TESTING STRATEGIES FOR k-out-of-n SYSTEMS UNDER FOREST TYPE PRECEDENCE CONSTRAINTS

by
AYDIN TANRIVERDİ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

SABANCI UNIVERSITY
Spring 2008

TESTING STRATEGIES FOR k-out-of-n SYSTEMS UNDER FOREST TYPE PRECEDENCE CONSTRAINTS

APPROVED BY:

Assistant Prof. Tonguç Ünlüyurt ....
(Thesis Supervisor)

Assistant Prof. Kemal Kılıç ....

Assistant Prof. Kerem Bülbül ....

Associate Prof. Bülent Çatay...

Assistant Prof. Cem Güneri...

DATE OF APPROVAL: ....

ACKNOWLEDGEMENTS

To start with, I would like to present my deep gratitude and thanks to my thesis advisor Dr. Tonguç Ünlüyurt for his endless support throughout my research in the last year of my master study. His continuous enthusiasm, encouragement and motivation helped me a lot in the preperation of this thesis. I also want to express my thankfulness to the committee members for their helpful reviews, comments and suggestions about the thesis. In addition, my cordial thanks go to Dr. Kemal Kılıç for his enormous encouragment. The financial support from Sabancı University for this thesis is also gratefully acknowledged. Last but by no means least, I would like to thank my family with all my heart for all their love and encouragement that I received all through my life, and to my dear friends Nazlı, Canan, Nalan, Sevan, Arda, Kutay, İlter, Can for their support and friendship.

ABSTRACT

This thesis investigates diagnosis strategies for *k*-out-of–*n* systems under precedence constraints. A *k*-out-of-*n* system consists of *n* independent components whose working probabilities of are known in advance. The system itself functions if at least k components function. The true state of the system is determined by the sequentially inspection of these components. This inspection is costly and the cost of inspection for each component is also known. This study aims to minimize expected cost of determining true state of such a system when there are forest type precedence constraints. Optimal inspection strategies are already known for series and parallel systems. In this study, modifications of these strategies are proposed for *k*-out-of-*n* systems. Numerical results are presented to evaluate and compare the proposed strategies

ÖZET

Bu tez *n*'in *k*'lısı (*k*-out-of-*n*) sistemlerde öncelik kısıtları oduğu zaman, tanılama stratejilerini araştırmaktadır. *n*'in *k*'lısı sistemler, çalışma olasılıkları önceden belli *n* tane bağımsız bileşenden oluşurlar. Sistemin kendisi eğer en az *k* tane bileşen çalışırsa çalışır. Sistemin gerçek durumu bileşenlerinin sırayla test edilmesiyle tespit edilir.Bu test işleminin bir maliyeti vardır ve her bileşenin testinin maliyeti de önceden bilinir. Bu çalışma *n*'in *k*'lısı bir sistemde koru (forest) tipi öncelik kısıtları varken, sistemin gerçek durumunu belirlemenin beklenen maliyetini asgariye düşürmeyi amaçlamaktadır. Seri ve paralel sistemler için en iyi test stratejileri zaten bilinmektedir. Bu çalışmada seri ve parallel sistemler için bulunan bu en iyi stratejiler *n*'in *k*'lısı sistemlere uygulanmak için değiştirilmiştir. Önerilen stratejilerin performanslarını hesaplamak ve karşılaştırmak için sayısal sonuçlar sunulmuştur.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

The problem of minimizing the expected cost of identifying true state of a system is encountered in many practical situations. The system consists of components which are either faulty or working. The system state is determined by the states of the components and the components should be tested individually until the state of the system is determined. This testing procedure is usually costly or time consuming and determining the sequence of testing that minimizes the expected cost is an important problem and it has been an important research area for several years.

The testing problems arise in many applications [17] including the design of interactive expert systems, reliability analysis of coherent systems, classification of pattern vectors[9], file screening/searching applications [5], manufacturing applications such as testing of machines before shipment, testing of manufacturing operations [1,10], design of screening procedures[1], wafer probe testing in electrical engineering [4]), best value, or satisficing search algorithms in artificial intelligence [16]), testing incoming patients against some rare but dangerous disease [18], organization and criterion of an applied research project [13], and in quiz shows choosing the right order of the quiz categories.

In this study, we focus on testing *k*-out-of-*n* systems. In these types of problems, the system is functional if at least *k* of *n* components are functional and system functionality only depends on the number of working and faulty components. This property is specific to *k*-out-of-*n* systems and for more general systems the system state may depend on the functionalities of the individual components in a more complicated manner. Typically, this dependency can be described by a monotone Boolean function.

In some applications, it is not possible to execute the tests for the components in any order. There may be precedence constraints among the tests due to physical, logical or technological reasons. The precedence constraints could also arise as a result of process analysis. The precedence constraints, like in other applications such as scheduling, can naturally be described by an acyclic directed graph. In this directed graph, the nodes correspond to the components or tests and an arc *(i,j)* means that component *j* cannot tested if component *i* has not been tested yet. . These precedence constraints increase the complexity of testing problem a lot. The common types of precedence constraints that have been studied in the literature are parallel chain precedence constraints [2] and the forest type precedence constraints [1]. We encounter the precedence constraints in many practical situations. For example, one wants to detect a rare disease through a series of tests but some certain tests should use the results of some other tests. The sequence of testing is restricted by the precedence constraints in this situation.

The *k*-out-of-*n* problems are studied very well in the literature. A strategy **S** for *k*-out-of-*n* problems is naturally represented by a binary decision tree see e.g. [17]. An optimal strategy for *k*-out-of-*n* problems, in which there are no precedence constraints, can be generated in polynomial time. The special cases *1*-out-of-*n* and *n*-out-of-*n* with forest type precedence constraints have also been solved optimally [6,7]. In [15] Simone and Kadane state the optimal conditions, which a strategy should provide, under general precedence constraints, but they do not give any algorithm to obtain such an optimal strategy. In [20] np hardness of *n*-out-of-*n* problem under the general precedence constraints has been proven. For none of the special precedence graphs, the general *k*-out-of-*n* problem has been solved optimally yet. Usually the solutions for testing problem under precedence constraints are based on reduction of the precedence graph [1,2]. Reduction of precedence graph means the partitioning of the precedence graph into subgraphs such that there is no violated precedence constraints when we order the subgraphs according to special permutations of components.

In this study, we propose solution methods for testing of *k*-out-of-*n* systems with forest type precedence constraints which generalize parallel chain precedence constraints. We apply two different reduction algorithms whose results are the same. The first reduction algorithm is proposed for forest type precedence constraints by Garey[1], and is used for *1*-out-of-*n* and *n*-out-of-*n* systems. This reduction technique has not been used in the solution of general *k*-out-of-*n* systems under the forest type precedence constraints. The second reduction algorithm which is used for parallel chain precedence constraints is proposed by Chiu et al [1]. We use this reduction technique for forest type precedence constraints by making some modifications. After reduction of the precedence constraints, we use Ben-Dov's optimal algorithm [3] which is for *k*-out-of-*n* systems without precedence constraints. Ben Dov's algorithm is based on finding a set such that testing a random element of this set will give optimal strategy. But under the precedence constraints the random selection does not give optimal solution anymore. Therefore we evaluate different selection methods from this set. Moreover we also evaluate strategies that are permutations of the components and we improve these permutations by applying simulated annealing method. As far as we know these are the first numerical results for k-out-of-n systems with forest type precedence constraints.

The remainder of the thesis can be outlined as follows. In chapter 2, we will formally define the problem and we will give details about different types of testing problems. A review of the literature on k out of n systems is also included in that chapter. In chapter 3, we will describe the commonly used testing strategies and modification of these strategies for k-out of-n problems with forest type precedence constraints. In chapter 4, we will give computational results of simulations and present a comparison of algorithms. In chapter 5, we will provide some concluding remarks and future research directions.

2. PROBLEM DESCRIPTION AND LITERATURE REVIEW

2.1. Problem description

Let's consider a system that consists of *n* components whose functionalities are not known yet. The component set of the system is N=$\{t_1, t_2, ..., t_n\}$ where x_i describes the functionality of component t_i as follows:t,

$$x_i = \begin{cases} 1 & \text{if } t_i \text{ is functional} \\ 0 & \text{otherwise} \end{cases}$$

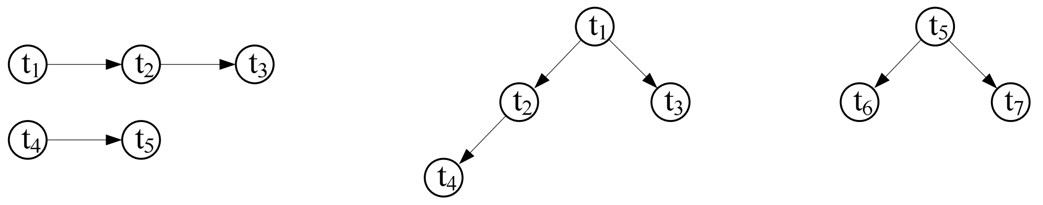
A state vector of the system $\mathbf{x} = (x_1, x_2, ..., x_n)$ is a boolean vector whose *i*th element shows the functionality of component *i* and the system functionality is characterized by the following function:

$$f(x) = \begin{cases} 1 & \text{if the system is working at system state x} \\ 0 & \text{otherwise} \end{cases}$$

In order to determine whether the system is working or not, we should learn the states of the components (not necessarily all of them) one by one. Initially, we do not know the states of the components but we have a prior working probability of each component. We should test a subset of the components to obtain the system's actual state. The testing procedure continues until the state of the system is determined. It is assumed that states of the components are independent random variables. Testing of each component is usually costly or time consuming; as such determining the sequence of testing is a vital issue to minimize the expected cost of determining true state of whole system. An inspection strategy ***S*** is a rule that specifies which component will be tested next, when we know the states of the already inspected components. So, an optimal inspection strategy is the one which has minimum total expected time or cost among all strategies.

In order to develop algorithms that produce good strategies, typically one should make use of the special structure of the system function and/or data. In certain applications, there are precedence constraints among the components. These constraints essentially state that, certain components are available for inspection if other certain components are inspected before. This situation generally can naturally be described by an acyclic directed graph. Each arc from node i to node j means component j cannot be tested before component i. [17].

