Exact and Heuristic Approaches to Detect Failures in Failed k-out-of-n Systems

Tonguc Yavuz[a], O. Erhun Kundakcioglu[a], Tonguç Ünlüyurt[b]

[a] *Ozyegin University, Istanbul, Turkey.*
[b] *Sabanci University, Orhanli, Tuzla, Istanbul, Turkey.*

Abstract

This paper considers a k-out-of-n system that has just failed. There is an associated cost of testing each component. In addition, we have apriori information regarding the probabilities that a certain set of components is the reason for the failure. The goal is to identify the subset of components that have caused the failure with the minimum expected cost. In this work, we provide exact and approximate policies that detects components' states in a failed k-out-of-n system. We propose two integer programming (IP) formulations, two novel Markov decision process (MDP) based approaches, and two heuristic algorithms. We show the limitations of exact algorithms and effectiveness of proposed heuristic approaches on a set of randomly generated test instances. Despite longer CPU times, IP formulations are flexible in incorporating further restrictions such as test precedence relationships, if need be. Numerical results illustrate that dynamic programming for the proposed MDP model is the most effective exact method, solving up to 12 components within one hour. The heuristic algorithms' performances are presented against exact approaches for small to medium sized instances and against a lower bound for larger instances.

Keywords: k-out-of-n systems, fault detection, integer programing, Markov decision processes, dynamic programing

1. Introduction and Literature Review

Systems have become more and more complicated, containing an increasing number of components, sensors, mechanical, and electronic sub-systems. In order for a system to be reliable, there is typically some sort of redundancy that is built in the system (Heydari and Sullivan 2018). That is, we do not need all components or sub-systems to function for the whole system to function. The main goal is to meet the functional requirements of the system without disruption. However, some components cannot meet their functional requirements over time, due to wear and tear, deteriorating external conditions, aging, or inappropriate usage. When some components fail, the whole system *may* fail depending on the set of failed components. Failures are inevitable and in case of a failure, the goal is to revive the system in the shortest possible time and/or with the minimum average cost. This requires *identification* of the subset of components that have caused the system failure. In this study, we focus on k-out-of-n systems. That is, the system works when at least k components out of n components work. In other words, the system fails when $n - k + 1$-st component fails. These systems have many applications. One particular reason for this wide range of applications is that k-out-of-n systems provide a practical redundancy that is easy to describe and implement.

k-out-of-n systems have been widely studied in the literature in the context of reliability, maintenance, and sequential testing (see e.g., (Krishnamoorthy and Ushakumari 2001, Flyyn and Chung 2004, Amari et al. 2004, Eryilmaz and Devrim 2019)). Two similar variants of the framework proposed in this work focus on finding out (i) whether the system works or fails (*sequential testing problem* or *function evaluation problem*) and (ii) the reason of failure (*failure detection problem*). In both problems, a feasible solution can be described as a binary decision tree and the goal is to minimize the expected cost. On the other hand, there are fundamental differences between the two problems. First of all, the probabilities regarding the results of tests change as tests are executed in the failure detection problem, whereas they are the same throughout the process in sequential testing problem. Various extensions of these problems have

Email addresses: tonguc.yavuz@ozu.edu.tr (Tonguc Yavuz), erhun.kundakcioglu@ozyegin.edu.tr (O. Erhun Kundakcioglu), tonguc@sabanciuniv.edu (Tonguç Ünlüyurt)

been studied in the literature such as the case with precedence constraints among tests (see e.g., (Wei et al. 2013), the case when tests can be executed in batches (see. e.g. (Daldal et al. 2017)), and the case when tests are not perfect (see e.g., (Wei et al. 2017)). n-out-of-n (i.e., series system) has attracted special attention due to its simplicity and typically extensions are studied for series systems first (see e.g., (Kovalyov et al. 2006)). Series systems also appear in many practical applications. Another difference is that in the failure detection problem the output is the set of components causing the failure, whereas in the sequential testing problem the output is the information that the system is working or failed along with a proof of that status. Next we review studies on sequential testing and failure detection problems.

Chang et al. (1990) study k-out-of-n systems for the minimum cost diagnosis of electronic wafers in the context of sequential testing and provide a polynomial time exact algorithm. Actually, this algorithm was previously proposed in (Ben-Dov 1981) with an incomplete proof. The basic idea of this algorithm dates back to (Butterworth 1972). Kang and Kim (2012) propose k-out-of-n systems to provide redundancy in sub-systems of a nuclear reactor for reliable operations. Such systems also appear in (Faghih-Roohi et al. 2014) in natural gas pipeline and energy production systems related applications where reliability is crucial. Sequential testing problem for k-out-of-n systems with imperfect tests is studied in (Wei et al. 2017). In this study, testing stops when a certain precision threshold for the correct diagnosis is reached as it is not always possible to find the exact state of the system due to results obtained from imperfect tests. In (Wei et al. 2017), heuristic algorithms are proposed when tests are imperfect and there are precedence constraints among tests for k-out-of-n systems. The case with precedence constraints have recently been studied in (Rostami et al. 2019) for n-out-of-n systems where exact methods are proposed to solve the problem by using structural results for speed up. Barron (2018) uses renewal theory to analyze the long time average cost for certain maintenance policies for k-out-of-n systems. Another line of related research considers general discrete functions described by a table rather than k-out-of-n functions. Each row of the table corresponds to a failure class depending on the states of the components and the goal is to find the correct failure class with minimum expected cost. Different variations of this problem are widely studied in the literature (see e.g., (Tu and Pattipati 2003, Kundakcioglu and Ünlüyurt 2007, Zhang et al. 2013)). Sequential testing problem has also been studied for more general systems than k-out-of-n systems in the literature. Typically, in these studies, the underlying function that determines the state of the system is a special monotone Boolean function (threshold systems, series-parallel systems etc.). Special cases in terms of data such as uniform costs and/or probabilities have also attracted attention. Researchers have proposed exact, approximate and heuristic approaches for different versions of the problem. A general review can be found in (Ünlüyurt 2004).

Failure detection problem is studied for series systems (i.e., n-out-of-n systems) in (Nachlas et al. 1990). An optimal algorithm is provided when tests are perfect and a heuristic algorithm is provided when tests are not perfect. The proof in (Nachlas et al. 1990) is corrected later in (Canfield and Nachlas 1991). (Shahmoradi and Ünlüyurt 2018) study the same problem for series systems when tests are unreliable. In this work, the model allows repetition of tests at most once after a positive result and the goal is the minimize the sum of expected inspection and misclassification costs. In (Ait-Kadi et al. 2018), the authors consider a series system that has failed for an embedded system and try to localize the reason of failure. This work assumes that there is a set of tests that can localize faults for a subset of the components. Failure detection problem is also studied in (Wang et al. 2012) in the context of wireless sensor networks for series systems and certain other systems corresponding to special network topologies. A similar application is considered by Bao et al. (2016), who extend the model by incorporating some additional practical considerations. In (Garshasbi 2016), ant colony optimization is used for fault localization in computer networks. The problem of identifying the source of failure after observing symptoms/alarms is considered in (Cheng and Wu 2016). Although the problem is similar to our general framework, in this paper, the authors propose machine learning methods to localize faults with the best precision when false alarms are possible.

Another line of research that is similar to failure detection problem of k-out-of-$n$systems, namely *discrete search problem*, aims to find an item that is hidden in one of the N boxes with the minimum expected cost. There are a number of variations of this problem. It is considered costly to check the boxes and the probability that the item is in a box is known apriori. This is essentially quite similar to the failure detection problem for series systems. For example, Kress et al. (2008) provide an optimal greedy algorithm for the search problem when only false positive results are possible. The problem is motivated by a homeland security application. Wagner and Davis (2001) consider a variation of the problem when there are simple precedence constraints and path dependent relationships are defined by group activities. The problem of maximizing the probability of finding the hidden object, where search is performed with limited number of sensors is considered in (Song and Teneketzis 2004). Another variation is considered in (Kadane 2015), where the search can be executed with different technologies with different costs and overlook probabilities. A

generalization of this problem considers finding k objects hidden in n boxes where searching each box is costly. This is very similar to failure detection problem for a k-out-of-n system, where the goal is minimizing cost. The difference is in the objective function, because probabilities are not a part of the input in this problem. Lidbetter (2013) and Lidbetter and Lin (2019) use game theoretic approaches to tackle such problems.

