee v_i = 2 \vee (\forall j \in N | (j,i) \in A : v_j = 2)\}$ and we let $\sigma^{-1}(\mathbf{v})$ contain only the maximal elements in the partially ordered set $(\zeta(\mathbf{v}), A(\zeta(\mathbf{v})))$, i.e. the activities without successors, where $A(\zeta(\mathbf{v}))$ is the subset of activity pairs in A with both elements in $\zeta(\mathbf{v})$. The function $\sigma : \mathcal{U} \mapsto 2^Q$ is such that $\{\sigma(U_1), \ldots, \sigma(U_{|\mathcal{U}|})\}$ is a partition of Q. The correctness of this approach derives from the following lemma:

Lemma 1 *For any feasible state* $\mathbf{v}$*, the set* $\sigma^{-1}(\mathbf{v})$ *is an element of* $\mathcal{U}$*.*

Proof 1 *The set of activities* $J_1 = \{i \in N : v_i = 1\}$ *needs to be an antichain of* A *for* $\mathbf{v}$ *to be feasible.*

The extension of set J_1 *with* $J_2 = \{i \in N : v_i = 0 \wedge (\forall j \in N | (j,i) \in A : v_j = 2)\}$ *is still an antichain: each of the additional activities can be started immediately.*

The extension of set $J_1 \cup J_2$ *with* $J_3 = \{i \in N : v_i = 2\}$ *is no longer an antichain, but restriction to only the maximal elements of* $\zeta(\mathbf{v}) = J_1 \cup J_2 \cup J_3$ *does preserve that property: any additional activity has no successors in* $J_1 \cup J_2 \cup J_3$*.*

It still remains to show that the antichain $\sigma^{-1}(\mathbf{v})$ *is inclusion-maximal. To see this, suppose that an activity* $j \in N \setminus \sigma^{-1}(\mathbf{v})$ *exists that can be added to* $\sigma^{-1}(\mathbf{v})$ *and the result is still an antichain. Three cases can be distinguished: (1)* $v_j = 0$*, in which case either* $j \in J_2$ *or* $\exists k \in N : (k,j) \in A \wedge v_k \in \{0,1\}$*. If* $v_k = 1$ *then* $k \in J_1$*, otherwise* $v_k = 0$*, and we can repeat the same reasoning; since there is only a finite number of activities, we will end up with a contradiction; (2)* $v_j = 1$*, which is impossible since all* $k \in N$ *with* $v_k = 1$ *are in* J_1*; or (3)* $v_j = 2$ *so* $j \in J_3$*, and either* $j \in \sigma^{-1}(\mathbf{v})$ *or* j *is a predecessor of one or more activities in* $J_1 \cup J_2 \cup J_3$*.*

As an example, the vector $\mathbf{v}_1 = (2,2,2,2,1,1,2,1,0,0,0,0)$ is a feasible state for the illustration project; $\sigma^{-1}(\mathbf{v}_1)$ is the UDC $\{4,5,6,7\}$.

When an activity finishes, the current state is left and another state is entered. The following observations will be useful in Section 3.3: call an activity i *eligible* if $v_i = 0$ and $\forall j \in N | (j,i) \in A : v_j = 2$.

Lemma 2 *If at least one new activity becomes eligible then the system moves to a different UDC, otherwise we remain in the same UDC.*

Proof 2 *This follows directly from the definition of* $\sigma^{-1}(\mathbf{v})$*: if no new activity becomes eligible then the only activities with a changing state are activities* i *whose state* v_i *goes from 1 to 2, leading to the same UDC. If a new activity becomes eligible then it was previously not in the current UDC.*

Lemma 3 *Inter-UDC-transitions can only lead from a lower- to a higher-ranked UDC.*

Proof 3 *If we move to a new UDC then at least one new activity is included in the UDC, which is implicit from Lemma 2. Since this activity was not previously eligible, the finishing activity that leads to the transition is a predecessor of the new activity, and so the UDC-rank strictly increases.*

Lemma 3 implies that state transitions between different UDCs only take place from one UDC to another UDC further right in Figure 2. Since there are links between certain, but not all, pairs of UDCs, we shall refer to the corresponding system of UDCs with an indication of the possible transitions as the *UDC-network*. The recognition of all inter-UDC-transitions is embedded in our enumeration of the UDCs, and for each UDC U_i we record the number l_i^I of incoming transitions.