Some special precedence types we frequently encounter are forest and parallel chain type precedence constraints. In parallel chain precedence constraints, there are disjoint subsets $N_1, N_2, ..., N_m$ of N where the precedence constraints exist only within each N_i. In addition, within each chain there is only one feasible testing strategy [2]. Garey in [1] defines forest type precedence constraints as "In each precedence graph either no component has more than one immediate predecessor, or no task in that component has more than one immediate successor." In other words, the precedence graph is a forest of in-trees and/or out-trees. Forest type precedence constraints obviously generalize parallel chain type precedence constraints. In figure 2.1, we see forest type and parallel chain type precedence constraints.



(a) **Parallel chain precedence constraints** **(b)** **Forest type precedence constraints**

Figure 2.1. Special Type Precedence Graph

2.2. Problem Types

2.2.1. Simple Series and Parallel Systems

Simple series and parallel systems are elementary but common. A series system is functional if and only if all the components are functional. So we continue to inspect the components until we find a faulty one or until we test the all components. The state function of the system is,

$f(x) = \min\{f(x_1), f(x_2), ..., f(x_n)\}$ Or $f(x) = x_1 \wedge x_2 \wedge, ..., \wedge x_n$

On the other hand, for the parallel systems, one functional component is enough to conclude that whole system is working. So we continue to inspect until we find a fault free component or until we test the all components. The state function of the system is,

$f(x) = \max\{f(x_1), f(x_2),..., f(x_n)\}$ Or $f(x) = x_1 \vee x_2 \vee,...,\vee x_n$

Although these are simple systems, the results that are obtained for these special cases are the milestones for the testing literature.

The basic and the simplest type of the testing problems are the simple series and simple parallel systems. A testing strategy for these problems corresponds to a permutation of components. For these systems, it is easy to find an optimal permutation. Let us define permutations τ and π as follows:

$$\frac{C_{\tau(1)}}{p_{\tau(1)}} \le \frac{C_{\tau(2)}}{p_{\tau(2)}} \le ... \le \frac{C_{\tau(n)}}{p_{\tau(n)}}$$

$$\frac{C_{\pi(1)}}{q_{\pi(1)}} \le \frac{C_{\pi(2)}}{q_{\pi(2)}} \le ... \le \frac{C_{\pi(n)}}{q_{\pi(n)}}$$

Then it is easy to show that τ is optimal for simple parallel systems and π is optimal for simple series systems. [6,7] The expected costs of these strategies can be written as follows:.

$$C(\tau) = \sum_{i=1}^{n} (C_{\tau(i)} \prod_{k=0}^{i-1} q_{\tau(k)}) \text{, where } q_{\tau(0)} = 1$$

$$C(\pi) = \sum_{i=1}^{n} (C_{\pi(i)} \prod_{k=0}^{i-1} p_{\pi(k)}) \text{, where } p_{\pi(0)} = 1$$

As Ünlüyurt states in [17] these strategies are very intuitive. For a parallel system the testing procedure ends when a working component is observed. So we are looking for a component which has minimum cost and maximum working probability. The component that provides these properties has minimum (c/p) ratio. The dual of this argument is valid for series systems.

In [1] Garey found optimum solution of the simple series systems in which testing sequence must satisfy forest type precedence constraints. The Algorithm mentioned in his study can also be applied to simple parallel systems. His algorithm is based on some reduction techniques that turn the system with precedence constraints into one where there are no precedence constraints. Since the testing procedure comes to halt when a faulty component is found, the optimal solution is a permutation of components. The results of Garey are not only restricted with the positive costs. They are also applicable for the negative costs (rewards). This algorithm will be described in detail later.

2.2.2. k-out-of-n Systems

These systems are the generalization of the *1*-out-of-*n* (parallel) systems and *n*-out-of-*n* (serial) systems. The system is functional if at least *k* of its components are functional. So we continue inspection procedure until we find *k* fault free components or *n-k+1* faulty components. The state function of the system can be described as follows:

$$f(x) = \begin{cases} 1 \;\; if \;\; x_1 + x_2 + \ldots + x_n \geq k \\ 0 \;\; if \;\; (1-x_1) + (1-x_2) + \ldots + (1-x_n) \geq n-k+1 \end{cases}$$

While, testing strategies for the simple parallel and simple series systems are represented by permutations of components, in general, the strategies for *k*-out-of-*n* systems can be represented by a binary decision tree. In Figure 2.2, we see a binary decision tree for a *2*-out-of-*3* problem.

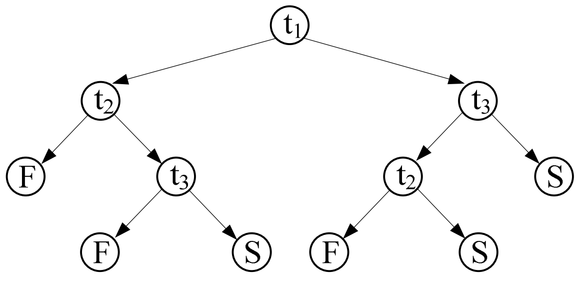


Figure 2.2. A Binary Decision Tree

Each node in the tree corresponds to a component. In the example, the root is indexed by the first component. This means testing starts with component 1. For each internal node in the binary decision tree, there are two outgoing arcs. So each internal node in the tree has two successors and leaf nodes have no successors. The two outgoing arcs from an internal node correspond to the result of the test of that node. Let us say the component associated with a node is t_i, then the right side successor of t_i is the component that we test next if we find that t_i is working and the left side successor is the component that we test next if t_i is faulty. The leaf nodes are "success" or "fail" nodes that show the state of the system. There is a unique path from the root to each leaf node. If the leaf node is success node, then there are *k* right arcs and less than (*n-k+1*) left arcs and for the fail nodes there are (*n-k+1*) left arcs and less than *k* right arcs on the path from the root to that leaf node. On each path from root to a leaf node we can observe the state of each component on this path and we can find the cost and probability of the path. The cost can be calculated by summing up all the costs associated to nodes on this path and the probability can be calculated by multiplying p_i s for the working state components and $1-p_i$ for the faulty components. In this framework, we can calculate the expected cost by multiplying path cost with the path probability. By summing up all paths' expected cost we will find the expected cost of our decision tree or expected cost of our testing strategy **S**.

E (C$_s$) $=\sum_i P(path_i) * Cost(path_i)$

$P(path_i) = \prod_{i,j} p_i * q_j$, where node set i corresponding to working components and node set *j* corresponding to faulty components.

$Cost(path_i) = \sum C_i$

Let us calculate the expected cost of binary decision tree in Figure 2.2 which is constructed for *2*-out-of-*3*system. Let us assume the following data for the costs and probabilities of the components.

$$C_{t1} = 5, C_{t2} = 8, C_{t3} = 4, p(t_1) = 0{,}4, p(t_2) = 0.5, p(t_3) = 0.8$$

There are 6 leaf nodes in the example so there are 6 paths. As an example let us calculate the cost of two leftmost paths.

$$C(t_1,t_2,F) = C_{t1} + C_{t2} \quad p(t_1,t_2,F) = (1-p_{t1})*(1-p_{t2}),$$

$$C(t_1,t_2,t_3,F) = C_{t1} + C_{t2} + C_{t3} \quad p(t_1,t_2,t_3,F) = (1-p_{t1})*(1-p_{t3})*p_{t2}$$

The others can be calculated by the same manner. The total cost of this decision tree is.

$$(C_{t1}+C_{t2})*(1-p(t_1))*(1-p(t_2)) + (C_{t1}+C_{t2}+C_{t3})*(1-p(t_1))*p(t_2) +$$
$$(C_{t1}+C_{t3})*p(t_1)*p(t_3) + (C_{t1}+C_{t2}+C_{t3})*p(t_1)*(1-p_{t3})$$

$C(E_s) = 13*0.3 + 17*0.3 + 9*0.48 + 17*0.12 = 15.36$

When there are no precedence constraints, Ben-Dov [3] obtains an optimal solution for the *k*-out-of-*n* problems, and the optimality of this solution is also proved by Chang et al [4]. His strategy tests a random component which is in both first *k* elements of permutation τ and first *n-k+1* elements of permutation π.

In [2], Chiu studies the sequential testing problem of the *k*-out-of-*n* systems with parallel type precedence constraints. They also give a sufficient optimality condition for the *k*-out-of-*n* problems with parallel chain precedence type. A reduction technique similar to Garey's reduction technique is used to handle the precedence relation. After the precedence relation problem is handled, they use Ben Dov's results, which are used for *k*-out-of-*n* systems for which there are no precedence constraints.

2.2.3. Series-Parallel Systems (SPSs)

An SPS is a specially structured network between two terminal nodes, called the source and sink. This system is functional if there is a path between source node and sink node. Simple parallel and simple series systems are special cases of general SPSs.

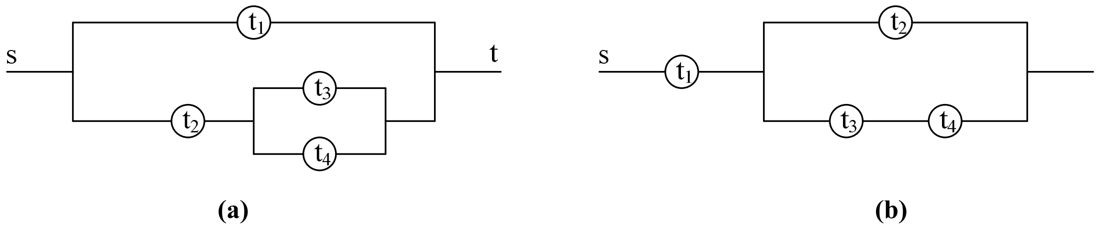


Figure 2.3. Two Examples of SPSs

In Figure 2.3 a the state function is $t_1 \vee (t_2 \wedge (t_3 \vee t_4))$

In Figure 2.3 b the state function is $t_1 \wedge (t_2 \vee (t_3 \wedge t_4))$

In this study we do not focus on SPSs. The results related to SPSs can be found in [19].

The testing problem for simple series systems and simple parallel systems without precedence constraint and with forest type precedence constraints have been solved optimally. For the general precedence type the optimum solution has not been found yet. The *k*-out-of-*n* systems without precedence constraints have been solved [3], but when there is precedence constraints the optimum strategy has not been determined yet. In this study we focus on sequential testing problem of *k*-out-of-*n* systems with forest type precedence constraints.

3. SOLUTION METHODS

In the previous chapter we mentioned some strategies that are used for the sequential testing problem of *k*-out-of-*n* system with certain precedence constraints. In this section we will analyze these algorithms in detail. As far as we know this is the first study for the *k*-out-of-*n* system with forest type precedence constraints. We will provide some numerical results for different strategies. The performance of permutation strategies is also compared to general binary decision tree solutions. We will use Ben-Dov's intersection algorithm and Garey's reduction algorithm together to find a good heuristic when there are forest type precedence constraints. We also modify Chiu's algorithm for the forest type precedence constraints which is originally designed for parallel chain precedence constraints.