Although k-out-of-n systems are widely studied in the context of maintenance and reliability as failure detection or discrete search problems, exact approaches for *failure detection* in k-out-of-n systems have not been studied in the literature", to the best of our knowledge.

The contributions of this paper can be summarized as follows:

a) We introduce and study the failure detection problem for k-out-of-n system generalizing the n-out-of-n systems studied in the literature.
b) We provide four exact, two heuristic approaches to solve the problem together with two lower bounding schemes for benchmarking at larger instances.
c) To the best of our knowledge, integer programming modeling and Markov decision processes are proposed for the first time to solve this class of problems.
d) We conduct numerical experiments to assess the effectiveness of different approaches.

The remainder of this paper is organized as follows: We explain basics of k-out-of-n systems and our assumptions in Section 2. Section 3 presents four exact methods (two integer programming and two Markov decision process based) and two heuristic approaches to solve the problem. Computational experiments are presented in Section 4, where we compare the performance of all proposed algorithms in terms of quality and running time. Finally, in Section 5 we provide concluding remarks and directions for future research.

2. Problem Definition

We consider a system consisting of n components. The system operates as long as a minimum of k components operate. If at least $n - k + 1$ components fail, the system goes out of order. This type of systems, referred to as k-out-of-n systems, are studied in different applications areas with different objectives in the literature. We assume that the lifetimes of the components are continuous random variables. Consequently, at the time of the failure we know that exactly $n - k + 1$ components are in failed state. Note that, as the exact number of failed and working components are known, when the system fails, the total number of possibilities for the states of all the components is $\binom{n}{n-k+1}$. Our main goal is to figure out the exact system state with the minimum expected cost as will be explained later.

The prior probability that component i works (fails) when the system is down is denoted by p_i (q_i). This might be derived from technical specs of the components or estimated by using historical data. Despite the fact that prior failure probabilities are independent, posterior probabilities depend on the tests conducted.

For each subset of size $n - k + 1$, we can compute the probability that the given subset of components is the reason for the failure. From this information, one can easily compute the probability that a certain component functions or fails, denoted by p_i and q_i, respectively. As the tests are conducted and results of the tests are obtained, it is necessary to update these probabilities. For instance, in an n-out-of-n system with nonzero prior failure probabilities, if the first test reveals a failure for a component, posterior failure probabilities for remaining components drop to zero.

The cost of testing component i is c_i. This might be actual cost or time required for the test or a combination of these. It is assumed that the tests provide perfect information regarding the states of the components.

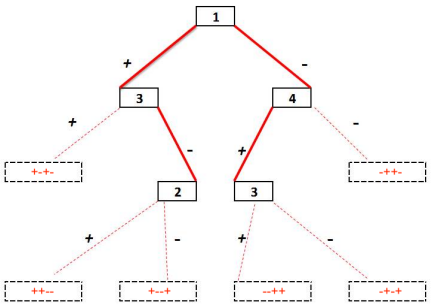
In order to figure out the exact state of all components, we should continue testing until we find all the $n - k + 1$ failed components or $k - 1$ functioning components. So a solution to the problem is a strategy that outputs the next component to test or the set of failed components, given the results of previously executed tests. In this work, we would seek to find the strategy with minimum expected cost among all strategies.

2.1. Binary Decision Trees

A solution for the problem is an algorithm that outputs the next component to test or the set of failed components, given the results of previously executed tests. Therefore, a solution strategy (testing policy) can naturally be represented by a binary decision tree. The *root and intermediate nodes* indicate testing a component. The leaf nodes represent possible system states. In a leaf node, the states of all components are known. Given k and n for a k-out-of-n

system, the binary decision tree that corresponds to a solution has a certain structure that depends only on k and n. The only difference between solutions are the node labels, i.e., the tests to be executed at each stage. Figure 1 shows a feasible solution for a 3-out-of-4 system. Each intermediate node in the tree corresponds to a component that is tested in that step. The left and right arcs correspond to the cases where the tested component works (+) or fails (−), respectively. *Leaf nodes* (dashed boxes) indicate the final state of the system describing the set of failed components.

Figure 1: A sample binary decision tree (3-out-of-4). According to this strategy, component 1 is tested first. If component 1 functions, component 3 is tested. Otherwise, component 4 is tested.



In this study, we focus on exact solution methods for **adaptive strategies**, represented by binary decision trees, where next component to inspect may depend on the results of previous tests. It is also possible to consider **permutation strategies** where the next component to inspect does not depend on the results of previous tests. Permutation strategies are easy to describe since we do not have to provide a binary decision tree. Thus, we utilize permutation strategies in our heuristic approaches, where a solution is described as a permutation of the components. In general, an optimal solution may or may not be a permutation strategy.

The leaf nodes in the decision tree correspond to subsets of components of size $n - k + 1$ that have failed. In other words, when we execute tests according to a binary decision tree and we reach a leaf node, the states of all components are known. Note that, a feasible test policy tests a component at most once on any path from the root node to a leaf node.

2.2. Computing the Expected Cost for a Strategy: A Numerical Example

Consider the decision tree for 3-out-of-4 system in Figure 1. Suppose the prior probabilities are as in Table 1.

Table 1: Prior probabilities and test cost of components

Component	Failure Probability ($q = 1 - p$)	Test Cost
1	0.1	10
2	0.2	20
3	0.3	30
4	0.4	40

There are 6 possible outcomes (leaf nodes of decision tree) for this system as highlighted in Table 2.

Table 2: Outcome probabilities

State ID	C_1	C_2	C_3	C_4	Prior Probability	Posterior Probability
1	+	+	-	-	0.0864	0.0864/0.2144
2	+	-	+	-	0.0504	0.0504/0.2144
3	+	-	-	+	0.0324	0.0324/0.2144
4	-	+	+	-	0.0224	0.0224/0.2144
5	-	+	-	+	0.0144	0.0144/0.2144
6	-	-	+	+	0.0084	0.0084/0.2144

For instance, the prior probability for first state can be calculated using $p_1 p_2 q_3 q_4 = 0.086$. More importantly, the conditional probability of that state is the probability of being in first state given that the system has failed (denoted by F) can be calculated. The details of the calculations are as follows:

$$P(++--|F) = \frac{P(++--, F)}{P(F)}$$

$$= \frac{0.0864}{0.0864 + 0.0504 + 0.0324 + 0.0224 + 0.0144 + 0.0084} = \frac{0.086}{0.2144}$$

We can calculate the expected cost for a strategy by multiplying the cost of reaching an outcome by associated conditional probabilities. For instance, the cost of reaching state 1, leaf node $(++--)$ in the bottom left corner in Figure 1, is the sum of test costs for 1, 3, and 2. A detailed breakdown of cost for reaching each leaf and associated probabilities are presented in Table 3.

Table 3: Expected cost for the strategy presented in Figure 1.

State ID	Tested Components	Conditional Probabilities	Cost	Expected Cost
1	1,3,2	0.0864/0.2144	60	24.18
2	1,3	0.0504/0.2144	30	7.05
3	1,3,2	0.0324/0.2144	60	9.07
4	1,4	0.0224/0.2144	50	5.22
5	1,4,3	0.0144/0.2144	80	5.37
6	1,4,3	0.0084/0.2144	80	3.13
			Sum:	54.02

2.3. Leaf Nodes

There are $\binom{n}{n-k+1}$ leaf nodes in a failed k-out-of-n system. We need to create system states for all leaf nodes together with associated posterior probabilities as in Table 2. Without these probabilities, we cannot calculate the expected cost for a strategy. Note that this mandatory process of creating possible system states can be time and memory consuming for larger instances (see Table 4).

