Sequential testing policies for complex systems under precedence constraints

Wenchao Wei*, Kris Coolen*, Roel Leus*†

Abstract. We study the problem of sequentially testing the components of a multi-component system to learn the state of the system, when the tests are subject to precedence constraints and with the objective of minimizing the expected cost of the inspections. Our focus is on \(k\)-out-of-\(n\) systems, which function if at least \(k\) of the \(n\) components are functional. A solution is a testing policy, which is a set of decision rules that describe in which order to perform the tests. We distinguish two different classes of policies and describe exact algorithms (one branch-and-bound algorithm and one dynamic program) to find an optimal member of each class. We report on extensive computational experiments with the algorithms for representative datasets.

Keywords: sequential testing, precedence constraints, branch and bound, dynamic programming.

1. Introduction

The diagnosis of a complex system that consists of a number of components, is relevant to many different fields. It requires a series of tests of the components to determine the state of the system, which can be either working or failing. Testing a component can cost manpower and other resources (such as electricity, water, oil, etc.) or entail other costs (e.g., pain to a patient) and the development of a testing strategy that minimizes the expected cost of all tests is therefore important in many situations. The state of the system depends on the states of the components, and a component’s state can only be discovered through inspection. In this paper, we consider the particular setting where the cost of testing each component as well as the (a priori) probability that it is working correctly, are known in advance. Furthermore, the components are assumed to work or fail independently of each other. In line with Ünlüyurt (2004), we observe that since the inspection of the system is usually repeated many times (the same system is inspected in various different situations; several equivalent systems are inspected; etc.), it is customary to consider the problem of finding a policy that minimizes the associated expected cost. This corresponds to the practical desire of minimizing the total cost in the long run.

The main goal of this article is to fill a gap in the existing literature: exact algorithms for computing optimal inspection policies and the corresponding computational results have, to the best of our knowledge, not been published to date for so-called \(k\)-out-of-\(n\) systems under precedence constraints (see the next section for a formal problem statement). This void is also recognized in the recent article by Çatay, Özlük, and Ünlüyurt (2011). In this text, we distinguish two different classes of policies and describe exact algorithms (one branch-and-bound algorithm and one dynamic program) to find an optimal member of each class. We report on extensive computational experiments with the algorithms for representative datasets.

The remainder of this article is structured as follows. Section 2 presents a number of definitions and a detailed problem statement. Subsequently, we survey the related scientific literature in Section 3. Different classes of policies are discussed in Section 4. A dynamic program and a branch-and-bound algorithm are described in Section 5 and Section 6 respectively. The computational performance of the algorithms is evaluated in Section 7. We round off the article with a summary in Section 8.

2. Problem statement

We study a general multi-component system consisting of a set \(N = \{1, 2, \ldots, n\}\) of components. Each component is in one of two states: either faulty (failed, down, not working) or fault-free (successful, up,
A scenario $x = (x_1, x_2, \ldots, x_n)$ is a vector consisting of $n$ binary values. Value $x_i = 0$ means that component $i$ fails, otherwise ($x_i = 1$) it is fault-free. We let $B$ represent the space of possible state vectors (so $B = \{0, 1\}^n$). The system’s state is the output of a Boolean mapping $f : B \mapsto \{0, 1\}$, which is also called the system function; $f(x) = 1$ means that the system works under scenario $x$, otherwise it fails.

The system function can for instance take the following form:

$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} \omega_i \cdot x_i \geq k \\ 0 & \text{otherwise} \end{cases}$$

Such functions are called threshold functions when the $\omega_i$ are arbitrary positive weights. A system function is said to be monotone if $f(x) \geq f(y)$ whenever $x_i \geq y_i$, for all $i = 1, 2, \ldots, n$. Intuitively, this property entails that replacing failing components of a functioning system by working ones cannot make the system fail. A component $i$ is relevant to the system if there is a scenario for which changing component $i$ from 0 to 1 changes the system state from not working to working. A system function is called coherent if it is monotone and all its components are relevant. Component $i$ is stronger than component $j$ if for all scenarios $x \in B$ with $x_i = x_j = 0$ we have $f(x \lor e_i) \geq f(x \lor e_j)$, where $e_i$ denotes the unit $i$th unit vector. A system function is regular if its components can be ordered such that an earlier component in this order is stronger than a later one. In this article, we examine $k$-out-of-$n$ systems, which are functional if at least $k$ (with $1 \leq k \leq n$) components are fault-free and fail if $n - k + 1$ or more components are faulty; this corresponds with a threshold function with unit weights. The relations between these different classes of system functions are shown in Figure 1.

The state of a component can only be discovered by a test of that component. The outcome of a component’s test is not known beforehand but only after the test; $p_i$ denotes the probability that component $i$ is fault-free. Consequently, the state vector is a random variable before the system diagnosis is started: $X = (X_1, X_2, \ldots, X_n)$ with each $X_i$ a Bernoulli random variable with success probability $p_i$. In this paper, we assume the variables $X_1, \ldots, X_n$ to be mutually independent.

The value $c_i > 0$ represents the cost of testing component $i$. Precedence constraints apply for the tests; these are represented by a precedence relation $A$ on $N$ (an irreflexive and transitive binary relation). When $(i, j) \in A$, component $j$ can only be tested after component $i$ has been tested. The set $\pi_i \subseteq N$ contains

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_i$</td>
<td>20</td>
<td>16</td>
<td>7</td>
<td>49</td>
<td>47</td>
</tr>
<tr>
<td>$p_i$</td>
<td>0.91</td>
<td>0.82</td>
<td>0.85</td>
<td>0.96</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Figure 2: Costs, probabilities and precedence graph of an example instance.
all the predecessors of component \( i \), that is, all components \( j \) with \((j, i) \in A\); similarly, \( \sigma_j \subset N \) is the set of successors of \( i \). The parameters and precedence graph for an example system with \( n = 5 \) are provided in Figure 2; the precedence graph depicted only contains the transitive reduction of \( A \), so no transitive arcs (such as \((1, 4)\)) are included. For the example instance, component 5 has \( \pi_5 = \{1, 2, 3\} \) and \( \sigma_5 = \emptyset \).