3.1. An optimal algorithm for the series and parallel systems with forest type precedence graph. [1]

Garey provides some reduction rules that turn the precedence graph into a graph without any arcs. Essentially, the reduction rules combine certain nodes or delete some arcs in the precedence graph. Eventually, we have a problem with no precedence constraints where permutation π is optimal. The expected cost of a strategy **S** that inspects the components in the order 1,2,…,n is as follows.

$$C(S) = \sum_{i=1}^{n} (C_i \prod_{j=0}^{i-1} P_j)$$

The following theorem is given for the comparison of expected cost of two neighborhood strategy.

Theorem: Let's define $R(t_i)$ as, $R(t_i) = C_i / (1 - p_i)$ where p_i can be used instead of (1- p_i) for the parallel systems and $\mathbf{S}_1$ become a solution of series system such that $\mathbf{S}_1 = t_1, t_2, ..., t_n$ .A neighborhood strategy $\mathbf{S}_2$ is obtained by changing the place of two adjacent components then $C(S_2) < C(S_1)$ if and only if $R(t_{i+1}) < R(t_i)$.

This theorem shows that if we want to obtain an optimal solution for *n*-out-of-*n* systems, we should inspect the components in ascending order according to **R**-values. Since, if there is a solution in which the **R**-values are not in ascending order, we can decrease the expected cost of the strategy by simply interchanging components until they are in ascending order. In order to use this theorem for the problems with precedence constraints, the exchange should be feasible with respect to the precedence constraints.

When there are precedence constraints, sometimes it is not possible to exchange the order of two tasks that are not in the correct order with respect to **R**-value, without violating the precedence constraints. In order to prevent violation of the precedence constraints, the precedence graph is reduced to independent block nodes each of which has a merit value. These blocks can be thought as single nodes. Within each block only one permutation is possible. Two reduction techniques for forest type precedence constraints and duals of these techniques are mentioned in this paper.

First reduction theorem:

Definition 3.1.1: A leaf node in precedence graph **G** is called "terminal", and if a node is not "terminal" it is called "nonterminal".

Definition 3.1.2: If (t_i, t_j) is a given precedence pair then t_i, is an "immediate predecessor" of t_j. In other words, t_i is a predecessor of t_j and, there is no other components that is successors of t_i and predecessors of t_j. In this situation t_j is also called "immediate successors" of t_i.

Definition 3.1.3: Minimal successors of t_i is t_j if t_j is an immediate successors of t_i and t_j has minimum **R**-value among all other immediate successors of t_i.

Theorem 3.1.1: "For any given task ordering problem which has a solution, let t_i be a nonterminal task which has only terminal successors. If t_j is a minimal successors of t_i such that $R(t_j) \le R(t_i)$ and t_j has no other immediate predecessors, then there is an optimal solution in which the subsequence, t_i, t_j occurs."

By using this reduction theorem t_i, t_j can be thought as a single component. This means that ti and t$_j$ should be inspected consecutively. We encounter two problems when combining these two components as a single component. First one is finding merit value of this component and the second one is updating the precedence graph.

Since, we find the ratio of inspection cost to the failing probability of component in merit value formula, in order to find the merit value of this new component we should first consider inspection cost and fail probability of this new component. The expected inspection cost of this new component is $C_i + P_l * C_j$ since we do not inspect component j if the component i is not working.

The failing state of this new component occurs if at least one of the components is in fail state. Then the probability that this combined component functions is the product of the working probabilities of these two components. The R-value for the combined component is the ratio of expected inspection cost to the failure probability which is 1- functioning probability. Then the R-value for the combined component can be written in the following way.

$R(i,j) = C_{i,j} / Q_{i,j} = C_i + P_i C_j / 1\text{-}(p_i p_j)$ where $Q_{i,j}=1-P_{i,j}$ and $P_{i,j}=p_i p_j$

After we obtain **R**-value of the new block we should update the precedence graph **G**. In order to update precedence graph, first we should delete arc between (t_i, t_j) and add new arcs between this new node and each successors of t_i and also between new node and each and predecessors t_i.

The second reduction theorem:

Let t_j be a terminal task having an immediate predecessor t_i such that $R(t_i) < R(t_j)$. In this situation we can update preceding graph **G** as the following manner. We delete the arcs between t_i and t_j and we add arcs between predecessors of t_i and node t_j.

As we can see intuitively, removing the arc between t_i and t_j never violates the original precedence constraints if we sort the components in ascending order according to
R-value. After we obtain block nodes, if we sort the blocks according to **R**-value the optimal solution for the simple series systems with forest type precedence constraint is found.

By using these reduction techniques an optimal algorithm for an *n*-out-of *n* system can be constructed as follows:

Step 0: Initially **G'**=**G**.

Step 1: If there are no arcs in the reduced graph **G'**, then sort the components in a ascending order and output this order as an optimal solution and STOP. Otherwise go to step 2.

Step2: Find a node t_i in **G'**, that has only terminal immediate successors

. Let the minimal successors of node t_i be node t_j.

Step2a: If $R(t_i) \geq R(t_j)$ then add a new node (t_i, t_j) and update **G'** and calculate **R**-values as explained above. Go to step1.

Step2b: Else, delete the arcs between t_i and all immediate successors of it in **G'**, and then add arcs between the predecessors of t_i and all successors of t_i. Go to step1.

This reduction algorithm continues until there are no precedence constraints. In the final sorted form of components none of the precedence relation is violated. Since if the **R-**value of a component is less than the **R-**value of its predecessor, a new component is added to the graph such that the predecessor comes before its successor, otherwise; the predecessor comes before successors already because of its **R-**value.

The dual of this problem is *1*-out-of-*n* systems. The algorithm explained in this paper can be applied to *1*-out-of-*n* systems with only changing **R**-value by **S**-value such that,

$S(t_i) = C_i / p_i$, and $S(t_i, t_j) = C_i + (1 - p_i) * C_J / 1 - ((1 - p_i) * (1 - p_j))$.

Example 3.1

Let us apply this algorithm to the following example. Fig 3.1.shows the forest type precedence graph for a system that consists of 7 components.

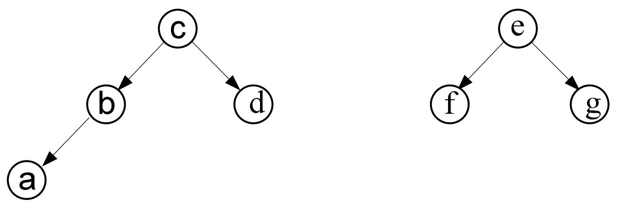


Figure 3.1.Precedence Graph for Example 3.1

The associated testing costs, probabilities and the R-ratios for the individual components are as follows.

$C_a = 10, p_a = 0.7, C_b = 5, p_b = 0.9, C_c = 15, p_c = 0.5, C_d = 10, p_d = 0.5$
$C_e = 5, p_e = 0.8, C_f = 10, p_f = 0.8, C_g = 15, p_g = 0.7$

$R(a) = 33.33, R(b) = 50, R(c) = 30, R(d) = 20, R(e) = 25, R(f) = 50, R(g) = 50$

Since component *b* has minimal successor *a* which satisfies the condition *R* (a)<*R*(b), (b,a) forms a block with the **R**-value 37.83. Now *c* has only terminal successors and minimal successor of *c* is *d*. So (c,d) forms a block with the **R**-value 26.6. Now (c,d) has only one immediate successor (a,b) and *R*(a,b)>*R*(c,d). So, the first tree is partitioned into the blocks (a,b) and (c,d). Since component e has only terminal successors *f* and *g* which satisfies *R* (e)<*R*(f)=*R*(g), the second tree is partitioned into three blocks (e),(f),(g) with the **R**-values 25, 50, 50 respectively. The final **R-**permutation is (e,c,d,b,a,f,g) and also *f* and *g* can be exchanged since their ratios are the same..

$S(a) = 14.4, S(b) = 5.5, S(c) = 30, S(d) = 20, S(e) = 6,5, S(f) = 12.5, S(g) = 21.1$

Since *S*(b)<*S*(a), we can delete the arc between *b* and *a*. Now *c* has three immediate successors and *S*(b)<*S*(c) so (c,b) is block with the **S**-value $15 + 5*0.5/1 - 0.5*0.1 = 18.42$, now the minimum successors of (c,b) is *a* which satisfies *R*(a)<*R*(c,b). So (c,b,a) is a block with the **S**-value $17.5 + 10*0.05/1 - 0.05*0.3 = 18.27$. Since *S*(c,b,a)< *S*(d), the first block is partitioned into two blocks (c,b,a) and (d) with the corresponding **R**-values 18.27 and 20. Since *S*(e)<*S*(f)<*S*(g) the second tree is partitioned into three **S**-blocks (e),(f),(g) with the **S**-values 6.5, 12,5, 21,1 respectively. The final **S**-permutation is (e,f,c,b,a,d,g).

3.2. An optimal algorithm for k-out-of-n Systems without precedence Constraints

The optimum algorithm for *k*-out-of-*n* systems without precedence graph is stated by Ben-Dov. The optimum testing procedure proposed in [3] is as following.

Let us define two sets by utilizing the permutations τ and π defined before. We take the first i elements of permutation τ and the first i elements of permutation π for defining the sets $\mathbf{U}_i$ and Vi**,** respectively.

$$U_i = \{\tau(j) \mid 1 \le j \le i\}$$
$$V_i = \{\pi(j) \mid 1 \le j \le i\}$$

If we take the intersection of the sets U_k *and* $V_{n-k+1} (U_k \cap V_{n-k+1})$ and inspect any of the elements in this set, we obtain an optimal strategy for *k*-out-of-*n* systems without precedence constraints. After we inspect the first component by using intersection set, one of the two possible states for the inspected component may be observed. If the inspected component is faulty and if we do not reach a result for whole system, we will have a new system with the parameters *k*-out-of-(*n-1*). If the inspected item is fault-free then new system will be (*k-1*)-out-of-(*n-1*). We should continue to apply this procedure till we find the correct state of the whole system. It is a surprising result that a randomly chosen component from the intersection leads us to an optimum strategy. In this point let's dwell on this surprising result. As a result of the testing procedure, we can observe either a faulty system or a working system. Assume that we know the system is faulty and we would like to prove this by finding (*n-k+1*) faulty components. That means, we have to inspect at least (*n-k+1*) components. We can get (*n-k+1*) faulty component in the first (*n-k+1*) tests. At this juncture, the question as to which (*n-k+1*) components should be chosen comes to fore. We search for an item that has both low cost and high probability of not working. It is obvious that the optimum strategy for this search is the permutation π. Even we obtain the result from first (*n-k+1*) tests we should inspect all of the components in V_{n-k+1} if the system is faulty. In the alternative case, when the system is working, we have to inspect at least *k* components. We should inspect all of the components in U_k for any optimal solution if we know that system is working. In short an optimal strategy should inspect all of the components in V_{n-k+1} so as to obtain a faulty system. In an attempt to obtain a fault-free system it should also inspect all components in U_k. Obviously, if there is such an item that it is in both U_k and V_{n-k+1} then it should be tested within all optimal sequences.