Table 4: Number of leaf nodes created for a given total number of components (n) in each row and number of failed components ($n-k+1$) in each column.

n \ $n-k+1$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
3	3														
4	4	6													
5	5	10													
6	6	15	20												
7	7	21	35												
8	8	28	56	70											
9	9	36	84	126											
10	10	45	120	210	252										
11	11	55	165	330	462										
12	12	66	220	495	792	924									
13	13	78	286	715	1,287	1,716									
14	14	91	364	1,001	2,002	3,003	3,432								
15	15	105	455	1,365	3,003	5,005	6,435								
16	16	120	560	1,820	4,368	8,008	11,440	12,870							
17	17	136	680	2,380	6,188	12,376	19,448	24,310							
18	18	153	816	3,060	8,568	18,564	31,824	43,758	48,620						
19	19	171	969	3,876	11,628	27,132	50,388	75,582	92,378						
20	20	190	1,140	4,845	15,504	38,760	77,520	125,970	167,960	184,756					
21	21	210	1,330	5,985	20,349	54,264	116,280	203,490	293,930	352,716					
22	22	231	1,540	7,315	26,334	74,613	170,544	319,770	497,420	646,646	705,432				
23	23	253	1,771	8,855	33,649	100,947	245,157	490,314	817,190	1,144,066	1,352,078				
24	24	276	2,024	10,626	42,504	134,596	346,104	735,471	1,307,504	1,961,256	2,496,144	2,704,156			
25	25	300	2,300	12,650	53,130	177,100	480,700	1,081,575	2,042,975	3,268,760	4,457,400	5,200,300			
26	26	325	2,600	14,950	65,780	230,230	657,800	1,562,275	3,124,550	5,311,735	7,726,160	9,657,700	10,400,600		
27	27	351	2,925	17,550	80,730	296,010	888,030	2,220,075	4,686,825	8,436,285	13,037,895	OoM	OoM		
28	28	378	3,276	20,475	98,280	376,740	1,184,040	3,108,105	6,906,900	13,123,110	OoM	OoM	OoM	OoM	
29	29	406	3,654	23,751	118,755	475,020	1,560,780	4,292,145	10,015,005	OoM	OoM	OoM	OoM	OoM	
30	30	435	4,060	27,405	142,506	593,775	2,035,800	5,852,925	OoM	OoM	OoM	OoM	OoM	OoM	OoM

The largest instance created is 19-out-of-28 with 13,123,110 leaf nodes, which takes less than 6 seconds to create with the computer specified in Section 4.

Given n, k, and the component failure probabilities, we have to calculate posterior outcome probabilities to calculate the total cost of a strategy. The number of outcomes (leaf nodes) is $O(n^k)$, i.e., not polynomial. One might

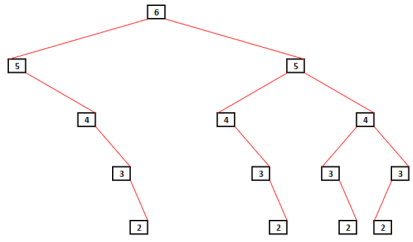
consider calculating the cost of the tree over different levels, multiplying the cost of the tree with the associated probabilities at that level. However, the test outcome probabilities depend on already tested components' outcomes, as the system is known to fail. Therefore, it is computationally challenging to even verify a solution in general k-out-of-n systems that is known to fail. One special case is the polynomially solvable n-out-of-n that is failed (or equivalently a failed 2-out-of-n) case, where there are only one failed (or working) component and a total of $O(n)$ states.

One theoretically interesting question is if finding the *next test to perform* is $\mathcal{NP}$-complete. In practice, this would be valuable as the maximum number of tests to perform is $O(n)$. If the best test to perform in each iteration can be found in polynomial time, the running time of the algorithm would still be polynomial, despite the lack of an entire decision tree.

3. Solution Methods

In this section, we explain our solution approaches for solving the problem. When we enumerate all possible solution strategies and select the one with minimum expected cost, 2-out-of-6 system can be solved in around 30 minutes whereas 3-out-of-6 system takes more than 8 hours to solve. This exponential growth in solution time for a *brute-force* approach is somewhat expected, as there are 14 intermediate, 15 leaf nodes, and more than 12 million feasible solutions for a 3-out-of-6 system. Figure 2 shows the general structure of the tree, where each intermediate node is labeled with the number of tests that can be performed, hence the number of feasible solutions $6^1 \times 5^2 \times 4^3 \times 3^4 \times 2^4 = 12,441,600$.

Figure 2: General layout for the decision tree of a 3-out-of-6 system that failed. That means there are 2 working components, thus two left branches or four right branches suffice. The numbers on the intermediate nodes indicate the number of alternative components that can be tested.



Because brute-force is impractical, we propose four exact solution approaches for solving small to medium-sized problems. We also propose two heuristic approaches for fast near-optimal solutions. These simple yet effective algorithms are able to solve the problem for not only larger systems, but also more general k-out-of-n systems with additional constraints.

3.1. Integer Programming Approaches

For a given k and n, we consider the binary decision tree, whose structure is fixed. We take advantage of the fact that the number and positioning of the leaf and intermediate nodes are known. Essentially, our models try to assign each intermediate node of the binary decision tree with a component in a feasible way. In different models, we define our decision variables in different ways to accomplish this with the minimum expected cost. In the first approach, leaf node based IP, we assign sequences to paths in the decision tree. In the second approach, intermediate node based IP, we assign leaf nodes to paths and components to intermediate nodes.

3.1.1. Leaf Node Based IP

We need the following the formal definitions, before we present our leaf node based IP formulation.

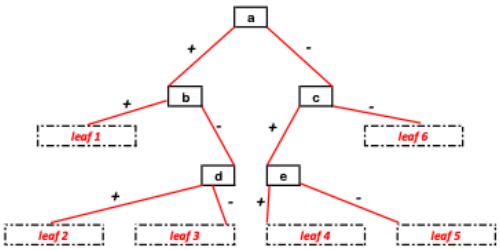
Path: The path of *nodes* to the leaves. The path to each leaf node includes the root node and intermediate nodes but excludes the leaf node. For instance, there are a total of 4 paths in a 3-out-of-4 system as in Figure 3:

Path 1: The leftmost *test pair*, (nodes a & b),
Path 2: *Test triplet* that results with consecutive + and − outcomes and leads to leaf nodes 2 and 3, (nodes a, b, d)
Path 3: *Test triplet* that results with consecutive − and + outcomes and leads to leaf nodes 4 and 5 (nodes a, c, e)

Path 4: The rightmost *test pair*, i.e., consecutive − and − outcomes, (nodes a & c).

Some paths lead to one leaf node (such as paths 1 and 4 above), whereas some lead to two (e.g., paths 2 and 3).

Figure 3: General structure for the decision tree of a 3-out-of-4 system



Sequence: The ordered list of components to be tested, that are candidates to be assigned to *paths* to leaf nodes (e.g., sequence 1, 4, 3 and sequence 1, 3 in the sample solution in Fig. 1).

Note that the path lengths vary from leaf node to leaf node. Given a k-out-of-n problem, we draw the corresponding decision tree first. The length of the paths is between $\min(n-k+1, k-1)$ and $n-1$. After the set of all nodes (N) and the set of all paths (P) are generated, we create the set of sequences (S). We present the parameters and the decision variables from Table 5.

Table 5: The parameters and the decision variable for leaf node based IP

Parameters	
$a_{p,s}$:	1 if sequence s is suitable for path p (i.e., they have the same length) 0 otherwise
$c_{p,s}$:	cost of sequence s in path p
v_s^n:	n-th component tested in sequence s
Decision Variables	
$x_{p,s}$:	1 if sequence s is assigned for path p 0 otherwise

For each sequence and compatible path of appropriate length, this cost is calculated by multiplying the total cost of tests in the sequence and conditional probability of reached output state at the end of the path.

Formulation

$$\min \quad \sum_{p\in P}\sum_{s\in S} c_{p,s}x_{p,s} \tag{1a}$$

$$\text{s.t.} \quad \sum_{s\in S} a_{p,s}x_{p,s} = 1, \qquad \forall p \in P \tag{1b}$$

$$\sum_{s\in S} v_s^n x_{p,s} - \sum_{\forall s\in S} v_s^n x_{p',s} = 0, \qquad \forall p, p' \in P, \forall n : p^{(n)} = p'^{(n)} \tag{1c}$$

$$x_{p,s} \in \{0,1\}, \qquad \forall p \in P, \forall s \in S \tag{1d}$$

The objective function consists of the total cost of sequences assigned to each path. Constraints (1b) assign exactly one sequence for each path, ensuring sequence is suitable for the path. Constraints (1c) ensure that for each node that is shared by more than one path, the components tested on the shared nodes are the same. That is, we consider each pair of paths and if their n-th level shares the same node, then we guarantee the assigned pair of sequences test same components at that level. Considering Fig. 3 above, path 1 and path 2 share the same nodes (a and b) in first and second nodes in sequence. Therefore sequences 1, 3 and 1, 3, 2 for these respective paths are feasible. On the other hand, despite being suitable (in terms of length), the sequences 1, 3 and 1, 4, 3 for paths 1 and 2 would have violated constraint (1c). Next, we present an alternative approach to formulate the same problem.