A schedule \( s \) for the system is a permutation of a subset of its components: it is the sequence of tests that are performed for a specific scenario. A solution to the sequential testing problem at hand is a policy (or strategy), which decides in which sequence to test which components. Formally, a policy \( \Pi \) is a function, mapping scenarios \( x \) to schedules \( s = \Pi(x) \). Let \( s_t \) denote the component in position \( t \) of the schedule \( s \). A feasible schedule \( s \) should satisfy the following condition:

\[
\forall i \in \{1, \ldots, |s|\}, \forall j \in \pi_{s_i} : \exists r < i \text{ such that } s_r = j,
\]

where \(|s|\) is the length of \( s \). In other words, a component can only be tested after all its predecessors have been tested. Preemption is not allowed in the diagnosis procedure: once started, a test cannot be interrupted until its inspection is finished. The policy continues stepwise with building a schedule until the system’s state is discovered.

Consider the five-component example system described in Figure 2: the schedule \( s_1 = (1, 2, 3, 4, 5) \) is feasible whereas \( s_2 = (3, 1, 2, 4, 5) \) is not. When \( k = 3 \) and under scenario \( x_1 = (1, 0, 0, 1, 0) \), schedule \( s_1 \) might be the output of the diagnosis procedure, where the system’s failure is discovered after testing component 5. Under a different scenario \( x_2 = (1, 1, 0, 0) \), the feasible schedule \( s_3 = (1, 2, 3) \) might be obtained: after component 3 is tested the system is known to succeed, so the diagnosis halts without testing the remaining components 4 and 5.

The flexibility inherent in a testing policy also allows to schedule different tests dependent on the outcomes of the previously conducted tests: the second component to be tested, for instance, might depend on the result (success or failure) of the first test. To capture this dynamic nature, a policy \( \Pi \) can also be represented by a binary decision tree (BDT). In such a BDT, each non-leaf node is labeled with the index of a component to be tested and has two child nodes. If the component is faulty then the left branch is entered, otherwise the right subtree is taken. Each leaf node represents a known system state: successful (s) or failed (f). Let \( L^0(\Pi) \) denote the set of leaf nodes labeled ‘f’, and \( L^1(\Pi) \) represents the set of leaf nodes labeled ‘s’. Figure 3 shows a policy for the example system with \( k = 3 \) (we need three working components for the system to function). For any node \( a \) in the BDT, there is a unique path from the root node to \( a \); let \( P(a) \) be the set of components associated with the nodes on that path (excluding \( a \)). Denote by \( P^0(a) \) (respectively \( P^1(a) \)) the subset of \( P(a) \) containing only the failed (respectively working) components. Given a scenario \( x \in \mathcal{B} \), we build a path from the root node to a leaf node in the following way: for a node labeled by component \( i \), we take the left successor if \( x_i = 0 \); otherwise we take the right successor. Therefore, for every scenario, there is a unique path terminating in a leaf node, which corresponds to the schedule generated for

![Figure 3: A policy for the example system with \( k = 3 \)](image-url)
that scenario. There may be more than one scenario for which this procedure terminates in the same leaf node, thus generating the same schedule. Define \( B(a) \) as the set of scenarios whose path ends in leaf node \( a \). In scenario \((1, 1, 0, 1, 1)\), for example, the policy sequentially tests components 1, 2, 3 and 4, after which three working components have been identified and so the policy ends at a leaf node labeled ‘s’; the same schedule is generated for scenario \((1, 1, 0, 1, 0)\).

We call a policy feasible if it only generates feasible schedules and if the inspection procedure terminates as soon as the state of the system is known. By \( F \) we denote the class of all feasible policies. The policy represented in Figure 3 is a feasible policy. We let \( C(x) \) denote the cost of schedule \( x \) for scenario \( x \). Specifically, if \( x \in B(a) \) for a leaf node \( a \), then this cost is

\[
\text{Cost}(\Pi, a) = C(\Pi, x) = \sum_{i \in P(a)} c_i.
\]

Let \( q_i = 1 - p_i \). The probability that the system is at one of the states corresponding to \( B(a) \) is

\[
\text{Prob}(\Pi, a) = \left(\prod_{i \in P^1(a)} p_i\right) \cdot \left(\prod_{i \in P^0(a)} q_i\right).
\]

The expected cost of policy \( \Pi \) is

\[
\mathbb{E}[\text{Cost}(\Pi, x)] = \sum_{a \in L^1(\Pi) \cup L^0(\Pi)} \text{Cost}(\Pi, a) \cdot \text{Prob}(\Pi, a),
\]

where \( \mathbb{E}[] \) is the expectation operator with respect to \( x \). Our objective is to find a strategy with minimal expected cost:

\[
\min_{\Pi \in F} \mathbb{E}[\text{Cost}(\Pi, x)].
\]

### 3. Related works

An extensive literature review of different types of sequential testing problems can be found in Unlüyurt (2004). Butterworth (1972) shows that the special cases of \( k = 1 \) (also called parallel system) and \( k = n \) (series system) without precedence constraints are polynomially solvable. A polynomial-time algorithm for arbitrary \( k \) was presented first by Salloum (1979), and independently by Ben-Dov (1981). Efficient implementations of this algorithm were proposed by Chang, Shi, and Fuchs (1990) (off-line algorithm requiring \( O(n^2) \) space and \( O(n^2) \) time) and by Salloum and Breuer (1997) (on-line algorithm requiring \( O(n) \) space and \( O(n \log(n)) \) time).