Because there are *k* components in U_k and (n-k+1) components in V_{n-k+1}, the set $U_k \cap V_{n-k+1}$ cannot be null. Any component in the set $U_k \cap V_{n-k+1}$ will be inspected in this optimal strategy. In addition, the working probability of an item and cost of inspection of an item do not depend on the sequence of item in the strategy, hence the randomly chosen item from $U_k \cap V_{n-k+1}$ will be used in the optimal strategy.

3.3. A Heuristic Solution for k-out-of-n Systems With Parallel Chain Precedence Constraint

Chiu [2] claims that the intersection algorithm proposed by Ben-Dov can be used to find a good solution under the parallel chain precedence constraints. They also give a sufficient condition for the strategies to be optimal for *k*-out-of-*n* systems with parallel chain precedence constraints. They do not evaluate performance of their algorithm numerically and they do not provide any selection method when there are more than one component in the intersection. Actually they use the same logic with Garey and the blocks obtained from these two reduction algorithms are the same. Now we describe the algorithm in Chiu 2 in detail.

Let I become an ordered set of components such that $I = (i_1, i_2, ..., i_j)$. Let us assume we are testing the components in the order induced by I and we stop as soon as we find a faulty component (simple series case).Then the **R**-value of I can be calculated as follows:

$$R(I) = \frac{C_{i_1} + p_{i_1} * C_{i_2} + ... + \prod_{k=1}^{j-1} p_{i_k} * C_{I_J}}{1 - \prod_{k=1}^{j} p_{i_k}}$$

As we said before, we can use the same reason in simple parallel systems. Let's define **S**-value of I as follows:

$$S(I) = \frac{C_{I_1} + q_{i_1} * C_{i_2} + ... + \prod_{k=1}^{j} q_{i_k}}{1 - \prod_{k=1}^{j} q_{i_k}}$$

Because we have two types of merit values (**S**-value and **R**-value) for a block, the precedence graph **G** can be partitioned into two type blocks according to **S**-value and **R**-value by using the following procedure

Blocking (reduction) procedure

Definition 3.3.1: A "chain" is the precedence constraint which only gives a unique inspection order. In the reduction procedure a chain is partitioned into two parts I_1, I_2. Let I_1^* becomes I_1 such that I_1^* has min **R**-value among all possible I_1 and I_1 is not empty. I_1^* is obtained as a first **R**-block for that chain. After removing the first **R**-block, we obtain a new chain. If this new chain is empty, blocking is complete for that chain; otherwise the procedure is applied to this new chain. The blocking procedure continues until all the chains are partitioned into blocks. For the **S**-blocks the same proceeding is done except, only **S**-value is used instead of **R**-value. After blocking is complete, we obtain **R**-blocks and **S**-blocks; each has **R**-probability and **S**-probability, **R**-cost and **S**-cost. There is no precedence relation between these blocks. Because of there are no precedence constraints between blocks, the intersection algorithm can be used for these blocks. Fig 3.2 shows a system with two parallel chain precedence constraints; let us obtain R-blocks and S blocks from these chains.

Example 3.2

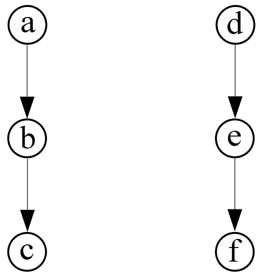


Figure 3.2. Parallel Chain Precedence Constraints for Example 3.2

$$C_a = 10, C_b = 5, C_c = 6, C_d = 7, C_e = 1, C_f = 12$$
$$p_a = 0.5, p_b = 0.25, p_c = 0.4, p_d = 0.8, p_e = 0.5, p_f = 0.6$$

$$R(a) = \frac{10}{0.5} = 20, \;\; R(a,b) = \frac{10 + 5 * 0.5}{1 - 0.5 * 0.25} = 14.29,$$
$$R(a,b,c) = \frac{10 + 0.5 * 5 + 0.5 * 0.25. * 6}{1 - 0.5 * 0.25 * 0.4} = 13.94$$

So our first block is (a,b,c) with the **R**-value 13.94

$$R(d) = \frac{7}{0.2} = 35, \;\; R(d,e) = \frac{7 + 0.8 * 1}{1 - 0.8 * 0.5} = 13,$$
T
$$R(d,e,f) = \frac{7 + 0.8 * 1 + 0.8 * 0.5 * 12}{1 - 0.8 * 0.5 * 0.6} = 16.57$$

The second chain is partitioned into two **R**-blocks, (d,e) and (f) with the R-values 13 and 30 respectively. As a result we obtain the R-permutation as (d,e,a,b,c,f).

$$S(a) = \frac{10}{0.5} = 20, \;\; S(a,b) = \frac{10 + 5 * 0.5}{1 - 0.5 * 0.75} = 20,$$
$$S(a,b,c) = \frac{10 + 0.5 * 5 + 0.75 * 6}{1 - 0.5 * 0.75 * 0.6} = 21{,}93$$

The first chain is partitioned into two S-blocks,(a,b) and (c) with the S-values 20 and 21,93

$$S(d) = \frac{7}{0,8} = 8.75, \quad S(d,e) = \frac{7 + 0.2 * 1}{1 - 0.2 * 0.5} = 8,$$
$$S(d,e,f) = \frac{7 + 0.2 * 1 + 0.2 * 0.5 * 6}{1 - 0.2 * 0.5 * 0.4} = 8.12$$

The **S**-blocks that are obtained from second chain are (d, e), (f) and final **S** permutation is (d, e, f, a, b, c).

Chiu also gives the following sufficient condition for optimality for *k*-out-of-*n* problems with the parallel chain precedence constraints. Let's define $\widetilde{R}$-ratio of a block N_{lj} as $\widetilde{R}_{LJ} = \min\{R(I_2) \mid I_2 \neq \varnothing, N_{lj} = I_1 I_2\}$ and $\widetilde{S}$-ratio of a block M_{lj} as $\widetilde{S}_{lj} = \min\{S(I_2) \mid I_2 \neq \varnothing, M_{lj} = I_1 I_2\}$.

"We say sequence α satisfies condition C1 if the $\widetilde{S}$ ratio of any block in the sequence is no less than the **S**-ratio of any block before it, which comes from a different chain. Similarly we say sequence β satisfies condition C2 if the $\widetilde{R}$-ratio of any block in the sequence is no less than the **R**-ratio of any block before it that comes from a different chain. The inspection procedure $\sigma \in \Omega$ (all inspection procedures) is optimal for the k-out-of-n system with parallel-chain precedence constraints under condition C1 and C2."

So far we mentioned optimal strategies for the special types of testing problems. Namely, optimal strategies are known for simple series and parallel systems under forest type precedence constraints and for k-out-of-n systems with parallel chain precedence constraints a sufficient condition for optimality is known for the algorithm of Chiu et. al. In this study we focus on different strategies for *k*-out-of-*n* systems with forest type precedence constraints. And we will compare different strategies in terms of their expected cost for randomly generated problem instances. Some of the strategies we will consider can be described by a permutation of the components while others can be described by a binary tree. Suppose the set of components is given by $S = \{t_1, t_2, t_3, ..., t_n\}$. Then a binary tree for a k-out-of-n system can be built by the following recursion where $P(S,k)$ corresponds to the testing procedure for the system **S** which is functional if the at least *k* of its components are functional..

$$.P(S,k)=\begin{cases}(test(t),\ \ P(S-\{t_i\},k),P(S-\{t_i\},k-1)) & if\ \ 0<k\le n\\ success & if\ \ k=0\\ failure & if\ \ k>n\end{cases}$$

For the recursive case, we choose component t_i by the different strategies. In the following section we will deliberate on these different strategies. Although we use the intersection algorithm after we reduce the forest type precedence graph, we are not sure about optimality of the strategy due to the precedence constraints. Let us analyze this situation in the following example. Let's consider a component set $S=\{t_1,t_2,t_3,t_4,t_5\}$ and two different *3*- out-of-*5* systems, S_1 *and* S_2 which are composed of these components.

In system S_1 there are no precedence constraints. Let π and τ be the permutations according to ascending order of C/q *and* C/p respectively. Then the intersection of first *k* elements of permutation π and first *(n-k+1)* elements of τ is not empty. After we test a component from this set, the resulting system will be *2*-out-of-*4* in the case the tested component is functioning or *3*-out-of-*4* system in the case the component fails. Let us assume the following data for the problem.

$C_1=2, C_2=2.6, C_3=2.4, C_4=2, C_5=2.58$

$p_1=0.4, p_2=0.5, p_3=0.5, p_4=0.6, p_5=0.5$

$\pi=t_1,t_3,t_4,t_5,t_2$

$\tau=t_4,t_3,t_1,t_5,t_2$

Initially, we have *k*=3 and (*n-k+1*)=3 for this example.The intersection set is $U_k\cap V_{n-k+1}=\{t_3,t_4\}$. Regardless of which component we choose, if the resulting system is *2*-out-of-*4* then $U_k\cap V_{n-k+1}$ will not include any additional component, only the component we did not choose for inspection will be included in the new intersection. If the new system is *3*-out-of-*4* then, t_1 and the component we did not choose for inspection will be included in the new intersection. The additional component, t_1 in the new intersection does not depend on the component that was chosen for the inspection.

If there are precedence constraints and if there are more than one component in the intersection then it becomes important which component that we choose for the inspection. In accordance with the item that is chosen the new intersection set may change. We can see this situation in the following example. We use the same data with the precedence constraints shown below.