3.1.2. Intermediate Node Based IP

In this formulation, the decision variables indicate which component is tested at the intermediate nodes (including the root node). I is the set of intermediate nodes, O is the set of leaf nodes (output states). P denotes the set of paths from the root node to the leaf nodes, *including the leaf nodes*. With this updated path definition, we observe there are a total of 6 paths in Figure 3, i.e., from the root node to each of the 6 outcome states. J is the set of components. We summarize parameters and decision variables in Table 6.

Table 6: Parameters and decision variable for intermediate node based IP

Parameters

$a^+_{p,i}$:	1 if node i test result is positive for path p, 0 otherwise.	$a^-_{p,i}$:	1 if node i test result is negative for path p, 0 otherwise.
$b^+_{j,o}$:	1 if component j works in output state o, 0 otherwise.	$b^-_{j,o}$:	1 if component j fails in output state o, 0 otherwise.
Pr_o:	Conditional probability of output state o.	c_j:	Cost of testing component j.
n^+_p:	Number of positive test results in path p.	n^-_p:	Number of negative test results in path p.

Decision Variables

$x_{i,j} = \begin{cases} 1, & \text{if component } j \text{ is tested in node } i, \\ 0, & \text{otherwise.} \end{cases}$	
$y_{o,p} = \begin{cases} 1, & \text{if path } p \text{ is assigned output state } o, \\ 0, & \text{otherwise.} \end{cases}$	
ξ_p:	Testing cost of path p.
χ_o:	Testing cost for output state o.

As the overall structure of the tree is known in advance for a failed k-out-of-n system, each path is computed through intermediate nodes and their outcomes (+ or -). This is reflected via $a^+_{p,i}$ and $a^-_{p,i}$ parameters. For instance, in Figure 3, $a^+_{\text{path to leaf 1, a}} = 1$ and $a^+_{\text{path to leaf 1, b}} = 1$, whereas $a^-_{\text{path to leaf 1, a}} = a^-_{\text{path to leaf 1, b}} = 0$. For brevity, we also label the number of positive and negative test results on a path as n^+_p and n^-_p, respectively. These can be computed as

$$n^+_p = \sum_{i \in I} a^+_{p,i}$$

$$n^-_p = \sum_{i \in I} a^-_{p,i}$$

For instance, in Figure 3, $n^+_{\text{path to leaf 1}} = 2$ and $n^-_{\text{path to leaf 1}} = 0$. On the other hand, $n^+_{\text{path to leaf 3}} = 1$ and $n^-_{\text{path to leaf 3}} = 2$. In the light of these parameter definitions, our model is as follows:

Formulation

$$\min \quad \sum_{o \in O} Pr_o \chi_o \tag{2a}$$

$$\text{s.t.} \quad \sum_{j \in J} x_{i,j} = 1, \qquad \forall i \in I \tag{2b}$$

$$\sum_{i \in I} (a^-_{p,i} + a^+_{p,i}) x_{i,j} \leq 1, \qquad \forall j \in J, \quad \forall p \in P \tag{2c}$$

$$\sum_{i \in I} \sum_{j \in J} (a^-_{p,i} + a^+_{p,i}) c_j x_{i,j} = \xi_p, \qquad \forall p \in P \tag{2d}$$

$$\sum_{p \in P} y_{o,p} = 1, \qquad \forall o \in O \tag{2e}$$

$$\sum_{o \in O} y_{o,p} = 1, \qquad \forall p \in P \tag{2f}$$

$$\sum_{i \in I} \sum_{j \in J} a^+_{p,i} b^+_{j,o} x_{i,j} \leq y_{o,p} + n^+_p - 1, \qquad \forall p \in P, \quad \forall o \in O \tag{2g}$$

$$\sum_{\forall i \in I} \sum_{\forall j \in J} a^-_{p,i} b^-_{j,o} x_{i,j} \leq y_{o,p} + n^-_p - 1, \qquad \forall p \in P, \quad \forall o \in O \tag{2h}$$

$$\chi_o \geq \xi_p + M(y_{o,p} - 1), \qquad \forall o \in O, \quad \forall p \in P \tag{2i}$$
$$x_{i,j} \in \{0, 1\}, \qquad \forall i \in I, \forall j \in J \tag{2j}$$
$$y_{o,p} \in \{0, 1\}, \qquad \forall o \in O, \forall p \in P \tag{2k}$$
$$\xi_p \geq 0, \qquad \forall p \in P \tag{2l}$$
$$\chi_p \geq 0, \qquad \forall o \in O \tag{2m}$$

The objective is to minimize the expected cost of testing by conditioning on all output states. Constraint (2b) assigns exactly one component to each node. Constraint (2c) ensures that a component is tested no more than once in each path. Constraint (2d) calculates the testing cost for each path by simply considering existence of test j on path p via $\sum_{i \in I}(a^-_{p,i} + a^+_{p,i})x_{i,j}$. Constraints (2e) and (2f) ensure that there is a one-to-one assignment between paths and outcome states. Constraint (2g) and (2h) ensure that output state is assigned to a suitable path by matching the required number of positive/negative tests. Constraint (2i) assigns the cost of outputs (χ_o) to a path cost (ξ_p) if $y_{o,p} = 1$, where M is large number.

Constraints (2g) and (2h) are sophisticated constraints that need further explanation. For example, consider the path to leaf 4 in Figure 3. This path has $n^+_{\text{path to leaf 4}} = 2$ and $n^-_{\text{path to leaf 4}} = 1$. We also have $a^-_{\text{path to leaf 4,a}} = a^+_{\text{path to leaf 4,c}} = a^+_{\text{path to leaf 4,e}} = 1$. This constraint set ensures output states with two + and two − outcomes can be assigned to this path, if and only if the corresponding test outcomes match with the components tested on the path. Take output state $(--++)$ as an example. Associate with this output state, $b^-_{1,(\text{-}\,\text{-}++)} = b^-_{1,(\text{-}\,\text{-}++)} = b^+_{1,(\text{-}\,\text{-}++)} = b^+_{1,(\text{-}\,\text{-}++)} = 1$, by definition. When $y_{(--++),\text{path to leaf 4}} = 1$, we have the following:

$$\sum_{i \in I} \sum_{j \in J} a^+_{\text{path to leaf 4},i} b^+_{j,(--++)} x_{i,j} \leq n^+_{\text{path to leaf 4}} = 2$$

$$\sum_{\forall i \in I} \sum_{\forall j \in J} a^-_{\text{path to leaf 4},i} b^-_{j,(--++)} x_{i,j} \leq n^-_{\text{path to leaf 4}} = 1$$

These constraints ensure either one of the following holds:

- $x_{\text{a},1} = x_{\text{c},4} = x_{\text{e},3} = 1$
- $x_{\text{a},1} = x_{\text{c},3} = x_{\text{e},4} = 1$
- $x_{\text{a},2} = x_{\text{c},4} = x_{\text{e},3} = 1$
- $x_{\text{a},2} = x_{\text{c},3} = x_{\text{e},4} = 1$

The less than or equal to sign with binary variable $y_{o,p}$ on the right hand side further ensures that output state cannot be coincidentally reached in any other path. If $y_{o,p} = 0$ in a feasible solution, that implies output state o is not assigned to path p, thus we should never reach either (i) n^+_p positive outcomes among supposedly positive components of that outcome or (ii) n^-_p negative outcomes among supposedly negative components. Going back to the example of output state $(--++)$, among components 1 & 2, no unassigned path for this state should ever reach 2 negative component 1 & 2 outcomes. Having said that, it can now be seen that in any feasible solution one and only one of constraints (2e) and (2f) would be binding.

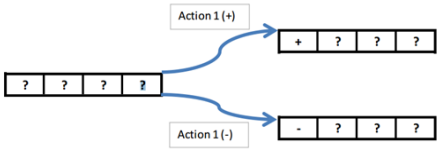
3.2. Markov Decision Process

Next, we propose a novel Markov Decision Process based method to solve the failure detection problem for k-out-of-n systems. Unlike previous integer programming based approaches, we develop a Markov decision process to model the uncertainty for test outcomes.

For each decision (intermediate) node, we have probabilistic outcomes of reaching the other states. The states are denoted by the state of each component as working (+), failed (−), or unknown (?). For instance, in a 3-out-of-4 system, there is one possible state in level zero (i.e., (????)) and 4 possible *actions* (i.e., testing component 1, 2, 3, or 4). In level one, we have the following states: (+???), (−???), (?+??), (?−??), (??+?), (??−?), (???+), (???−). Figure 4 shows a set of possible connections between level zero and level one. Obviously, if our action in level zero is "test component 1", that leads us to two states (+???) or (−???) with respective probabilities 0.78918 and 0.21082. These probabilities are obtained from Table 2 by summing up the posterior probabilities.