Jędrzejowicz (1983) shows that the problem of finding an optimal testing procedure is NP-hard for coherent systems. Cox, Qiu, and Kuehner (1989) prove that this problem remains NP-hard for general threshold functions and describe several efficient heuristics. Boros and Unlüyurt (1999) describe an optimal procedure for a certain subclass of regular systems (so-called double regular systems) that contains the class of \( k \)-out-of-\( n \) systems. With precedence constraints, the testing problem for series systems \((k = n)\) is similar to the R&D project scheduling problem in De Reyck and Leus (2008). These authors show that their problem is NP-hard under general precedence constraints; this result transfers to the related testing problem. Polynomially solvable cases have been studied when the precedence graph has a special structure. Garey (1973) shows that series systems are polynomially solvable when the precedence graph is a forest. In Chiu, Cox, and Sun (1999), an optimal diagnosis procedure is provided for certain \( k \)-out-of-\( n \) systems with parallel-chain precedence constraints. Monma and Sidney (1979) give a polynomial algorithm for series systems with a series-parallel precedence graph.
Çatay et al. (2011) report computational results for an ant-colony algorithm that solves series (and parallel) systems under general precedence constraints, and compare its performance with a branch-and-bound algorithm. To the best of our knowledge, exact solution procedures have to date not been published for general \( k \)-out-of-\( n \) systems with an arbitrary precedence graph. Çatay et al. (2011) also state that although there are a few analytical results related to testing of systems under precedence constraints available, no computational results are reported in the literature. This paper intends to fill this gap.

4. Policy classes

The most straightforward class of policies is the class \( \mathcal{F} \), which contains all feasible policies representable by a BDT. An optimal policy in this class (with minimum expected cost) is said to be globally optimal. To measure the quality of a policy class \( \mathcal{C} \) for a given \( k \)-out-of-\( n \) instance, we define the relative optimality gap \( \gamma(\mathcal{C}) \) as the relative deviation from the global optimum. If \( \xi(\mathcal{C}) \) denotes the expected cost of an optimal policy in policy class \( \mathcal{C} \), then

\[
\gamma(\mathcal{C}) = \frac{\xi(\mathcal{C}) - \xi(\mathcal{F})}{\xi(\mathcal{F})}.
\]

Note that \( \xi(\mathcal{F}) > 0 \) since all costs are positive. Below we describe two subsets of \( \mathcal{F} \).

4.1 Dominant policies

Let \( a_1 \) and \( a_2 \) be two nodes in the BDT of a feasible policy. We say that \( a_1 \) and \( a_2 \) are equivalent if the following conditions are satisfied:

\[
P(a_1) = P(a_2) \text{ and } |P^1(a_1)| = |P^1(a_2)|.
\]

Equivalent nodes can be seen to have an equivalent ‘history’, in the sense that the same set of components has been tested already and the same number of successes was obtained. A subclass of \( \mathcal{F} \) is the set \( \mathcal{D} \) of dominant policies, for which the subtrees emanating from every pair of equivalent nodes are identical. The policy depicted in Figure 3 is not a dominant policy; this can be seen by considering the equivalent nodes \( a_1 \) and \( a_2 \) (gray): the subtrees emerging from these two nodes differ in the testing order of components 4 and 5. The policy shown in Figure 5 is an example of a dominant policy.

In Section 5 we will develop a dynamic program to find an optimal dominant policy. The class \( \mathcal{D} \) of dominant policies always contains a globally optimal policy:

**Theorem 1.** There exists a dominant policy that is globally optimal, i.e., \( \gamma(\mathcal{D}) = 0 \).

**Proof:** Let \( \Pi \) be a globally optimal policy with expected cost \( \xi^* = \xi(\mathcal{F}) \) that is not dominant. We can transform \( \Pi \) into a dominant policy with an expected cost at most \( \xi^* \) as follows: choose two equivalent nodes \( a_1 \) and \( a_2 \) with different subtrees underneath. The subtree with root \( a_i \) is a BDT representation for a policy of a \( (k - |P^1(a_i)|)\)-out-of-\( (n - |P(a_i)|) \) system with expected cost \( \xi_i \), \( i = 1, 2 \). Assume that \( \xi_1 \leq \xi_2 \). By replacing the subtree with root \( a_2 \) by the subtree with root \( a_1 \), we obtain a policy with expected cost \( \xi^* - \text{Prob}(\Pi, a_1)(\xi_2 - \xi_1) \leq \xi^* \). This procedure can be repeated until all equivalent nodes have identical subtrees emerging from them.

4.2 Elementary policies

Inspired by priority policies for standard scheduling problems, we also study elementary policies, which are characterized by a total order of the component set \( N \). To ensure feasibility, the total order should be an extension of \( A \). Such an order can be represented by a list (or permutation) \( L = (l_1, l_2, \ldots, l_n) \) of the elements of \( N \). The policy follows the list and tests components one by one until the system’s state is
known. For a given scenario $x$ of a $k$-out-of-$n$ system, the elementary policy $\Pi$ characterized by a list $L$ generates a unique schedule $\Pi(x; L)$ by iterating through the list from left to right. The schedule stops when $k$ components succeed or $n-k+1$ components fail. We denote the class of elementary policies by $\mathcal{E}$. For the example system with $n = 5$ and $k = 3$, the BDT representation of the policy defined by $L = (1, 2, 3, 4, 5)$ is shown in Figure 4.