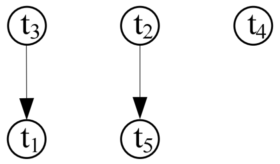


Figure 3.3. A Parallel Chain Precedence Graph

After the blocking procedure is applied, the following blocks are obtained. Blocks that are obtained from **R-**values are $[t_3, t_1]$,$[t_2, t_4]\, t_5$ and blocks that are obtained from **S**-values are $t_3, t_1, [t_2, t_4], t_5$. The ascending order of the blocks according to **R** and **S** values respectively are, $[t_3, t_1]\,[t_2, t_4]\, t_5$ .and $[t_2, t_4], [t_3, t_1], t_5$ .The set $U_k \cap V_{n-k+1}$ is obtained as $\{t_3, t_2\}$. If we select t_3 from this set and additionally if t_3 is not functioning then the new intersection will be $\{t_{1,} t_2\}$. On the other hand, if we select t_2 and also t_2 is not functioning then new intersection will be $\{t_3, t_4\}$. Although we obtain the same system after the first inspection, the next intersection and consequently the strategy may change according to the component that we choose from the intersection. As a result of this, Ben Dov's algorithm is not always optimal for the *k*-out-of-*n* systems with precedence constraints. Because of this, we analyze the performance of different selection methods of the component from the intersection. We also modify the reduction technique used in [3] for the forest type precedence constraints.

3.4. Strategies For k-out-of-n Problem Under The Forest Type Precedence Constraints

Now we can mention some strategies that we apply to solve *k*-out-of-*n*-problems with forest type precedence constraints

3.4.1. Using Intersection Algorithm

In order to solve this problem, we can use Ben-Dov's idea which chooses available items from intersection of two permutations. For the forest type precedence constraints we modify Chiu's reduction algorithm and obtain the two permutations accordingly. For the reduction case, we can also use Garey's Algorithm. In addition, we compare different selection methods if there are more than one component are available in the intersection set.In the next section we will analyze these strategies.

In order to use intersection algorithm, first we should obtain the independent blocks. This is performed by modifying Chiu's reduction algorithm.

Let us first apply this procedure to the example given in Figure 3.1.

The 4 chains for the leave nodes are as follows:

c-b-a, c-d, e-f, e-g.

The chain c-b-a is partitioned into the following **S** and **R** blocks.

R(c)=30,

$R(c,b) = 15 + 5 * 0.5 / 1 - 0.9 * 0.5 = 31.81,$

$R(c,b,a) = 15 + 5 * 0.5 + 10 * 0.5 * 0.9 / 1 - 0.5 * 0.9 * 0.7 = 32.11$

the block with the minimum ratio is (c).

So c is the first block for chain c-b-a , when we continue to apply blocking procedure we obtain the following blocks.

R-blocks; for the first tree (c), *R*(c) = 30,(b,a),*R*(b,a)=37.83, (c,d),*R*(c,d)=26.6 for the second tree *R*(e)=25, (f), *R*(f)=50, (e), *R*(f)=25, (g), *R*(g)=50

Now we will sort each block for each tree without repeating any components.

c-d-b-a, e-g-f (e-f-g) are our two **R**-chains and these chains can be used to find **R**-blocks. By the same manner the **S**-chains will be obtained as c-b-a-d, e-f-g after this transformation from forest type precedence constraints to parallel chain precedence constraints, the blocking procedure mentioned in 3 can be applied to these chains.

This procedure can be described as follows:

Step1) Blocking procedure for forest type precedence constraints:

Step1.1):

Obtain a chain for each leaf node of the precedence graph. (This chain is the path from the root to the leaf node)

i) Form a set of leaf nodes *LN* whose elements are without any successors.

ii) If $LN = \varnothing$ all chains are obtained. Go to step 2. Otherwise, add a new chain for each element i of LN.

iii) For each chain C_i, Current node CN =node i

iv) Add CN to beginning of C_i.

v) Update CN as immediate predecessor of CN

If CN is null chain is obtained remove i from LN and go to ii.

Otherwise go to iv.

Step1.2):

After we obtain one chain for each leaf node, we form a single **R**-chain and a single **S**-chain for the chains that are rooted from the same node.

i) put the chains that have same root into the same set

ii) For each chain in each set find I_{1R}^{*} such that if we partition a chain into two parts I_{1R} and I_{2R}, I_{1R}^{*} has min **R**-value among all possible I_{1R}. Remove I_{1R}^{*} from the chain and continue this process with the new chain until chain is empty.

iii) If all the chain in all sets is blocked according to **R**-value go to (iv). Otherwise go to (ii).

iv) For each chain in each set find I_{1S}^{*} such that if we partition a chain into two parts I_{1S} and I_{2S}, I_{1S}^{*} has min **S**-value among all possible I_{1S}. Remove I_{1S}^{*} from the chain and continue this process with the new chain until chain is empty. If all the chain in all sets is blocked according to **S**-value go to (v).

v) In each set, sort the blocks in a ascending order respect to the **R** and **S** value. Some items appear more than once within **R**-permutation and **S**-permutation. If we delete these repetitions in each permutation only the first appearance remains and, we come up with two permutations for each set. These permutations become our new R-chains and S chains.

Step1.3)

R-chains and **S**-chains are used to form **R**-blocks and **S**-blocks respectively. When we sort these blocks according to **R**-value and **S**-value we obtained a single **R**-permutation and a single **S**-permutation for whole system.

Step2) Obtaining the intersection set

U_k = First k components of permutation **S**

V_{n-k+1} = First n-k+1 elements of permutation **R**.

Add node i to AI such that i in the set $U_k \cap V_{n-k+1}$ and i has no predecessor. Until this point we form the intersection and available item set that contains the items that have no predecessor in intersection set. Because in both permutations a successor i always come after its predecessor j, if the i is in intersection set then j is also in intersection set.

Step 3) Selection from intersection

Selection from AI

In this part we use the following selection methods to choose component from the intersection when there are multiple components in the intersection and we compare these selection methods in the results part.

i) min c/p, choose the component that has the minimum c/p value.

ii) random selection, choose the component randomly.

iii) minimum of total index value (**MI**)

Now we will describe another selection method. Let the index of an item i in the permutation **R** be a_i and the index of i in permutation **S** be b_i then in the set AI we select $i*$ such that

$I^*(MI) = \min\{(a_i + b_i) \mid i \in AI\}$.

If there is a solution that provides Chiu's both conditions C1 and C2 then the selection method min of sum of index in α *and* β will give us the optimal solution.

Let's assume a procedure δ satisfies C1 and C2 then **R**-value of any block is less than or equal **R**-value of the blocks that comes after it. Because if $\widetilde{R}$-ratio of a block B_i is greater than **R**-value of B_j it implies that **R**-value of B_i is greater than **R**-value of B_j. If **R**-value of B_j is the smallest then **R**- index of first item of block B_j in β is also smallest. It is similar for the **S**-index in α. So if there is a solution that satisfies C1 and C2 then the chosen item has min sum of index.

3.4.2. A Greedy Approach by Defining a New Merit Value

Intuitively, if the number of needed successes is less than the number of needed failures then we should give more weight to test the components that is likely to be functioning and less costly.(sorting according to *c/p*). Otherwise *c/q* can be the dominating permutation. . We define the following merit value for each item to use this intuition.

Min ((c/p)*(k/ (n-k+1)), (c/ (1-p))*((n-k+1)/k))

As we can see from the formula as *k* increases the weight of (*c/(1-p)*) will also increase and the algorithm is favor of (*c/(1-p)*). As *k* decreases the weight of (*c/p*) will increase and the algorithm give more importance to testing component which has min (*c/p*).

The strategies that we mentioned so far can be represented by a binary decision tree. Although it is an effective way to use intersection algorithm to find a good inspection procedure, it is not easy to represent a solution in its entirety and to calculate its expected cost. As we mentioned before a strategy for *k*-out-of-*n* testing problem can be indicated by a binary decision tree. The size of the binary decision tree can be exponential in *k* and *n*. The number of nodes in a binary decision tree can be given by the following recursion.

$$N(n,k) = \begin{cases} N(n-1,k) + N(n-1,k-1) + i & \\ s & if \;\; k = 0 \\ f & if \;\; k > n \end{cases}$$

We assign 1 to nodes that we want to count. For example if we assign 1 to i we count only internal nodes. If we count all of the nodes we assign 1 to i , s and f . The size of binary decision tree for some (*n,k*) pairs are given in the following table:

Table 3-1. Size of Binary Decision Tree for Different (n,k) Pairs

n,k	7	8	9	10
20	232.559	406.979	587.859	705.431
40	10.759.231	36.312.407	104.902.511	262.256.279
60	872.541.559	5.889.655.529	34.683.527.009	180.354.340.451
80	3.599.158.127	30.142.949.321	221.048.295.027	1.436.813.917.681
100	20.471.735.855	222.630.127.433	2.127.354.551.035	18.082.513.683.805

As we can see from the table, the tree grows exponentially and this does not depend on the strategy that we use.

Since the total number of vertices in the binary decision tree is $\Omega(2^n)$ any method which explicitly generates the binary decision tree requires exponential time and memory. Ming-Feng et al [4] show that the binary decision tree that describes the Intersection algorithm can be represented by the block-walking representation, which is of size $O(n^2)$ and can be computed in time $O(n^2)$.

Definition 3.4.1: For any vertex *v* in a binary decision tree, define its tested unit set **TU** (*v*) to be the set of units tested along the path from the root to v, including *v*.

Definition 3.4.2: For any vertex v in a binary decision tree, define its tested state **TS** (*v*) to be an ordered pair (i, j) where i and j are the number of fault-free and faulty units tested along the path from the root to *v,* excluding *v.* If all the vertices with the same test state have the same test unit then the testing procedure can be simplified to a block walking representation. They prove that if the items are labeled according to ascending order of S value and if we take the item that has min label in $U_k \cap V_{n-k+1}$ the solution can be designated by a block walking method. The cost in block walking model can be obtained in a bottom-up fashion by iteratively computing expected cost at each grid point.

$$C(i,j)\begin{cases} 0 & if \quad i=k \\ 0 & if \quad j=n-k+1 \\ C^{+}_{I,J}+C^{-}_{i,j} & otherwise \end{cases}$$

$$C^{+}_{i,j}=CR_{i,j}+PR_{I,J}*C^{-}_{i+1,j}+QR_{i,j}*C^{+}_{i,j+1}$$

$$C^{-}_{i,j}=CL_{i,j}+PL*C^{-}_{i+1,j}+QL_{i,j}*C^{+}_{i,j+1}$$

Where, $CR_{i,j} =$ cost of item in the right side of grid point (i, j)

$PR_{i,j} =$Probability of item in the right side of grid point (i, j)

$QR_{i,j}$=1- $PR_{i,j}$

Although Block walking method can be used instead of binary decision tree for the representation of some optimal strategies, it is not always possible to represent any optimal strategy with a block walking diagram if there are precedence constraints. In the following example this situation is illustrated in detail.