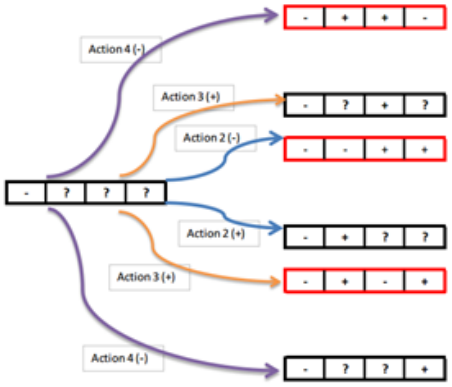
For the lower level states probability calculations and outcome states get more interesting. For example, in (−???) node of level one, we have 3 possible actions. If our action is test component 3, we reach two states: $(-\,+\,-+)$ and

Figure 4: Connection between level zero and level one (3-out-of-4)



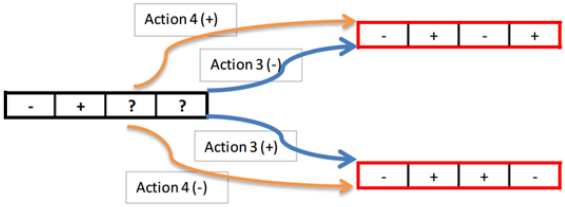
$(-?+?)$ with respective probabilities of $\frac{0.0144/0.2144}{0.21082}$ and $\frac{(0.0224+0.0084)/0.2144}{0.21082}$. These are the conditional probabilities of corresponding outcome states' probabilities given the initial state. Note that in $(-???)$ node, when component 3 is also failed, we don't have any unknown (?) component and reach a leaf node, i.e., $(-+-+)$. Figure 5 presents the set of all actions that can be taken in node $(-???)$ and corresponding outcome states.

Figure 5: State connection (3-out-of-4) when component 1 is known to fail



It should also be noted that starting at a state, two different actions might lead to the same state, depending on the outcome as illustrated in Figure 6.

Figure 6: State connection (3-out-of-4) when component 1 is known to fail and 2 is known to work



The main aim in our proposed MDP scheme is to find a stationary policy with the *minimum expected cost per unit time*. Here, a *stationary policy* indicates, whenever the system is in a certain state, the rule for making the decision is always the same regardless of the time. To solve this problem in an MDP scheme, all states are supposed to be communicating with each other. Therefore, the moment we detect the system state (i.e., a leaf node is reached), we assume there is a single choice that leads to the initial state where no component state is known. This way, finding the optimal strategy is synonymous to minimizing the expected cost per unit time over a system that recurs back to the initial state, the moment its state is found out.

There is one issue with minimizing the expected cost per unit time over a system that recurs back to the initial state: the levels of the leaf nodes are not the same. Thus, the time that it takes to go back to the initial state (root node) is not constant. We can explain how this can be an issue as follows: take two leaf nodes with same probability

and same total test cost in a policy. Their total contribution to the expected cost per unit time in the MDP scheme should be the same. However, if they are at different levels, the node that is reached faster (at a lower level) has a larger average cost contribution. In order to compensate for this, we introduce *artificial* states to guarantee the same path length between consecutive root node visits. We solve the minimum expected cost per unit time over this MDP, where after each leaf node, the necessary number of artificial nodes are visited before going back to the root node.

Next, we briefly present the LP that needs to be solved to find the minimum expected cost per unit time.

3.2.1. LP Formulation for MDP

In this model, testing a component is considered as an action a with cost c_a. The conditional probability of output state o is p_o. $d_{s,a}$ is an indicator that takes value 1 iff action a is available in state s. $p_{s'\to s,a}$ is the conditional probability of reaching state s from state s' using action a. That can be easily computed using posterior probabilities and components to be tested.

We have two sets of decision variables. $\pi_{s,a}$ is the probability of taking action a in state s.

MDP based Model

$$\min \quad \sum_{\forall s\in S}\sum_{\forall a\in A} c_a\pi_{s,a} \tag{3a}$$

$$\text{s.t.} \quad \sum_{\forall s\in S}\sum_{\forall a\in A} \pi_{s,a} = 1, \tag{3b}$$

$$\sum_{\forall s'\in S}\sum_{\forall a\in A} p_{s'\to s,a}\pi_{s',a} = \sum_{\forall a\in A} d_{s,a}\pi_{s,a}, \qquad \forall s \in S \tag{3c}$$

$$\pi_{s,a} \geq 0, \qquad \forall s \in S, \forall a \in A. \tag{3d}$$

The objective function is that of the classical MDP; the summation of the products of the probabilities of the chosen actions ($\pi_{s,a}$) and the associated cost of these actions (c_a). We assume the system is repetitively checked, returning to the beginning state (?,?,?,?), and the goal is identify the stationary deterministic policy that would minimize the long-run expected cost. Constraint (3b) ensures that sum of probabilities of all states is equal to 1 and constraint (3c) is the balance constraint for MDP.

3.2.2. Dynamic Programming for MDP

In this section, we explain our dynamic programming (DP) routine to solve the MDP. We create all the leaf nodes and initialize their cost to zero. Next, depending on the number of unknowns (i.e., ?'s) in a state, we identify the *level* of each state. This is not mandatory as we already know if state s can be reached from state s' using action a. However, we use the level information for a more efficient procedure for the DP. As there are a total of n components, a state with $n-l$ unknowns appear in l^{th} level, which we denote by S^l. There is no state in level $n-1^{\text{st}}$ and states in a level are not necessarily connected to one level up. Therefore, when we compute the cost of a state, we collectively consider all the levels under that level. We iteratively compute the following recursive equation starting with level n all the way up to level zero.

$$f^l(s) = \min_{\{a|d_{s,a}=1\}} \left\{ c_a + \sum_{s':p_{s\to s',a}>0} \left[p_{s\to s',a} f^{l'}(s') \right] \right\}, \quad s \in S^l \tag{4}$$

At level zero we have the state of unknowns for each component, where (4) gives the optimal objective function value. DP recursion can be backtracked to find the optimal strategy.

3.3. Heuristic Approaches

When the problem size reaches certain limits, the exact approaches are computationally intractable. In order to overcome this issue, we develop two heuristics to find a quick near-optimal solutions for our problem.

Our first heuristic finds a permutation strategy that simply tests the minimum cost component available. That is, the tests are ordered in increasing cost order and the tests are performed in sequential order until a leaf node is reached. In order to compute the expected cost of this policy we employ the following efficient procedure.

First, we sort all components in the system according to their test costs, where costs in increasing order are $c_{(1)}, c_{(2)}, \ldots, c_{(n)}$. $C_{(1)}$ is the component with minimum cost and $C_{(n)}$ is the component with maximum cost. The component state for each leaf node is reordered based on the ordered components with respect to the test costs. After this ordering, the number of tests required to reach each leaf node is computed as the minimum of the two numbers of tests to reach components known to be working or failed. In a failed k-out-of-n system, a leaf node is reached when either $k-1$ working or $n-k+1$ failed components are found out. Thus, for each leaf node, we count the number of tests (in increasing cost order) to reach either one of $k-1$ working or $n-k+1$ failed components. The minimum of these two numbers gives us the number of tests needed to reach that leaf node, say m, and sum of corresponding test costs (i.e., $\sum_{i=1}^{m} c_{(i)}$) gives us the cost of reaching that leaf node. Finally, we compute the total expected cost by the inner product of these costs with associated leaf node probabilities.

The second heuristic is a combination of the above permutation strategy and dynamic programing for MDP. We observe that the optimal solution for small instances can be found in reasonable time using the dynamic programming approach. In the second heuristic, we apply a permutation strategy until we reach small sub-systems. Next, we solve these small sub-systems to optimality using dynamic programming. We limit our permutation strategy until there are 9 components, which can then be solved in seconds.

3.4. Lower Bounding Scheme

This section develops two simple algorithms to find a lower bound that helps to assess the quality of our heuristics for relatively large instances.