The concise representation and easy implementation are advantages of elementary policies; the main drawback is the fact that this class may not contain a globally optimal policy, i.e., there are $k$-out-of-$n$ instances with $\gamma(\mathcal{E}) > 0$. This undesirable effect is not always present: for $k = 1$ and $k = n$ the class of elementary policies coincides with the class of feasible policies. For a counterexample with $k = 2$ and $n = 3$, let $A = \{(1, 2), (1, 3)\}$, $c_1 = c_2 = c_3 = 1$ and $p_1 = 0.5$, $p_2 = \epsilon$, $p_3 = 1 - \epsilon$. A globally optimal policy achieves an expected cost of $2 + \epsilon$ when $\epsilon < 0.5$, while the two elementary policies defined by lists $(1, 2, 3)$ and $(1, 3, 2)$ both have expected cost 2.5. For the example 3-out-of-5 system presented in Section 2, an optimal elementary policy corresponds with the list $(1, 2, 3, 4, 5)$ and has objective value 63.62 (see Figure 4); a globally optimal policy for the same instance has objective value 62.23 (see Figure 5 for one such policy). An optimal elementary policy can be found by implicitly enumerating all feasible lists in a branch-and-bound framework (Section 6).

5. Dynamic programming

In this section we present a dynamic programming (DP) algorithm to find an optimal dominant policy, and briefly discuss how certain implementation issues have been tackled.

5.1 Algorithm

A state $(Y, t)$ in the dynamic program (also abbreviated as DP) represents an intermediate step in the diagnosis procedure, with $Y \subset N$ components still to be tested and $t$ successes to be achieved (with $t$ an integer between 1 and $k$). Each state $(Y, t)$ can also be seen to represent a $t$-out-of-$|Y|$ sequential-testing subproblem. The state space $\Psi$ contains all the states that can be visited during the testing procedure. A state $(Y, t) \in \Psi$ is called a feasible state and should respect the precedence constraints among the components. We require that

$$Y \in \mathcal{Y} \text{ with } \mathcal{Y} = \{Y \in 2^N : Y \neq \emptyset \text{ and } \sigma_i \subset Y \text{ for all } i \in Y\}$$

and

$$\max\{1, k - |N \setminus Y|\} \leq t \leq \min\{|Y|, k\}. \quad (2)$$

For every feasible state $(Y, t) \in \Psi$, let $E(Y) = \{i \in Y \mid \forall j \in \pi_i : j \not\in Y\}$ denote the set of components that are eligible for testing in state $(Y, t)$ (because all their predecessors have been tested); set $E(Y)$ contains the possible decisions in state $(Y, t)$. After component $i$ is tested, the subsequent state is $(Y \setminus \{i\}, t - 1)$ if component $i$ succeeds and $(Y \setminus \{i\}, t)$ if component $i$ fails. The value function $G$ of the DP computes the
minimum expected cost for \( t \)-out-of-\( |Y| \) subsystems \( Y \subseteq N, t \leq k \), which correspond to states \((Y, t)\). The value-function values for different states are related through the following recursion:

\[
G(Y, t) = \min_{i \in E(Y)} \left\{ c_i + p_i \tilde{G}(Y \setminus \{i\}, t - 1) + q_i \tilde{G}(Y \setminus \{i\}, t) \right\}, \tag{3}
\]

where \( \tilde{G} \) coincides with \( G \) on \( \Psi \) and takes the value zero otherwise. The minimum expected cost of a \( k \)-out-of-\( n \) system is given by \( G(N, k) \).

For the 3-out-of-5 system with precedence graph as in Figure 2, we present in Figure 6 the feasible states \((Y, t)\) (squared boxes) together with the eligible component sets \( E(Y) \) (gray nodes) and the state transitions (arc-node-arc connections between two consecutive states); an eligible component can either fail (dotted arc) or succeed (solid arc). The final state transitions identify the system either as working (ellipse-shaped ‘success’ node) or not working (ellipse-shaped ‘fail’ node).

The globally optimal policy for the example instance depicted in Figure 5 was found by the DP, and can be seen to be dominant. This policy tests component 4 at state \((\{4, 5\}, 1)\) and 5 at state \((\{4, 5\}, 2)\), so it is not elementary. An optimal elementary policy, shown in Figure 4, chooses component 4 for both of the two states.

### 5.2 Implementation

For the generation of the state space \( \Psi \) we need to enumerate all elements \((Y, t) \in 2^N \times \mathbb{N} \) that satisfy conditions (1) and (2) in an order that allows us to compute its value-function \( G(Y, t) \) using only value-function values of previously generated states required by recursion (3). We observe that any ordering \( \leq \Psi \) satisfying \((Y_1, t_1) \leq \Psi (Y_2, t_2) \iff |Y_1| \leq |Y_2| \) is sufficient. Such an order could be obtained by enumerating all \( Y \in 2^N \) in non-decreasing order of their cardinality \(|Y|\), thus checking each \( Y \) for condition (1). Note that this condition on \( \leq \Psi \) is independent of the \( t \)-values.

We have followed a more efficient approach proposed by Creemers, Leus, and Lambrecht (2010), who order the elements of \( \mathcal{Y} \) based on the number of successor components of the inclusion-maximal antichains (‘uniformly directed cutsets’, UDCs) of the partial order \( A \). The UDCs for the 5-component system that is partially ordered as in Figure 1 are \( U_1 = \{1, 2\} \), \( U_2 = \{2, 3\} \) and \( U_3 = \{4, 5\} \). UDCs are ordered in non-decreasing order of the number of successor components of that UDC (with arbitrary ties). In the example system \( U_3 \) has no successors, \( U_2 \) has two successors, and \( U_1 \) has three successors; therefore, the UDC-order is \((U_3, U_2, U_1)\). With a UDC \( U \), we associate a subset \( \sigma(U) \) of \( \mathcal{Y} \), as follows: \( Y \in \sigma(U) \) if \( Y \) contains all successor components of \( U \) and one (or more) components of \( U \). In order to guarantee that the \( \sigma(U) \) form a partition of \( \mathcal{Y} \) we only allow those component subsets \( Y \) into \( \sigma(U) \) for which the eligible
set $E(Y)$ is contained in $U$. For a UDC $U$, the elements of $\sigma(U)$ are ordered in non-decreasing cardinality order (ties again resolved arbitrarily). For the example we first enumerate the elements of $\sigma(U_3)$, selecting lowest-cardinality subsets first: $(\{4\}, \{5\}, \{4, 5\})$. Next, we find the order $(\{2, 4, 5\}, \{3, 4, 5\}, \{2, 3, 4, 5\})$ for $U_2$, and finally the order $(\{1, 3, 4, 5\}, \{1, 2, 3, 4, 5\})$ for $U_1$. Notice that $E(\{2, 3, 4, 5\}) = \{2, 3\} \not\subseteq U_1$, so $\{2, 3, 4, 5\} \notin \sigma(U_1)$.