In the example project, the UDC $U_9 = \{1, 3, 10\}$ is associated with 18 states, which are listed in Table 1. Since the completion of activity 2 makes new activities eligible, only states in which the status of activity 2 is in $\{0, 1\}$ are included in $\sigma(U_9)$. There are two outgoing transitions, both associated with the completion of activity 2: if activity 4 is idle or active then the new UDC is $\{3, 5, 8, 10\}$, otherwise (in case activity 4 is completed) $\{5, 7, 8, 10\}$ is entered. The number l_9^I of incoming transitions is 1, coming from $\{1, 3, 4, 6\}$.

The feasible states are indexed based on (1) the rank of the UDC they belong to; (2) the number of the UDC at that rank; and (3) the 'tertiary value' for the state. For each UDC U we identify the activities in U as $\rho_i \in N$, for $i = 1, \ldots, |U|$; the activities are ordered in increasing index (we omit the parameter U to operator ρ). For U_9, we have $\rho_1 = 1$, $\rho_2 = 3$ and $\rho_3 = 10$. The *tertiary value* $\tau(\mathbf{v})$ of a state $\mathbf{v} \in \sigma(U)$ is defined as follows:

$$\tau(\mathbf{v}) = \sum_{i=1}^{|U|} v_{\rho_i} 3^{i-1}.$$

We use these tertiary values to order the states in each UDC; this enables the implementation of a binary search procedure for efficient look-up of state information. The tertiary values of the states in $\sigma(U_9)$ are given in Table 1.

completed	in progress	idle	tertiary value
10,3	1		25
10,3		1	24
10	3,1		22
10	3	1	21
10	1	3	19
10		3,1	18
3	10,1		16
3	10	1	15
3	1	10	13
3		10,1	12
	10,3,1		10
	10,3	1	9
	10,1	3	7
	10	3,1	6
	3,1	10	4
	3	10,1	3
	1	10,3	1
		10,3,1	0

Table 1: A listing of all elements of $\sigma(U_9) = \sigma(\{1, 3, 10\})$.

3.3 Details on the dynamic program

In Section 3.2 we have discussed the construction of the UDC-network. This network serves as the backbone of the state space of the CTMDC. In the second step of the algorithm (the first step was the construction of the UDC-network), we apply a backward SDP-recursion to determine optimal decisions. For ease of description, we resort to yet another characterization of a state $\mathbf{v} \in Q$: we let $I(\mathbf{v})$, $X(\mathbf{v})$ and $L(\mathbf{v})$ represent the activities in N that are idle, active and finished, respectively; these sets were represented in the first three columns of Table 1. There is a one-to-one correspondence between (not necessarily feasible) state vectors $\mathbf{v}$ and mutually exclusive sets I, X and L with $I \cup X \cup L = N$, and in our actual implementation we only use the latter characterization of states, together with the tertiary values. The key instrument of the SDP-recursion is the *value function* $G(\cdot)$, which determines the NPV of each feasible state at the time of entry of the state, conditional on the hypothesis that optimal decisions are made in all subsequent states. In the definition of the value function $G(I, X)$, we supply sets I and X of idle and active activities (which uniquely determines the finished activities).

At the entry of a state $\mathbf{v} \in Q$, a decision needs to be made whether to start a set of eligible activities (and if so, which), or not to start any activities; the latter decision is possible only if $X(\mathbf{v}) \neq \varnothing$. If no activities are started, a transition towards another state takes place after the first completion of an element of $X(\mathbf{v})$. The probability that activity $i \in X(\mathbf{v})$ finishes first among the active activities equals $\mu_i / \sum_{k \in X(\mathbf{v})} \mu_k$. The expected time to the first completion is $\left(\sum_{i \in X(\mathbf{v})} \mu_i \right)^{-1}$ time units (the length of this timespan is also

exponentially distributed). The appropriate discount factor to be applied for this timespan is $\sum_{k\in X(\mathbf{v})}\mu_k/\left(r+\sum_{k\in X(\mathbf{v})}\mu_k\right)$. The expected discounted NPV to be obtained from the next state, on condition that no new activities are started, therefore equals

$$\frac{\sum_{k\in X(\mathbf{v})}\mu_k}{r+\sum_{k\in X(\mathbf{v})}\mu_k}\sum_{i\in X(\mathbf{v})}\frac{\mu_i}{\sum_{k\in X(\mathbf{v})}\mu_k}G(I(\mathbf{v}),X(\mathbf{v})\setminus\{i\}). \tag{1}$$