Example 3.3

The system consists of the following components with the costs and probabilities,

$S=\{a,b,c,d,e\}$

$C_a = 8, C_b = 4, C_c = 1, C_d = 1, C_e = 1$

$p_a = 0.5, p_b = 0.5, p_c = 0.5, p_d = 0.5, p_e = 0.1$

and following precedence graph is given.

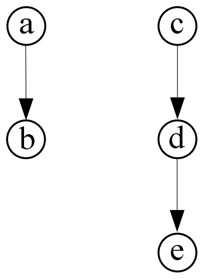


Figure 3.4. Parallel Chain Precedence Graph for Example 3.3

We searched all solution space with enumeration and we obtained the unique optimal strategy represented by the decision tree shown in the Figure 3.5

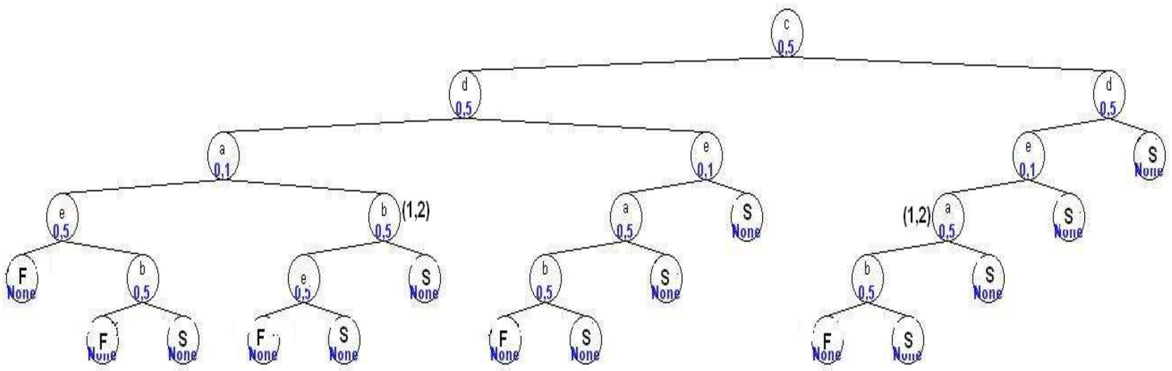


Figure 3.5. A Binary Decision Tree for 2-out-of-5 System

In this figure we label two nodes which have the same state, (1, 2). If the tested unit **(TU)** for each node that has the same state is equal then we can reflect this binary decision tree with block walking method. But in our example **TU** of the labeled node in the left is {c,d,a,b}and **TU** of the labeled node in the right is {c,d,e,a}. because these two sets are not equal, we cannot represent this tree with a block walking method.

Neither the optimal strategies nor many of other strategies can be represented by block walking method under the precedence constraints. Because of this, it is an important problem to evaluate performance of a strategy when we ar study on large instances. It is not possible to compute the expected cost of a strategy that cannot be represented by a block walking diagram in a reasonable time. On the other hand it is possible to calculate expected cost of a permutation strategy in time $O(n^2)$. Permutation strategies are the fixed sequence of components that we determine at the beginning of testing procedure and that do not depend on the results of the previous inspections if the system state is not determined yet. If the system state cannot be determined after inspecting a set of components, the next component to inspect next is the next component in the permutation. Let's construct the following matrix with entries $C_{i,j}$ showing, the expected value of remaining cost to determine the state of *k*-out-of-*n* system if the *i* of (*i*+*j*) inspected components are fault –free.

$$\begin{array}{ccccc}
C_{0,0} & C_{0,1} & .. & .. & C_{0,k} \\
C_{1,0} & . & .. & .. & .. \\
.. & .. & .. & .. & .. \\
.. & .. & .. & C_{n-k,k-1} & C_{n-k,k} \\
C_{n-k+1,0} & .. & .. & C_{n-k+1,k-1} & 0
\end{array}$$

$C_{i,j}$ can be calculated by the following recursion

$$C_{i,j} = \begin{cases} C(a_{i+j+1}) + p(a_{i+j+1}) * C_{i,j+1} + 1 - p(a_{i+j+1}) * C_{i+1,j} & \text{if } j < k \text{ and } i < n-k+1 \\ 0 & \text{otherwise} \end{cases}$$

where $C(a_i) =$ Cost of i^{th} item in permutation ω and $p(a_i) =$ Probability of i^{th} item in ω.

It is easy to see that for the base cases $C_{i,k}$ and $C_{n-k+1},_{j}$ the expected value of remaining cost is zero, since we obtain the state of the system. So it is computationally easy to compute the expected cost of permutation strategies.

3.4.3. Permutation Solutions

Let us denote by **RI** the remaining items that have not been tested yet and by **AI** the items that have not been tested yet and that are available to test according to the precedence constraints. The following procedure gives us a feasible solution for the *k*-out-of-*n* systems with any type precedence constraints.

-Find all t_i such that $t_i \in AI$

-choose t_i^* such that t_i^* has min *c/p* value (in this step we also used *c* value instead of c/p) among all t_i.

3.4.4. Improving Permutation Solutions

Our initial experiments showed that the permutation solutions are inferior to the intersection solutions. In order to find out whether the permutation solutions can be improved easily, we used a simple hill climbing algorithm on the initial permutation solutions.

Hill climbing algorithm is a simple local search algorithm. It takes an initial solution and searches for the neighbors of this solution. Then, it compares the objective function value of the current solution with the objective function value of the neighborhood solution which has the best objective function value. If the objective function value of the neighborhood solution is better than the objective function value of the current solution, the algorithm moves the neighborhood solution automatically. Otherwise the algorithm terminates.

We obtained our initial permutation solutions by using c/p and c values. After we obtained an initial solution, we just tried to improve it by using hill climbing. We generated all the neighborhoods by the following generation method. We interchanged each component i in ω with the components which come after i. Each permutation which was obtained with a single swap operation of two components forms a new permutation strategy. We only did feasible swapping operations which do not violate precedence constraints.

4. COMPUTATIONAL RESULTS

The algorithms that we presented in the previous chapter are coded in C# language. In this chapter we perform the evaluation of the strategies by experimenting them with different randomly generated instances with different parameters. We compare each approach with others in terms of the expected cost for each instance. We apply 8 different approaches to random instances. These can be summarized as follows:

- Permutation c/p: At each step we test the component, which has no parent in the precedence graph and has minimum c/p value. It gives a permutation strategy that will be denoted by (perm,c/p)
- Permutation c: At each step we test the component which has no parent in the precedence graph and minimum c value. It gives a permutation strategy that will be denoted by (perm,c)
- Minimum merit value: We choose the component which has no parent and which has minimum merit value (Min ((c/p)*(k/ (n-k+1)), (c/ (1-p))*((n-k+1)/k)). It does not necessarily give permutation strategies. (min merit)
- Local search with the initial solution obtained from permutation c/p (c/p, LS): It gives a permutation strategy.
- Local search with the initial solution obtained from permutation c (c, LS): It gives a permutation strategy.
- Selecting a component from ($U_k \cap V_{n-k+1}$). The selected component has no predecessor in the remaining components and it has min (c/p) (int, min(c/p)). It does not necessarily give permutation solutions.
- Selecting a component randomly from ($U_k \cap V_{n-k+1}$). The selected component has no predecessor in the remaining components and it (int,rand) does not necessarily give a permutation strategy.
- Selecting a component from ($U_k \cap V_{n-k+1}$) the selected component has no predecessor in remaining items and which has minimum sum of index (int, MI). It does not necessarily give a permutation strategy

In order to simulate the performance of the strategies, we generate random problem instances. The fault-free probabilities are chosen randomly from uniform distributions with parameters (0.01, 0.99), (0.25, 0.75), (0.5, 0.75), (0.75, 0.99). The testing costs of the components are also generated randomly with uniform distribution on (1, 99). For the experiments, we consider k-out-of-n systems with *n*=20, 40, 60, 80 and 100 components. For each value of *n*, *k* varies from 1 to *n/2*. We do not use *k* values which are more than *n/2* because these problems are the dual of the problems that are already generated. For each *(n,k)* pair, we generate 10 random instances for each probability distribution. In total, 40 random problems are generated for each *(n,k)* pair. There are 150 different *(n,k)* pairs so altogether there are 6000 problem instances.

We generate the precedence graphs randomly as follows. We assign each component a random depth between 0 to (max.depth-1). The range for max depth is from 1 to number of components and it is a user specified value. After we assign depth of each component, we assign the depth of its immediate predecessor. This depth is the next small assigned number as a depth. If there are more than one component with the same depth, one of the components is chosen randomly as an immediate predecessor.

The strategies that are represented by binary decision tree could not be evaluated for large (*n*, *k*) pairs. We could obtain the expected cost of diagnosing of all (20, *k*) pairs under each strategy. For large values of *n* values, although it is possible to compute the expected cost of the strategies which are represented by binary decision trees only for small *k* values, we could evaluate the performance of permutation solutions for all (*n*, *k*) values as described in the previous chapter. (*n*,*k*) pairs, for which cost of the strategies which are represented by binary decision tree could be computed, are shown in Table 4.1.

Table 4-1. (n,k) Pairs for Which We Computed Expected Cost of Binary Decision Tree Strategies

n=20	for all k values
n=40	K=1,2,3,4,5,6
n=60	k=1,2,3,4,5
n=80	k=1,2,3,4
n=100	K=1,2,3

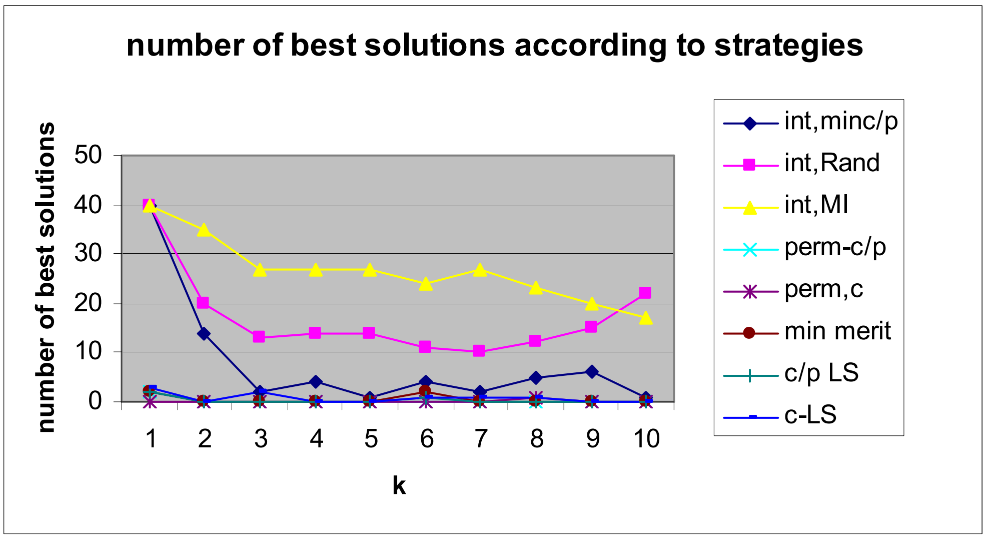


Figure 4.1. Number of Best Solutions v.s Strategies

In Figure 4.1 we show the number of instances for which different algorithms provide the best solution for different k values when n=20.