The DP recursion uses a bitwise representation for a set of system components. A subset of $n$ components is represented by an array of size $\lceil n/32 \rceil$ containing 32-bit integers. A component $k$ 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.

We have used a hash table data structure enabling fast storage and lookup of the value function (for background, see Standish, 1995). Elements of a hash table are pairs consisting of a key and a value. In our DP the keys are the subsets of $N$ that satisfy (1) (in other words, the elements of $Y$), and the value corresponding to a given key $Y$ is an array holding the values $G(Y, t)$ for all $t$ that satisfy (2). 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 $g(Y) := h(Y) \mod L$ 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. 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 $g(Y) = h(Y) \mod L$ can be avoided since the result of the modulo operator coincides with the first $k$ bits of $h(Y)$. 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; in our implementation we have opted for a (standard) method called ‘separate chaining’ (Sedgewick, 2002; Standish, 1995). In Section 7 we discuss an appropriate choice for the hash function $h$ and the table size $L$ that strikes a balance between fast data access and efficient memory usage.

6. Branch-and-bound algorithm

In this section, we develop a branch-and-bound (B&B) algorithm to find an optimal policy in the class $\mathcal{E}$ of elementary policies. Each elementary policy is parameterized by a permutation of the elements of $N$, and we will enumerate all extensions of $A$ in a tree structure. At each level of the search tree, we consider all components that are eligible to be included in the corresponding position in the list. The search tree is illustrated in Figure 7 for the example instance introduced earlier.
Table 1: Computation of the values $\kappa(g, r)$ for node A in the search tree of the 3-out-of-5 example system

<table>
<thead>
<tr>
<th>$r$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g = 0$</td>
<td>1</td>
<td>$q_1$</td>
<td>$q_1q_2$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$g = 1$</td>
<td>0</td>
<td>$p_1$</td>
<td>$q_1p_2 + p_1q_2$</td>
<td>$q_1q_2p_3 + (q_1p_2 + p_1q_2)q_3$</td>
<td>0</td>
</tr>
<tr>
<td>$g = 2$</td>
<td>0</td>
<td>0</td>
<td>$p_1p_2$</td>
<td>$(q_1p_2 + p_1q_2)p_3 + p_1p_2q_3$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$\sum_g \kappa(g, r)$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$1 - q_1q_2q_3 - p_1p_2p_3$</td>
</tr>
</tbody>
</table>

As an initial upper bound, we use the output of a fast greedy heuristic procedure (Algorithm 1). We explore the enumeration tree following a depth-first strategy in order to find a feasible solution quickly, in the hope of achieving fast updates of the upper bound. The node with lowest lower bound is examined first; we discuss our lower bounds below. A comparison of the algorithmic performance of different orders of exploration of the children of a given node is provided in Section 7.2.

For each node $\nu$ in the search tree, we compute a lower bound $LB(\nu)$ on the best objective function obtainable from that node downwards. Let $l(\nu)$ represent the level in the tree at which $\nu$ is situated and $\phi(\nu)$ the set of tests added up until $l(\nu)$. The root node $R$ is at level 0, so $|\phi(\nu)| = l(\nu)$. The bound consists of two parts: $LB(\nu) = R(\nu) + R'(\nu)$, where $R(\nu)$ is the exact contribution to the objective function of the tests in $\phi(\nu)$, while $R'(\nu)$ represents a lower bound on the contribution to the objective of the tests in $N \setminus \phi(\nu)$ in any of the child nodes of $\nu$ in the enumeration tree. The partial sequence of components built to reach node $\nu$ is represented as $(j_1, j_2, \ldots, j_{l(\nu)})$.

For the computation of $R(\nu)$, we first determine the value $\kappa(g, r)$ for integer values $g$ and $r$, representing the probability that we will execute the component $j_r$ scheduled in position $r$ and that $g$ successes have been reached so far (not counting the outcome of the test of $j_r$). It can be seen that $\kappa(g, r)$ can be different from zero only for $g = 0, 1, \ldots, \min\{l(\nu) - 1; k - 1\}$ and $r = g + 1, \ldots, \min\{l(\nu); n - k + 1 + g\}$. We initialize $\kappa(0, 1) = 1$ and then recursively compute

$$\kappa(g, r + 1) = p_{j_r}\kappa(g - 1, r) + q_{j_r}\kappa(g, r).$$

This recursion is computed only for the combinations $(g, r + 1)$ that allow for non-zero $\kappa(g, r + 1)$, as specified higher. We subsequently determine

$$R(\nu) = \sum_{r = 1}^{l(\nu)} c_{j_r} \cdot \sum_g \kappa(g, r),$$

where the index $g$ sums over all non-zero $\kappa$-values. We apply this to node A in Fig. 7, with $k = 3$. The computations for $\kappa$ are written out in Table 1; note that $l(A) = n = 5$ and that $(j_1, j_2, j_3, j_4, j_5) = (1, 2, 3, 4, 5)$. When $l(\nu) = n$ then $LB(\nu) = R(\nu)$ and $R'(\nu) = 0$.