The second alternative is to start a non-empty set of eligible activities $S\subseteq\sigma^{-1}(\mathbf{v})\cap I(\mathbf{v})$ when state $\mathbf{v}\in Q$ is entered. This leads to incurring a cost $\sum_{i\in S}c_i$ and an immediate transition to another state in the same UDC, with no discounting required. The corresponding immediate NPV (in expectation), conditional on set $S\neq\varnothing$ being started, is

$$\sum_{i\in S}c_i+G(I(\mathbf{v})\setminus S,X(\mathbf{v})\cup S). \tag{2}$$

The total number of decisions S that can be made is $2^{|\sigma^{-1}(\mathbf{v})\cap I(\mathbf{v})|}$ if we include $S=\varnothing$ according to Equation (1), otherwise it is one less. The decision corresponding with the highest value in (1) and (2) determines $G(I(\mathbf{v}),X(\mathbf{v}))$, which completes our description of the SDP-recursion. The optimal objective-function value is $\max_\Pi E[g(\mathbf{s}(\mathbf{D};\Pi),\mathbf{D})]=G(N,\varnothing)$.

In order to quickly look up the necessary $G()$-values for the recursion, we use the tertiary values. For a current state $\mathbf{v}$ and decision $S\subseteq\sigma^{-1}(\mathbf{v})\cap I(\mathbf{v})$, the UDC and the tertiary value of the destination state are easily determined, and this information is stored as additional data at the construction of the UDC-network. The value function for the appropriate arguments is then retrieved via binary search.

The recursion starts in the final UDC in state $(2,2,\ldots,2,0)$, so we omit states $(2,2,\ldots,1)$ and $(2,2,\ldots,2)$. Stepwise, the value function is computed for states associated with lower-ranked UDCs. As the algorithm progresses, the states in higher-ranked UDCs will no longer be required for further computation and therefore the memory they occupy can be freed – whence the usefulness of the decomposition of the project network into different UDCs. Our overall search procedure is described as Algorithm 1. The trouble-free functioning of our approach is confirmed by the following result, which, together with Lemma 3, guarantees that we only look up $G()$-values for states that have already been created.

Lemma 4 *For an arbitrary UDC U_i, in any state $\mathbf{v}\in\sigma(U_i)$, the backward SDP-recursion only needs value-function look-ups within the same UDC for states $\mathbf{u}$ with $\tau(\mathbf{v})<\tau(\mathbf{u})$.*

Proof 4 *Any activity's status can only change from idle to active to completed as time progresses, and when the status of the other activities remains unchanged, the status idle, active and completed will always correspond with lowest, middle and highest tertiary value.*

Algorithm 1 Global algorithmic structure

```
Generate the UDC-network
G({n}, ∅) = c_n
for i = |U| − 1 downto 1 do
  Allocate storage for all states in σ(U_i)
  for all states v ∈ σ(U_i) in decreasing τ(v) do
    Determine an optimal decision and the value function
  end for
  for all UDCs U_k ≠ U_i reached by a look-up do
    Decrement l_k^I
    if l_k^I = 0 then
      Free storage occupied by σ(U_k)
    end if
  end for
end for
```

4 Performance

The algorithms currently available in the literature are able to solve project instances with up to 25 activities, with performance depending on the density of the network. An actual comparison is possible only with the algorithm of Tilson et al. [10]; Buss and Rosenblatt [2] always start *all* eligible activities as soon as possible (but *after* a delay period, which is individually chosen for each activity).

Out of 30 randomly generated networks with 25 activities, Tilson et al. solve 29, 20 and 0 networks when the order strength OS amounts to 75%, 50% and 25%, respectively (OS is the number of comparable activity pairs divided by the maximum possible number of such pairs). The main bottleneck of their approach is the memory constraint. Using a Pentium 4 with 2.8 GHz CPU-speed and 512 MB of RAM, Tilson et al. are restricted to project networks featuring a maximum of 600,000 states. In our model, storage requirement for 600,000 states amounts to a maximum of 4.58 MB (including both the storage of tertiary values as well as NPV-values for each state), enabling the solution of significantly larger instances. Our experiments are performed on an AMD Athlon with 1.8 GHz CPU-speed and 2,048 MB of RAM. Under this configuration, a state space of maximum 268,435,456 states can be entirely stored in memory. Moreover, the decomposition of the project network into UDCs yields additional significant reductions in storage requirements. During our experiments, the instance with the largest size of Q to be successfully analyzed had 867,589,281 states (resulting in an improvement of memory efficiency with factor 360 if correct allowance is made for the difference in RAM).