Let us recall that we have 40 instances for each *(n,k)* pair. We obtain similar results for other values of n and k. The results can be found in table form in the appendix. As we can conclude from the Figure 4.1, generally the strategy that chooses the component from the set $(U_k \cap V_{n-k+1})$ according to the sum of minimum index performs better than other strategies. More precisely, best solutions were given by this strategy under the conditions n=20 and k from 1 to *n/2*. For k=1 all selection method from $(U_k \cap V_{n-k+1})$ gave the same result due to the fact that there is only one element in intersection.

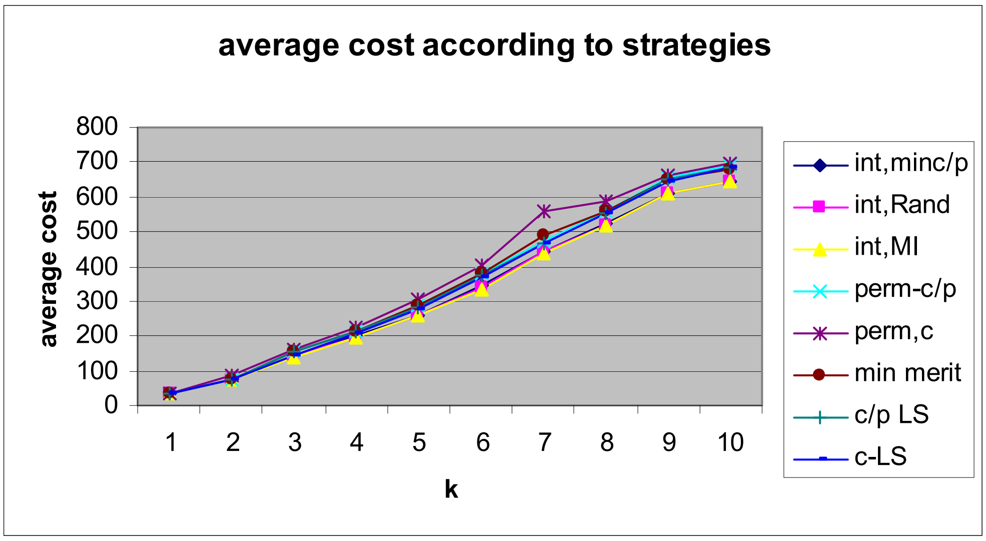


Figure 4.2. Average Cost v.s Strategies

Figure 4.2 shows the average expected cost versus k value when *n*=20. The average expected cost value increases as *k* increases as we expect. As we can see from the figure the gap between the performances of the strategies also increases as *k* increases. When *k*=1 the strategies using intersection algorithm is optimal. In addition to these, for small *k* values permutation *c/p* is nearly gave the same expected cost value with the intersection algorithms.

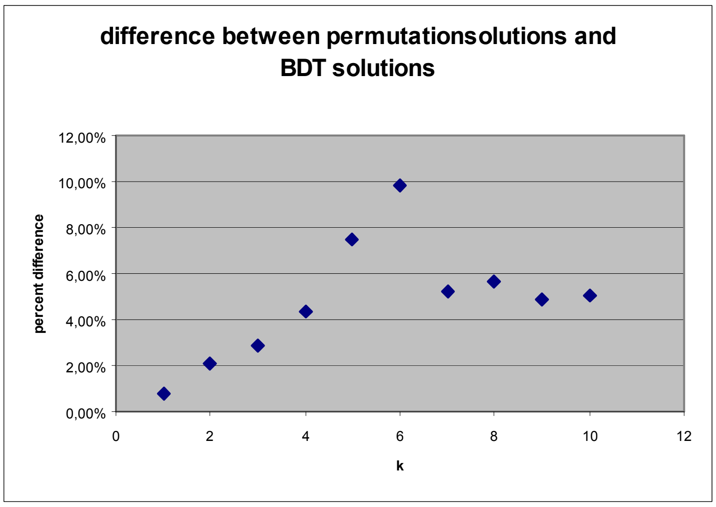


Figure 4.3. Percent Difference Between Best of BDT and Best of Permutation Solutions

For large values of (*n*,*k*) pairs, we were able to evaluate only permutation strategies. In Figure 4.3, we compare the best permutation solutions with the best binary decision solutions for n=20. When k is small, the difference between the testing costs under the permutation strategy and the intersection algorithms is small, but as k increases permutation solutions deteriorate.

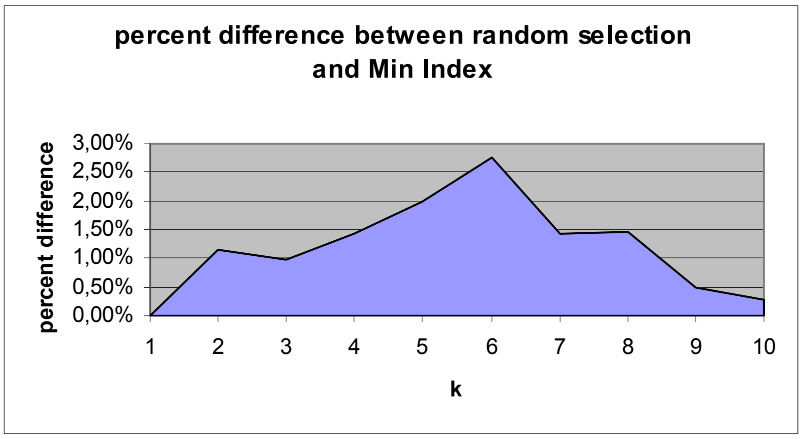
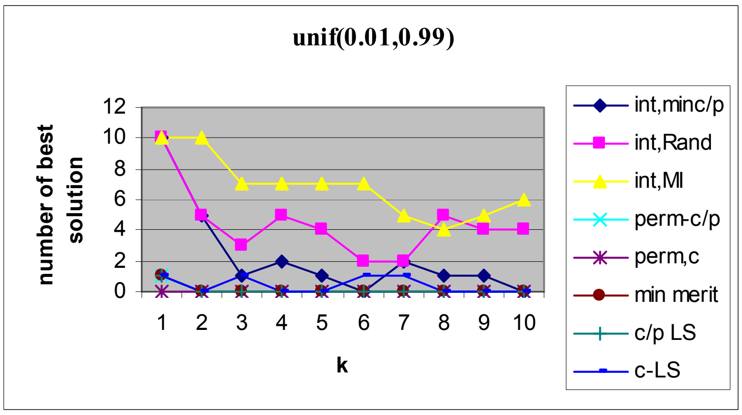
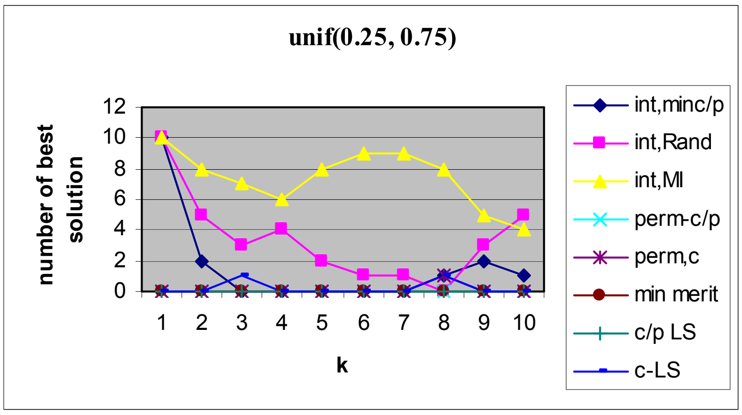


Figure 4.4. Percent Difference Between the Intersection Algorithm With Random Selection and Intersection Algorithm With Min Index Selection Method

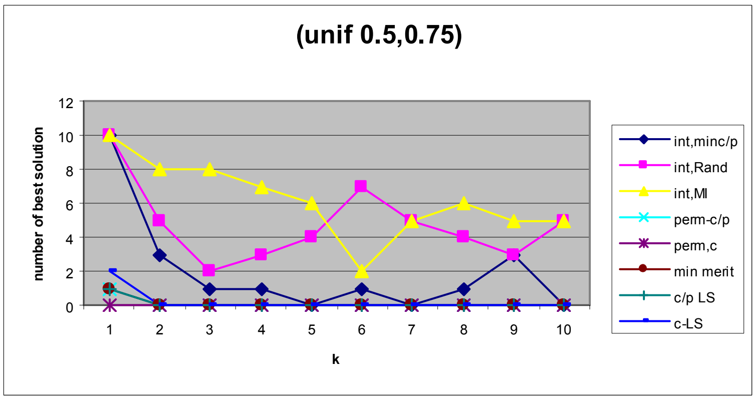
In order to predict range of difference of expected cost values under different selections methods from $(U_k \cap V_{n-k+1})$ we form the Figure 4.4. The comparison between average cost under the random selection from the intersection and average cost under the selection according to minimum index is presented in this figure. As we conclude from the graph this difference is usually under 3%.