Algorithm 1 Heuristic procedure

1: $L = \emptyset$; $E =$ set of components with all predecessors in $L$;
2: Define $r_i := p_i$ if $k < \left\lfloor \frac{n}{2} \right\rfloor$, and $r_i := q_i$ otherwise;
3: while $E \neq \emptyset$ do
4: Choose a component $i$ from $E$ with smallest ratio $c_i/r_i$ and put it at the end of $L$;
5: Update $E$;
6: end while
7: Return $L$;
For the components involved in the second part \( R'(\nu) \) of \( LB(\nu) \), there is still flexibility with respect to the scheduling of the tests. We examine two alternative bounds for this part, written as \( R'_1(\nu) \) and \( R'_2(\nu) \). In \( R'_1(\nu) \), in order to underestimate the contribution to the objective function, we will underestimate the probability of incurring each of these costs separately. We therefore complete the partial sequence \( (j_1, \ldots, j_{\ell(\nu)}) \) for each component \( j \in N \setminus \phi(\nu) \) separately, sequencing all the components that are not in \( \sigma_j \cup \{j\} \) before \( j \), and then determine the probability of incurring \( c_j \) by completing the table for \( \kappa \) beyond \( r = l(\nu) \). For node \( B \) in the search tree for the example instance, this means that we use the sequence \((1, 2, 3, 4, 5)\) to evaluate the probability of incurring \( c_5 \) (exactly as in Table 1) and sequence \((1, 2, 3, 5, 4)\) for \( c_4 \) (note that \((4, 5) \notin A\) and \((5, 4) \notin A\)).

Another lower bound \( R'_2(\nu) \) is calculated by removing the precedence constraints between any two unscheduled elements in \( N \setminus \phi(\nu) \). The probability of acquiring \( g \) successful components after the test identified by node \( \nu \) is \( \kappa(g, l(\nu) + 1) \); in this case, what remains is a \((k - g)\)-out-of-\((n - l(\nu))\) subsystem. This subsystem can be solved to optimality in polynomial time using the algorithm in Chang et al. (1990). Bound \( R'_2(\nu) \) is the sum of the objective values of these subsystems, weighted with the corresponding probabilities and summed over all values \( g \) with nonzero \( \kappa(g, l(\nu) + 1) \).

We refer to the two bounds with \( R'_1 \) and \( R'_2 \) as \( LB1 \) and \( LB2 \), respectively. Bound \( LB1 \) can be found very fast, and so in our implementation we only compute \( LB2 \) when \( LB1 \) is ‘close to’ (but below) the global upper bound, in an attempt to prune unpromising nodes. The ‘closeness’ cutoff is set to 1%, which was chosen after some preliminary experiments with an eye to balancing the number of nodes and the runtime. In Section 7.2, we provide experimental evidence that computing LB2 in only part of the search nodes is better than always or never.

### 7. Computational results

In this section, we briefly discuss our overall experimental setup (Section 7.1), we provide computational comparisons of different implementations of the two algorithms (Section 7.2) and we report on the computational performance of the best implementations (Section 7.3). Finally, we further analyze a number of characteristics of the datasets and algorithms in Section 7.4.

#### 7.1 Experimental setup

All algorithms are coded in C++ using Microsoft Visual Studio 2010. The experiments are run on a Dell Laptop Latitude E6400 with an Intel Core 2 2.66-GHz processor and 4 GB RAM, equipped with Windows XP Professional Service Pack 3.

Random instance sets were generated for various values of \( n \) using the random network generator RanGen (Demeulemeester, Vanhoucke, & Herroelen, 2003). Each dataset contains 10 instances for each of the values 0.4, 0.6 and 0.8 of the network density parameter order strength \( OS \) (the number of comparable components pairs divided by the maximum number \( n(n - 1)/2 \) of such pairs, see Mastor, 1970). The costs of the components are generated as independent realizations of a discrete uniform integer random variable on \([0, 50]\), and success probabilities are chosen randomly from \([0.2, 0.2, 0.8]\) and \([0.8, 1]\), leading to 30 instances for each combination of \( OS \) and \( n \). All data are available on-line\(^1\).

#### 7.2 Implementation choices for the algorithms

##### 7.2.1 Dynamic programming

As explained in Section 5.2, the keys \( Y \in \mathcal{Y} \) of the hash table are represented by an array of integers \((y_0, \ldots, y_l)\) of size \( l + 1 = \lceil n/32 \rceil \). A straightforward hash function \( h_s(Y) = \sum_{i=0}^{l} y_i \) results in equal

Table 2: Average number of lookups to retrieve an element from the hash table and corresponding CPU times for a 25-out-of-50 system

<table>
<thead>
<tr>
<th></th>
<th>OS = 0.4</th>
<th>OS = 0.6</th>
<th>OS = 0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lookups</td>
<td>CPU</td>
<td>lookups</td>
</tr>
<tr>
<td>$h_s \mod L$</td>
<td>526</td>
<td>551.39</td>
<td>294</td>
</tr>
<tr>
<td>$h_m \mod L$</td>
<td>463</td>
<td>497.27</td>
<td>292</td>
</tr>
<tr>
<td>$(h_w \circ h_s) \mod L$</td>
<td>1.9</td>
<td>23.12</td>
<td>1.7</td>
</tr>
<tr>
<td>$(h_w \circ h_m) \mod L$</td>
<td>1.8</td>
<td>20.6</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Figure 8: Average number of lookups as a function of the hash-table length (scaled) for a 25-out-of-50 system

hash values for different permutations of the components of $Y$. Multiplication with a position-dependent power of a large prime number $M$, resulting in a hash function $h_m(Y) = \sum_{i=0}^{t} y_i M^i$, prevents this from happening, and is a common hash function used for strings (Knuth, 1998). The first two lines of Table 2 show the performance of hash functions $h_s \mod L$ and $h_m \mod L$ for a 25-out-of-50 system with different order strengths. The size $L$ of the hash table is chosen to be the largest power of two smaller than $|Y|$; we will motivate this choice below. Besides CPU times, the table also contains the average number of lookups needed to retrieve an element from the hash table. A good hash function only needs a low number of calls to retrieve an element from the table, i.e., it results in a low number of collisions. We observe that both $h_s$ and $h_m$ turn out to be rather poor hash functions.