Our first lower bounding scheme utilizes the same approach in our heuristic in that, the tests are ordered and the costs of reaching leaf nodes are computed in the same fashion. It can be easily shown that these costs are the minimum possible costs of reaching these leaf nodes. This holds true for any distribution of posterior probabilities. We next sort cost of reaching leaf nodes. For the sake of obtaining a lower bound, we take the inner product of cost of reaching leaf nodes in increasing order and posterior probabilities of leaf nodes in decreasing order, despite the mismatch. Note that, if all output states coincide without a mismatch, i.e., posterior probabilities in decreasing order are in line with cost of reaching leaf nodes in increasing order, it can be shown that our heuristic is guaranteed to provide an optimal solution. This argument also shows that if the posterior probabilities are uniform, an optimal solution is to execute the minimum cost test at each iteration.

The second lower bounding algorithm uses the adaptive strategy. To reach a leaf node, we need to find all failed or working components. If all the components that we test are working or failed to reach the leaf node, these are the best cases. We can find the testing costs for these two cases with summation of failed or working components' costs. We choose minimum of these costs for each leaf node, and multiply by the corresponding probability of relevant leaf node. We sum them up to find a lower bound. For larger instances, we use the maximum of these two lower bounds to compute the gaps and evaluate our heuristics.

4. Numerical Results

This section presents computational experiments conducted to show the limitations of exact approaches and performance of proposed heuristic algorithms. All computations are performed using Java codes, calling Gurobi 8.0 to solve optimization problems, on a 3.5 GHz Intel Xeon (E5-1650 v2) computer with 16 GB DDR3 ECC (1866 MHz) RAM and the macOS HighSierra operating system.

The component failure probabilities are uniformly generated between 0 and 1. Testing costs are generated uniformly as integers between 1 and 100. Due to the symmetric nature of the problem we only create instances where $k = 2, \ldots, \lfloor n/2 \rfloor + 1$. To clarify, there are $k-1$ working components in a failed k-out-of-n system, whereas there are $k-1$ failed components in a failed $n+2-k$-out-of-n system. Therefore, for any k and n, we did not create

$n + 2 - k$-out-of-n if we create a k-out-of-n system. For a better presentation, we show *number of failed components* $(n - k + 1)$ in all tables. IP1 stands for leaf node based IP, IP2 is intermediate node based IP, MDP shows classical LP solution for the proposed MDP framework, and dynamic programming approach for MDP model is abbreviated as DP in these tables.

4.1. Exact Approaches

First, Table 7 shows a set of small instances and how each exact approach performs in terms of solution time. There is one hour time limit for each solution method. We create 3 instances for each problem size and present the average solution time for each method. If an algorithm fails to reach optimality for at least one instance out of three, average of three optimality gaps are included in parenthesis.

Table 7: Time spent to reach optimality using exact algorithms on small instances (in seconds). Optimality gap average for three trials are included in parenthesis. No optimality gap implies all three instances are solved to optimality, i.e., 0%.

			Number of Components: n				
		Method	4	5	6	7	8
Number of Failed Components: $n-k+1$	2	IP1	0.0026	0.0935	3.5558	1450.3089	3601.9872 (92.42%)
		IP2	0.0349	0.3344	31.5076	1064.4983	2447.0332 (26.79%)
		MDP	0.0005	0.0012	0.0034	0.0184	0.1147
		DP	0.0017	0.0010	0.0015	0.0049	0.0222
	3	IP1			11.1839	3600.1323 (61.08%)	3601.8168 (99.00%)
		IP2			696.5207	3600.0124 (70.02%)	3600.0342 (68.95%)
		MDP			0.0060	0.0431	0.2651
		DP			0.0015	0.0120	0.0940
	4	IP1					3600.8722 (99.42%)
		IP2					3600.1979 (65.10%)
		MDP					0.4497
		DP					0.1522

It can be clearly seen that MDP based methods outperform IP approaches. For instance, none of the three 5-out-of-8 instances generated are solved by IP1 and IP2 in one hour time frame, whereas all are solved by MDP or DP in under one second. When number of working and failed components are close to each other, leaf node based IP encounters memory problems due to the number of sequences and the number of leaf nodes. It can be easily concluded from Table 7 that among smaller instances DP outperforms other methods, where leaf node based IP performs poorly.

Next, we consider larger instances that can be considered medium/large-sized. As IP1 and IP2 fails to solve all instances with 8 components, we do not use these method for larger instances. We set the time limit to 1 hours. As Table 8 reflects, DP usually performs better than MDP. OoM and TL denote an out of memory error and time limit is reached, respectively. We do not report any solutions exceeding 1 hour time limit, but we want to note that DP solution reaches optimum solution on average in 3,900 seconds for 13 out of 14 systems, in 9,500 seconds for 12 out of 14, in 4,100 seconds for 10 out of 13, and in 16,300 seconds for 7 out of 13. Creating the states and the mathematical model exceed 1 hour time limit, so we do not report their solution time for 13 out of 14. We face out of memory problem while creating the mathematical model for some instances.

Table 8: Solution time for medium/large instances' (in seconds)

	Number of Failed Components: $n-k+1$									
	2		3		4		5		6	
n	MDP	DP	MDP	DP	MDP	DP	MDP	DP	MDP	DP
9	0.26	0.12	4.36	0.75	11.33	1.57	-	-	-	-
10	3.06	0.71	29.93	4.90	120.98	12.96	190.87	17.79	-	-
11	17.24	3.75	219.65	29.69	1,342.69	99.92	4,108.36	153.31	-	-
12	88.27	18.55	1,652.32	175.72	OoM	1,122.87	OoM	1,641.73	OoM	2,009.47
13	484.07	98.40	OoM	1,181.81	OoM	TL	OoM	TL	OoM	TL
14	TL	506.10	OoM	TL	OoM	TL	OoM	TL	OoM	TL

It should be noted that we do not run our algorithms on n-out-of-n (or equivalently 2-out-of-n instances) with one failed (or working) component, as these problems are known to be polynomially solvable (see (Nachlas et al.

1990)). We conclude our set of small to medium sized problems (every instance with up to 12 components) with the performance profile of exact approaches in Figure 7.

Figure 7: Performance profile for 145 instances (3 instances of each problem size) with 4 to 12 components

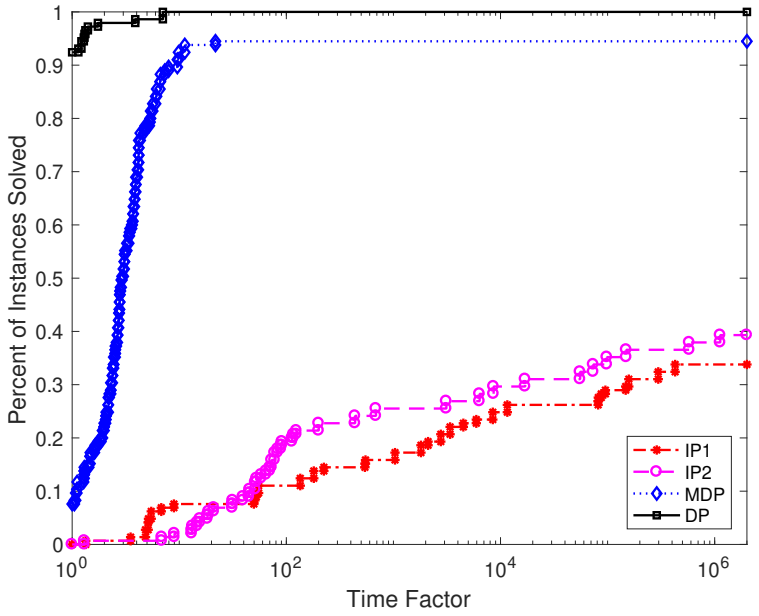


Figure 7 shows that DP clearly outperforms all algorithms and intermediate node based IP can solve more instances (despite long runtimes) compared to leaf node based IP.

4.2. Heuristic Approaches

Next, we present the performance of our heuristic approach. We randomly create 10 instances for each problem size and compare our heuristic approach against (i) optimal solution for small/medium instances up to 12 components and (ii) lower bound for large instances with 13 to 20 components. In order to show the tightness of our lower bound, we also report the gap between lower bounds and optimal solution for small/medium instances.

Table 9 shows the optimality gap for our heuristics together with the lower bound performance on small/medium instances. Note that we report the second heuristic for cases with more than ten components because of the limit on the number of components to be solved exactly. Thus, the gaps that are not reported are supposed to be zero.