The actual datasets examined by Tilson et al. are not available from the authors, so we have decided to generate our own scheduling instances. We have used RanGen [3] to create a dataset with 30 instances for each of the parameter settings $OS = 0.4$, 0.6 and 0.8, and this for different values of n. The sign of the cash flows is unimportant to our algorithm; in the generated instances all activities apart from the final one have negative cash flows and the final activity has a positive cash flow (which is also significantly larger in absolute value).

n	n_s			$\|Q\|$		
	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	30	30	30	71	206	695
20	30	30	30	484	4,006	55,016
30	30	30	30	1,995	49,388	1,560,364
40	30	30	29	7,860	534,014	47,072,515
50	30	30	4	26,667	4,346,215	526,020,237
60	30	30	0	92,003	42,278,506	
70	30	22	0	286,831	216,027,815	
80	30	9	0	829,741	743,325,011	
90	30	0	0	2,596,419		
100	30	0	0	6,868,100		
110	30	0	0	24,235,588		
120	30	0	0	146,639,043		

n	Average CPU Time			Average statespace use		
	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$	$OS = 0.8$	$OS = 0.6$	$OS = 0.4$
10	0.00	0.00	0.00	0.25	0.37	0.44
20	0.00	0.03	0.90	0.22	0.27	0.38
30	0.01	0.64	52.98	0.15	0.24	0.30
40	0.06	13.29	4,273	0.15	0.28	0.29
50	0.27	171.56	99,216	0.16	0.24	0.17
60	1.28	4,048		0.16	0.33	
70	5.37	33,203		0.16	0.19	
80	19.13	115,377		0.13	0.11	
90	86.86			0.16		
100	301.03			0.17		
110	1,774			0.19		
120	19,215			0.16		

Table 2: Computational results. We report the number of successfully analyzed networks out of 30 ("n_s"), the average size of the state space ("$|Q|$"), the CPU-time required to find an optimal policy and the average fraction of states simultaneously in memory.

Our results are presented in Table 2, gathered per combination of values for OS and n. The table shows that networks of up to 40 activities are analyzed with relative ease. When $n = 50$, however, the optimal solution of most networks with low order strength ($OS = 0.4$) is beyond reach when the system memory is restricted to 2,048 MB. When $OS = 0.6$ performance is limited to networks with $n = 80$. We observe that the density of the network is a major determinant for both the computation times as well as the benefits of the UDC-decomposition: order strengths and computation times clearly display an inverse relation. With respect to compuational performance, Tilson et al. report only their longest

computation time, which amounts to 210 seconds. For problems of comparable complexity (i.e. requiring a statespace size of approximately 600,000 states) our algorithm takes 14 seconds to compute the optimal NPV. Not taking into account the difference of computational performance (in fact, the CPU used by Tilson et al. is significantly more powerful than the one used in this study), this results in an improvement of factor 15.

References

[1] V.G. Adlakha and V.G. Kulkarni. A classified bibliography of research on stochastic PERT networks: 1966-1987. *INFOR*, 27:272–296, 1989.

[2] A.H. Buss and M.J. Rosenblatt. Activity delay in stochastic project networks. *Operations Research*, 45:126–139, 1997.

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

[4] S.E. Elmaghraby. *Activity Networks: Project Planning and Control by Network Models*. Wiley, 1977.

[5] G. Igelmund and F.J. Radermacher. Preselective strategies for the optimization of stochastic project networks under resource constraints. *Networks*, 13:1–28, 1983.

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

[7] A. Ludwig, R.H. Möhring, and F. Stork. A computational study on bounding the makespan distribution in stochastic project networks. *Annals of Operations Research*, 102:49–64, 2001.

[8] D.R. Shier and D.E. Whited. Iterative algorithms for generating minimal cutsets in directed graphs. *Networks*, 16:133–147, 1986.

[9] F. Stork and M. Uetz. On the generation of circuits and minimal forbidden sets. *Mathematical Programming*, 102:185–203, 2005.

[10] V. Tilson, M.J. Sobel, and J.G. Szmerekovsky. Scheduling projects with stochastic activity duration to maximize EPV. *SSRN eLibrary*, 2006. Working Paper Series; available at http://ssrn.com/paper=1015785.