The quality of the hash function 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 function we tested is Wang’s (2007) “32-bit Mix Function” $h_w$. From the last two lines in Table 2, we see that the number of lookups and CPU times significantly decrease when $h_w$ is applied after hash functions $h_s$ and $h_m$. In all remaining computations, we will use the implementation of $(h_w \circ h_m) \mod L$.

To determine an appropriate hash-table size $L$, we have plotted in Figure 8 the number of lookups as a function of $L/|Y|$. We conclude that a table size at least half the size of the state space is sufficient, because the improvement for larger sizes is only minor. In the remainder of this text, all results therefore pertain to $L$ being chosen as the largest power of two smaller than $|Y|$ (and thus between $|Y|/2$ and $|Y|$).
7.2.2 Branch and bound

We first examine different options for the order of exploration of the eligible components from a given search node. With the combined use of the two lower bounds as described in Section 6, Figures 9 and 10 depict the average number of nodes in the search tree and the average runtime for three different orders of exploration; ‘c/q’ stands for lowest ratio $c_i/q_i$ first, ‘c/p’ for lowest $c_i/p_i$ first, and ‘bound’ gives priority to the nodes with best available lower bound (either LB1 or LB2). All these results are averages for the ten instances with $n = 10$, $OS = 0.4$ and all $p_i \in [0.2, 0.8]$. We find that the combined lower bound contains the best information for lowering the runtimes.

We have also explored three different lower-bounding strategies: only LB1, always LB2, and the combination of LB1 and LB2 described in Section 6. The results can be found in Figures 11 and 12 (these results pertain to the same instance set as Figures 9 and 10). Based on these figures, we decide to use the combined bounding strategy in the remainder of our experiments.

7.3 Algorithmic performance

Table 3 summarizes the performance of the DP algorithm. We have examined the two values 5 and $\lfloor n/2 \rfloor$ for $k$. We have tested 10 instances per combination of $n$, $k$, $OS$ and the probabilities. The instances for given $n$, $k$ and $OS$ corresponding with different probability ranges are gathered in one cell of 30 instances. All CPU times are reported in seconds, and the columns labeled ‘#’ indicate the number of instances solved to guaranteed optimality for the corresponding settings. The unsolved instances in Table 3 were all interrupted due to insufficient memory. We observe that higher order strength (a denser precedence graph) leads to lower CPU times, which is also intuitive. This is confirmed by Table 4, where we report the average size of the state space for different system size $n$ and order strength $OS$; all the data in Table 4 pertain to $(n/2)$-out-of-$n$ instances. When $OS = 0.8$, all instances with 120 components per system were solved to guaranteed
Table 3: Average CPU times used by DP for different values of \( n \), OS and \( k \)

<table>
<thead>
<tr>
<th>( OS = 0.8 )</th>
<th>( OS = 0.6 )</th>
<th>( OS = 0.4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = 5 )</td>
<td>( k = \frac{n}{2} )</td>
<td>( k = 5 )</td>
</tr>
<tr>
<td>( n )</td>
<td>CPU #</td>
<td>CPU #</td>
</tr>
<tr>
<td>10</td>
<td>0.00 30 0.00 30</td>
<td>0.00 30 0.00 30</td>
</tr>
<tr>
<td>20</td>
<td>0.00 30 0.00 30</td>
<td>0.00 30 0.01 30</td>
</tr>
<tr>
<td>30</td>
<td>0.00 30 0.00 30</td>
<td>0.03 30 0.05 30</td>
</tr>
<tr>
<td>50</td>
<td>0.00 0.14 0.01 30</td>
<td>0.14 30 0.35 30</td>
</tr>
<tr>
<td>60</td>
<td>0.01 0.30 0.02 30</td>
<td>0.6 30 1.65 30</td>
</tr>
<tr>
<td>70</td>
<td>0.02 0.07 0.03 30</td>
<td>2.94 30 9.46 30</td>
</tr>
<tr>
<td>80</td>
<td>0.06 0.17 0.18 30</td>
<td>10.98 30 40.18 30</td>
</tr>
<tr>
<td>90</td>
<td>0.14 0.49 0.49 30</td>
<td>40.67 30 77.5 3</td>
</tr>
<tr>
<td>100</td>
<td>0.32 1.28 1.28 30</td>
<td>107.92 3 0 0</td>
</tr>
<tr>
<td>110</td>
<td>0.69 2.56 2.56 30</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>120</td>
<td>1.45 6.46 6.46 30</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>

Table 4: Average size of the state space for different values of \( n \) and OS (\( k = n/2 \))

<table>
<thead>
<tr>
<th>( OS )</th>
<th>( n = 30 )</th>
<th>( n = 60 )</th>
<th>( n = 90 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>2294</td>
<td>64704</td>
<td>816523</td>
</tr>
<tr>
<td>0.6</td>
<td>19080</td>
<td>3060521</td>
<td>110121539</td>
</tr>
<tr>
<td>0.4</td>
<td>168411</td>
<td>148507840</td>
<td>-</td>
</tr>
</tbody>
</table>

The results of the B&B algorithm are summarized in Table 5. Its stopping criterion is a time limit of 15 minutes (the time limit for the DP was also 15 minutes, but this was never reached). Overall, despite the lower number of feasible policies scanned (all elementary policies are dominant, but not vice versa), the B&B-algorithm turns out to be quite restricted in the size of the instances solved compared to the DP. Again, runtimes increase with decreasing order strength; the same can be deduced from Table 6, in which the average number of nodes visited by the B&B-algorithm for \( k = 5 \) is shown for different OS-values. We also deduce from Table 6 that the number of explored search nodes grows rapidly with increasing \( n \).