As Table 9 shows, our simple heuristic has an average optimality gap of 30-40%. When the subtrees of size 9 are solved exactly, these gaps typically decrease by around 15-25%. What makes this analysis interesting is the fact that, optimality gaps are usually smaller for computationally challenging instances, where $k \approx n/2$. This holds true for both heuristic approaches. Considering the fact that first heuristic runs in under one second, these solutions can be utilized in a different solution framework. Table 9 also shows how far our lower bounds are from optimality. For challenging instances, our lower bounds are 20-35% less than the optimal solution, which gives an idea on what to expect from our heuristic vs. lower bound gaps for larger instances.

Table 10 shows the results for large instances. We randomly create three instances for large problems. We cannot find the optimal solution in an hour, therefore we use lower bounds to evaluate the performance of heuristics. We select the maximum of the two lower bounds described to come up with a tight lower bound. As in the small/medium sized instances, both heuristics' average gaps decrease when the problem gets more challenging ($k \approx n/2$). As expected, the second heuristic performs marginally better. However, the second heuristic solution time increases quite drastically for larger sized instances. Problems larger than 12-out-of-19 cannot be solved in 1 hour with our second heuristic because the exponential growth in the number of subtrees of size 9, which we pre-defined.

Table 9: Optimality gap for the first heuristic and lower bound performance (i.e., $\frac{LB-OPT}{OPT}$)

		First Heuristic Gap			Second Heuristic Gap			First Lower Bound Gap			Second Lower Bound Gap		
n	$n-k+1$	MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
4	2	0.00%	10.35%	67.20%	-	-	-	-46.96%	-12.91%	-1.45%	-21.80%	-15.40%	-7.54%
5	2	1.45%	17.13%	55.41%	-	-	-	-56.65%	-21.71%	-2.01%	-29.26%	-19.73%	-3.56%
6	2	7.04%	21.09%	32.60%	-	-	-	-50.58%	-29.25%	-11.58%	-35.35%	-28.74%	-18.86%
7	2	2.80%	32.17%	95.77%	-	-	-	-67.08%	-43.32%	-22.58%	-42.91%	-31.27%	-16.26%
8	2	16.75%	47.37%	71.68%	-	-	-	-65.77%	-40.27%	-26.80%	-44.33%	-37.58%	-24.99%
9	2	8.78%	39.13%	79.77%	-	-	-	-60.61%	-44.31%	-21.54%	-52.63%	-41.35%	-32.97%
10	2	6.34%	45.67%	86.65%	4.45%	20.92%	58.36%	-68.32%	-42.85%	-22.91%	-61.09%	-43.66%	-28.96%
11	2	14.83%	65.50%	150.53%	1.98%	30.48%	84.72%	-77.02%	-53.79%	-14.76%	-59.53%	-45.92%	-31.59%
12	2	18.08%	46.18%	88.94%	14.73%	32.00%	52.39%	-76.86%	-49.06%	-19.58%	-65.66%	-53.63%	-42.84%
6	3	4.32%	16.60%	45.70%	-	-	-	-62.25%	-30.07%	-13.20%	-26.34%	-18.06%	-9.20%
7	3	0.17%	16.78%	38.01%	-	-	-	-33.62%	-18.87%	-7.99%	-34.07%	-25.04%	-8.31%
8	3	8.04%	22.20%	60.58%	-	-	-	-36.76%	-26.88%	-17.43%	-39.46%	-29.67%	-14.19%
9	3	14.46%	40.93%	75.61%	-	-	-	-58.13%	-37.55%	-12.65%	-43.56%	-30.66%	-15.66%
10	3	15.64%	32.24%	62.58%	1.97%	12.59%	24.25%	-54.62%	-41.12%	-26.77%	-44.97%	-34.71%	-28.05%
11	3	25.16%	43.14%	71.96%	8.53%	24.86%	45.27%	-69.16%	-44.60%	-18.10%	-47.44%	-37.92%	-22.79%
12	3	12.89%	39.79%	72.03%	4.79%	19.06%	37.28%	-60.65%	-44.65%	-33.09%	-55.78%	-48.95%	-42.33%
8	4	8.48%	16.41%	27.64%	-	-	-	-37.45%	-24.24%	-12.02%	-29.46%	-24.28%	-18.37%
9	4	8.91%	23.33%	45.97%	-	-	-	-40.75%	-25.23%	-12.02%	-31.48%	-24.94%	-19.46%
10	4	5.39%	29.96%	68.10%	1.07%	7.32%	15.82%	-62.50%	-36.62%	-24.85%	-39.00%	-26.14%	-14.31%
11	4	5.30%	29.39%	49.56%	3.63%	14.29%	26.47%	-52.01%	-38.65%	-23.97%	-40.43%	-31.54%	-21.22%
12	4	6.62%	23.02%	46.23%	3.09%	11.78%	25.81%	-54.93%	-37.87%	-14.88%	-46.66%	-40.90%	-32.90%
10	5	9.37%	28.63%	53.18%	1.74%	6.95%	15.43%	-49.14%	-27.80%	-14.14%	-34.89%	-23.80%	-13.81%
11	5	11.57%	31.27%	58.25%	1.07%	11.87%	18.46%	-49.24%	-29.64%	-17.09%	-31.32%	-25.56%	-7.73%
12	5	13.29%	29.38%	60.10%	1.15%	12.89%	22.63%	-40.52%	-28.08%	-13.06%	-40.64%	-30.76%	-24.36%
12	6	9.01%	21.37%	49.87%	3.45%	8.13%	17.17%	-38.68%	-24.99%	-11.41%	-35.94%	-29.64%	-20.88%

Table 10: Heuristic results for large instances (time in seconds and gap between lower bound, $\frac{H-LB}{LB}$)

		Number of failed components: $n-k+1$								
n	Heuristic	2	3	4	5	6	7	8	9	10
13	1	0.00 (282.48%)	0.00 (137.47%)	0.00 (118.82%)	0.00 (111.42%)	0.00 (62.05%)				
	2	0.00 (222.53%)	0.05 (107.44%)	0.33 (96.49%)	1.10 (68.13%)	1.85 (40.46%)				
14	1	0.00 (220.02%)	0.00 (172.04%)	0.00 (87.05%)	0.00 (93.29%)	0.00 (80.18%)	0.00 (61.44%)			
	2	0.00 (131.64%)	0.08 (124.30%)	0.70 (66.59%)	2.91 (79.88%)	6.43 (57.06%)	7.74 (37.61%)			
15	1	0.00 (289.06%)	0.00 (173.01%)	0.00 (116.14%)	0.00 (87.47%)	0.00 (57.85%)	0.00 (73.65%)			
	2	0.01 (240.73%)	0.12 (159.82%)	1.36 (91.92%)	6.99 (73.39%)	18.53 (50.00%)	29.55 (51.07%)			
16	1	0.00 (279.45%)	0.00 (201.77%)	0.00 (106.46%)	0.00 (85.94%)	0.00 (81.47%)	0.00 (65.58%)	0.00 (66.24%)		
	2	0.01 (240.51%)	0.20 (150.57%)	2.49 (94.60%)	15.34 (72.62%)	51.22 (63.62%)	102.30 (49.97%)	123.20 (54.59%)		
17	1	0.00 (264.59%)	0.00 (184.06%)	0.00 (150.50%)	0.00 (105.51%)	0.00 (109.39%)	0.00 (97.70%)	0.00 (66.98%)		
	2	0.01 (257.15%)	0.31 (168.25%)	4.49 (130.88%)	31.44 (92.21%)	130.70 (90.31%)	306.93 (67.89%)	547.69 (47.48%)		
18	1	0.00 (396.86%)	0.00 (215.48%)	0.00 (169.70%)	0.00 (123.02%)	0.00 (103.15%)	0.00 (96.20%)	0.00 (69.10%)	0.00 (66.88%)	
	2	0.01 (330.25%)	0.46 (192.41%)	7.61 (145.78%)	62.31 (106.20%)	298.15 (85.18%)	1,085.41 (72.82%)	1,767.60 (55.76%)	TL	
19	1	0.00 (357.29%)	0.00 (259.74%)	0.00 (158.18%)	0.00 (101.92%)	0.00 (112.60%)	0.00 (105.90%)	0.00 (74.55%)	0.00 (70.08%)	
	2	0.02 (308.44%)	0.64 (216.67%)	12.40 (134.49%)	120.40 (89.91%)	761.70 (96.44%)	2,609.45 (86.93%)	TL	TL	
20	1	0.00 (349.25%)	0.00 (272.78%)	0.00 (230.64%)	0.00 (206.73%)	0.00 (117.44%)	0.00 (113.30%)	0.00 (80.66%)	0.00 (69.04%)	0.00 (68.10%)
	2	0.02 (329.06%)	0.93 (238.12%)	19.73 (207.88%)	226.47 (174.34%)	1,590.28 (106.47%)	TL	TL	TL	TL

We do not provide a performance bound on our heuristics, however even the first heuristic never exceeds 4 times the optimal solution. Considering the marginal improvement in the second heuristic, we conjecture a hybrid approach focusing on the lower levels of the tree might be more rewarding (e.g., look-ahead approaches). Another promising aspect of our heuristic approach is that, for the largest instance we solved (i.e., 9-out-of-20 with 184,756 leaf nodes),

we obtain an answer that is 70% away from optimal in under 10 milliseconds, on average. The bottomline is, when $k \approx n/2$, the permutation strategy performs quite well. When the problem is imbalanced (i.e., number of working and failed components differ), this information should be utilized, rather than applying the minimum cost test.