7.4 Analysis

The optimal expected cost and the CPU time for the DP algorithm as a function of \( k \) are presented in Figure 13, for different probability ranges for a 20-component system. We observe that both the objective as well as the runtimes initially increase until a maximal value is reached, and subsequently decrease. The location of the maximum in the optimal expected cost depends on the probability ranges, whereas the corresponding curves for the runtimes almost coincide. We conjecture that the runtimes for different probability ranges are very similar because they are proportional with the size of the state space, and thus independent of the choice of the probabilities. The CPU times become larger when \( k \) approaches \( n/2 \). This is to be expected because the state space of a \( k \)-out-of-\( n \) system is the largest when \( k \) equals \( n/2 \) (this is easily derived from (2)). The observed phase transition (Hogg, 1996; Monasson, Zecchina, Kirkpatrick, Selman, & Troyansky, 1999) displays a very clear easy-hard-easy pattern.

The shifted cost curves correspond to the shifts in Figure 14, which shows the probability that the optimality; this is only true up to \( n = 50 \) when \( OS = 0.4 \).

The shifted cost curves correspond to the shifts in Figure 14, which shows the probability that the
Table 5: Average CPU times used by B&B for different values of \( n \), \( OS \) and \( k \)

<table>
<thead>
<tr>
<th>( OS = 0.8 )</th>
<th>( OS = 0.6 )</th>
<th>( OS = 0.4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k = 5 )</td>
<td>( k = \frac{n}{2} )</td>
<td>( k = 5 )</td>
</tr>
<tr>
<td>( n )</td>
<td>CPU</td>
<td>#</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>0.07</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>0.02</td>
<td>20</td>
</tr>
<tr>
<td>40</td>
<td>190.82</td>
<td>20</td>
</tr>
<tr>
<td>50</td>
<td>4.85</td>
<td>10</td>
</tr>
<tr>
<td>60</td>
<td>43.78</td>
<td>10</td>
</tr>
<tr>
<td>70</td>
<td>100.12</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6: Average number of search nodes explored by the B&B algorithm for different values of \( n \) and \( OS \) (\( k = 5 \))

<table>
<thead>
<tr>
<th>( OS )</th>
<th>( n = 10 )</th>
<th>( n = 50 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>216</td>
<td>10968043</td>
</tr>
<tr>
<td>0.6</td>
<td>805</td>
<td>35221397</td>
</tr>
<tr>
<td>0.4</td>
<td>7731</td>
<td>295722130</td>
</tr>
</tbody>
</table>

Figure 13: Optimal expected cost (left) and required runtime (right) for the DP as a function of \( k \) for different probability intervals of a system with \( n = 20 \) components

The system is functional for the three different interval ranges (in literature this is frequently called the yield), again dependent on \( k \). When the yield is close to 1, the optimal strategy will start testing components that have high probabilities of success and stop when the system is successful, thus reducing the total expected cost of identifying the system’s state; symmetrically, when the yield is close to 0, an optimal strategy will seek to confirm system failure as quickly as possible. When the yields are around 0.5, the system ex ante is neither prone to be functional nor not working, and the minimum total expected cost will be higher than for the other cases. This intuitive argument is confirmed by the left picture in Figure 13. Chang et al. (1990) observe a similar behavior for the \( k \)-out-of-\( n \) testing problem without precedence constraints.

In Figure 15, the optimal costs and CPU times of the same 20-component system are now plotted for the B&B algorithm. The cost curves have the same shape as for the DP, although the optimal expected costs are slightly higher because we optimize over the smaller class of elementary policies (\( E \subset D \)). Contrary to DP, the CPU time for B&B as a function of \( k \) for a given system follows a pattern (phase transition) that roughly resembles that of the expected cost, and so is dependent on the probability ranges. Since the structure of the
search tree is not affected by changing $k$, we conclude that there must be differences in the strength of the lower bounds.

The average relative optimality gap $\gamma(\mathcal{E})$ between DP and B&B of the 90 different 5-out-of-10 systems is 0.7%. We observe that, for our data, the subclass of elementary policies does not always contain a globally optimal policy but does retain high-quality solutions on average. Figure 16 displays the optimality gap of systems with $n = 10$ and $p_i \in [0.2, 0.8]$ for different order strengths, as a function of $k$. The gap tends to grow with decreasing order strength and when $k$ moves closer to $n/2$. The maximum gap is reached at about the same $k$-value as the maximum expected cost, so dependent on the probability intervals.

8. Summary

In this paper, we have examined different classes of testing policies for $k$-out-of-$n$ systems under general precedence constraints. The class of dominant policies always contains a globally optimal member and is solved to optimality by dynamic programming; the class of elementary policies does not always contain a globally optimal policy, and an optimal member of this class is found by a branch-and-bound algorithm. Our dynamic program is quite efficient and can solve large and complex instances, but is unable to solve certain instances whose requirement exceeds our computer memory. The elementary policies have the advantage of a compact representation, and the branch-and-bound algorithm does not have the limitation of memory issues, but it is limited in the size of the instances that can be solved to guaranteed optimality within reasonable time limits.
Figure 16: Relative optimality gap of a $k$-out-of-10 system

References