5. Concluding Remarks

In this paper, we have presented two IP based and two Markov decision process based exact algorithms for *failure detection problem*. The proposed solution algorithms are novel since exact approaches have not been studied for k-out-of-n system in the literature, to the best of our knowledge.

Markov decision process based methods perform better for small and medium size problems. We demonstrate the effectiveness and limitations of the proposed approaches by conducting extensive numerical experiments.

IP based solutions have poor performance, but they can be useful for large instances upon improvements such as column generation and benders decomposition. In leaf node based IP, we create all possible patterns to build to model. Instead of that, we can create a feasible set of test patterns and apply *column generation*.

For larger instances, the number of states increase exponentially for the LP formulation solving Markov decision process. One future direction might be decomposing this problem. We can postpone creating states that are not promising to avoid out of memory problems. Structure of dynamic programing may also allow us implement parallel programing techniques to improve solution times.

In this work, we mainly focus on exact methods. We also develop two simple heuristics for larger instances, where solution time for exact methods exceeds one hour. These heuristics are fast and easy to implement. The first heuristic uses permutation strategies, and the second one hybridize permutation and adaptive strategies. In order to assess the performance of the heuristics, we utilize a lower bound as well. As a future extension to solve large problems, one might utilize look-ahead approaches. Moreover the algorithm can decide which subtrees to solve to optimality using average exact solution time for each tree size from our study.

In terms of the model, one may consider the case when more than $n - k + 1$ components cause the failure of the system, for instance, in the case of a sudden shock. In this case, a similar framework can be utilized by using probabilistic information regarding the number of components that have failed. Furthermore, the number of possible sets of components that have failed will be larger than our model, which would make exact approaches even more challenging.

References

M. Heydari, K. Sullivan, An integrated approach to redundancy allocation and test planning for reliability growth, Computers and Operations Research 92 (2018) 182–193.

A. Krishnamoorthy, P. Ushakumari, k-out-of-n: G system with repair:the d policy, Computers and Operations Research 28 (2001) 973–981.

J. Flyyn, C. Chung, A heuristic algorithm for determining optimal replacement policies in consecutive k-out-of-n systems, Computers and Operations Research 31 (2004) 1335–1348.

S. Amari, H. Pham, G. Dill, Optimal design of k-out-of-n: G subsystems subjected to imperfect fault-coverage, IEEE Transactions on Reliability 53 (2004) 567–575.

S. Eryilmaz, Y. Devrim, Reliability and optimal replacement policy for a k-out-of-n system subject to shocks, Reliability Engineering and Systems Safety 188 (2019) 393–397.

W. Wei, K. Coolen, R. Leus, Sequential testing policies for complex systems under precedence constraints, Expert Systems with Applications 40 (2013) 611–620.

R. Daldal, Ö. Özlük, B. Selçuk, Z. Shahmoradi, T. Ünlüyurt, Sequential testing in batches, Annals of Operations Research 253 (2017) 97–116.

W. Wei, H. Li, R. Leus, Test sequencing for sequential system diagnosis with precedence constraints and imperfect tests, Decision Support Systems 103 (2017) 104–116.

M. Kovalyov, M.C.Portmann, A. Oulamara, Optimal testing and repairing a failed series system, Journal of Combinatorial Optimization 12 (2006) 279–295.

M.-F. Chang, W. Shi, W. K. Fuchs, Optimal diagnosis procedures for k-out-of-n structures, IEEE Transactions on Computers 39 (1990).

Y. Ben-Dov, Optimal testing procedures for special structures of coherent systems, Management Science 27 (1981) 1410–1420.

R. Butterworth, Some reliability fault-testing models, Operations Research 20 (1972) 335–343.

H. G. Kang, H. E. Kim, Unavailability and spurious operation probability of k-out-of-n reactor protection systems in consideration of ccf, Annals of Nuclear Energy 49 (2012) 102–108.

S. Faghih-Roohi, M. Xie, K. M. Ng, R. C. Yam, Dynamic availability assessment and optimal component design of multi-state weighted k-out-of-n systems, Reliability Engineering & System Safety 123 (2014) 57–62.

W. Wei, K. Coolen, F. Nobibon, R. Leus, Minimum-cost diagnostic strategies for k-out-of-n systems with imperfect tests, Discrete Applied Mathemetics 222 (2017) 185–196.

S. Rostami, S. Creemers, W. Wei, R. Leus, Sequential testing of n-out-of-n systems: Precedence theorems and exact methods, European Journal of Operational Research 274 (2019) 876–885.

Y. Barron, Group maintenance policies for an r-out-of-n system with phase type distribution, Annals of Operations Research 261 (2018) 79–105.

F. Tu, K. R. Pattipati, Rollout strategies for sequential fault diagnosis, IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 33 (2003) 86–99.

O. E. Kundakcioglu, T. Ünlüyurt, Bottom-up construction of minimum-cost and/or trees for sequential fault diagnosis, IEEE Transactions on Systems, Man and Cybernetics. Part A. Systems and Humans 37 (2007) 621–629.

S. Zhang, K. R. Pattipati, Z. Hu, X. Wen, Optimal selection of imperfect tests for fault detection and isolation, IEEE Transactions on Systems, Man, and Cybernetics: Systems 43 (2013) 1370–1384.

T. Ünlüyurt, Sequential testing of complex systems: A review, Discrete Applied Mathematics 142 (2004) 189–205.

J. Nachlas, S. Loney, B. Binney, Diagnostic-strategy selection for series systems, IEEE Transactions on Reliability 39 (1990) 273–280.

R. Canfield, J. Nachlas, Comments on "Diagnostic-strategy selection for series systems" by J.A. Nachlas et. al., IEEE Transactions on Reliability 40 (1991) 165.

Z. Shahmoradi, T. Ünlüyurt, Failure detection for series systems when tests are unreliable, Computers and Industrial Engineering 118 (2018) 309–318.

D. Ait-Kadi, Z. Simeu-Abazi, A. Arous, Fault isolation by test scheduling for embedded systems using a probabilistic approach, Journal of Intelligent Manufacturing 29 (2018) 641–649.

B. Wang, W. Wei, W. Zeng, K. Pattipati, Fault localization using passive end-to-end measurements and sequential testing for wireless sensor networks, IEEE Transactions on Mobile Computing 11 (2012) 439–452.

F. Bao, Y. Pang, W.-J. Zhou, W. Jiang, Y. Yang, Y. Liu, C. Qian, Coverage-based lossy node localization for wireless sensor networks, IEEE Sensors Journal 16 (2016) 4648–4656.

M. Garshasbi, Fault localization based on combines active and passive measurements in computer networks by ant colony optimization, Reliability Engineering and Systems Safety 152 (2016) 205–212.

M. Cheng, W. Wu, Data analytics for fault localization in complex networks, IEEE Internet of Things Journal 3 (2016) 701–706.

M. Kress, K. Lin, R. Szechtman, Optimal discrete search with imperfect specificity, Mathematical Methods of Operations Research 68 (2008) 539–549.

B. Wagner, D. Davis, Discrete sequential search with group activities, Decision Sciences 32 (2001) 557–573.

N. Song, D. Teneketzis, Discrete search with multiple sensors, Mathematical Methods of Operations Research 60 (2004) 1–13.

J. Kadane, Optimal discrete search with technological choice, Mathematical Methods of Operations Research 81 (2015) 317–336.

T. Lidbetter, Search games with multiple hidden objects, SIAM Journal on Control and Optimization 51 (2013) 3056–3074.

T. Lidbetter, K. Y. Lin, Searching for multiple objects in multiple locations, European Journal of Operational Research 278 (2019) 709–720.