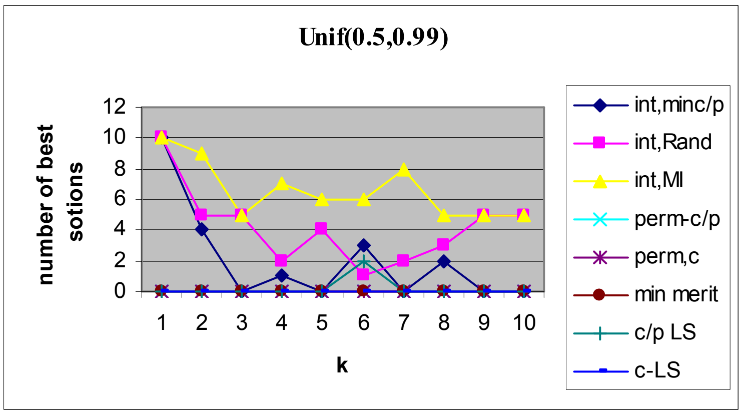
(a)



(b)



(c)



(d)

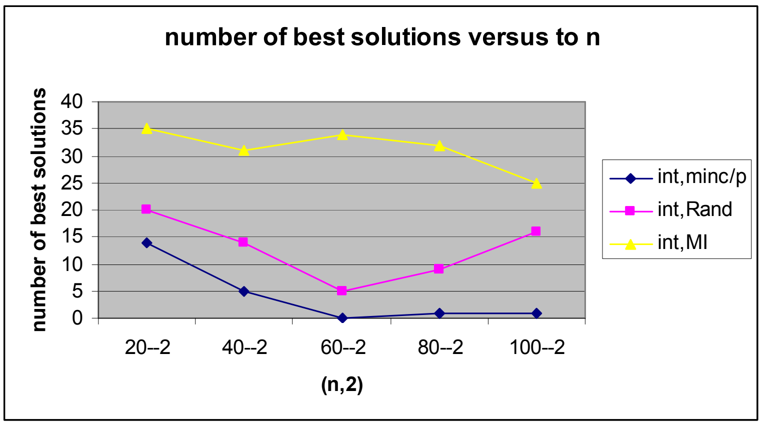
Figure 4.5. Number of Best Solutions v.s Strategies for Different Probability Distributions

Now we examine the results in terms of the range of the probability values, In Figure 4.5, we plot the number of best solutions with respect to k for different probability distributions when there are 20 components. When p_i's are from (0.75, 0.99) we see that the performance of the random selection and sum of min index selection are close to each other. The probabilities in these instances are closer to each other, so the costs play an important role to construct the intersection. This leads to few elements in the intersection and closer performance of different selection methods

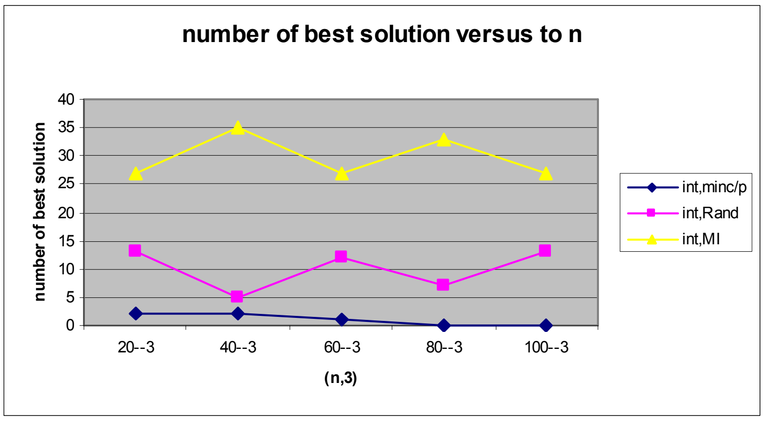
Table 4-2. Percent Difference Between all Strategies and Intersection Algorithm With Random Selection Method

n	int,minc/p	Int,Rand	int,MI	perm-c/p	perm,c	min merit	c/p LS	c-LS
20	0,72%	0,00%	-0,31%	7,40%	12,23%	7,71%	6,35%	5,00%
40	0,82%	0,00%	-0,81%	7,01%	14,34%	5,68%	6,24%	4,84%
60	0,46%	0,00%	-0,46%	5,58%	12,27%	4,07%	4,96%	3,29%
80	0,31%	0,00%	-0,22%	4,43%	11,71%	3,14%	3,82%	2,39%
100	0,26%	0,00%	-0,23%	3,11%	11,45%	1,98%	2,99%	2,00%

In order to compare the algorithms among themselves with respect to the size of the problem, we consider the algorithm that selects randomly from $(U_k \cap V_{n-k+1})$ as a benchmark. Table 4.2 indicates the percent difference of performance of all algorithms from the performance selecting a random component from $(U_k \cap V_{n-k+1})$. We take this algorithm as a benchmark because the optimal algorithm that Ben-Dov states when there are no precedence constraints tests a random component from the intersection. The negative difference means a better performance than the benchmark algorithm. Although we expect that the gap between performances of algorithms increase as n increase we encounter with an opposite situation. These result can be a conclusion of that we can only get data for small k values.



(a)



(b)

Figure 4.6. Number of Best Solutions v.s to Different (n,k) Pairs.

In Figure 4.6(a) and 4.6(b) we see the number of best solutions for *k*=3 and k=*4* for different *n* values. As we say before the most of the best solutions is given by the sum of min index selection method.

We also compare the permutation strategies for the large values of (n,k). We take permutation c/p as benchmark strategy. Because k values are small than the (n-k+1) values, permutation c/p generally performs better than permutation c. We obtain different improvement rates for different n values as a result of hill climbing. Following table summarize this information. This table is obtained by taking average of percent differences

Table 4-3. Percentage Difference Between all Permutation Strategies and Permutation According to Ascending Order of c/p

n	perm-c/p	perm,c	c/p LS	c-LS
20	0,00%	4,77%	-0,89%	-1,95%
40	0,00%	4,53%	-1,15%	-2,79%
60	0,00%	3,29%	-1,63%	-3,87%
80	0,00%	3,68%	-1,74%	-3,69%
100	0,00%	3,29%	-2,01%	-4,15%

As a conclusion, the selection method sum of minimum index performs better among the other algorithms. The permutation strategies show worse performance but the expected costs are not much bigger from the expected costs that are obtained from the intersection algorithm. Because finding a permutation solution is easy than finding a binary decision tree solution, it is also reasonable to use permutation solutions for testing problem.

5. CONCLUSION

In this thesis we study the k-out-of-n testing problem under forest type precedence constraints. Our objective is to find a strategy that minimizes the expected cost of diagnosing the system state which is functional if at least k of its component are functional. In the literature k-out-of-n problem without precedence constraints is solved optimally. The special cases, 1-out-of-n and n-out-of-n testing problems under forest type precedence constraints are also solved optimally. For the parallel chain precedence type which is a special type of forest type constraints, an optimality condition is given and a heuristic method is suggested. The optimal inspection strategy for the general k-out-of-n systems under forest the forest type precedence constraints. The main contributions of this study can be summarized as follows.

- We modified some well known algorithms, which are already applied to more special cases, to solve k-out-of-n systems under forest type precedence constraints.
- We evaluate the performance of these algorithms for the small (n,k) pairs.
- We improve some permutation strategies by using hill climbing. For the large size instances, we evaluate the performance of permutation strategies.

Future research directions in this area may be as follows.

- Finding optimal strategy for k-out-of-n systems under both parallel chain precedence constraints and forest type precedence constraints.
- Finding the optimal strategy for both special and general cases under general precedence constraints.
- Finding the optimal permutation strategies for k-out-of-n systems under all of parallel chain precedence constraints, forest type precedence constraints and the general precedence constraints.

REFERENCES

[1]M.R. Garey, Optimal task sequencing with precedence constraints, Discrete Math. 4 (1973) 37-56

[2]S.Y. Chiu, L.A. Cox Jr., X. Sun, Optimal sequential inspections of reliability systems subject to parallel chain precedence constraints, 1997 (personal communication).

[3]Y]. Ben-Dov, Optimal testing procedures for special structures of coherent systems, Manage. Sci.27(12) (1981) 1410-1420

[4]M.-F. Chang, W. Shi, W.K. Fuchs, Optimal diagnosis procedures for k-out-of-n structures, IEEE Trans. Comput. 39(4) (1990) 559-564.

[5] J.Y. Halpern, Evaluating Boolean function with random variables, Internat. J. Systems Sci. 5 (6) (1974) 545–553. [6] B. Alidaee, Optimal ordering policy of a sequential model, J. Optim. Theory Appl. 83 (1994) 199-205.

[7] R.W. Butterworth, Some reliability fault-testing models, Oper. Res. 20(1972) 335-342.

[8]P. Jedrzejowicz, Minimizing the average cost of testing coherent systems: complexity and approximate algorithms, IEEE Trans. Reliab. R-32(1) (1983) 66-70

[9] L.A. Cox Jr., Y. Qiu, W. Kuehner, Heuristic least-cost computation of discrete classification functions with uncertain argument values, Ann. Oper. Res. 21 (1989) 1–21

[10] S.O. Duffuaa, A. Raouf, An optimal sequence in multicharacteristics inspection, J. Optim. Theory Appl. 67 (1) (1990) 79–87.

[11] H.A. Simon, J.B. Kadane, Optimal problem-solving search: all-or-none solutions, Arti7cial Intelligence 6 (1975) 235–247.

[12] J.B. Kadane, Quiz show problems, J. Math. Anal. Appl. 27 (1969) 609–623.

[13] W.B. Joyce, Organizations of unsuccessful R&D projects, IEEE Trans. Engrg. Manage. EM-18 (2) (1971) 57–65.

[14] C.L. Monma, Sequencing with general precedence constraints, Discrete Appl. Math. 3 (1981) 137–150.

[15] H.A. Simon, J.B. Kadane, Optimal problem-solving search: all-or-none solutions, Artifcial Intelligence 6 (1975) 235–247.

[16] R. Greiner, Finding optimal derivation strategies in redundant knowledge bases, Artificial Intelligence 50 (1990) 95–115

[17] Ünlüyurt T.,Sequential testing of complex systems: a review, Discrete Applied Mathematics142 (2004) 189-205

.[18] D. Dubois, M.P. Wellman, B. D'Ambrosio, P. Smets (Eds.), Guess-and-verify heuristics for reducing uncertainties in expert classification systems, uncertainty in artificial intelligence, Proceedings of the Eighth Conference, Morgan Kaufman, Los Altos,CA, 1992.

[19] Boros, E., Ünlüyurt, T., Sequential testing of series-parallel systems of small depth, In: Computing Tools for Modeling, Optimization and Simulation, 39-74, 2000, Laguna and Velarde eds., Kluwer Academic Publishers, Boston.

[20] Reyck, B., Leus, R., R&D project scheduling when activities may fail, IIE Transactions (2008) 40, 367–384

APPENDIX: NUMBER OF INSTANCES FOR WHICH EACH ALGORITHMS PRODUCES BEST RESULTS

A-1 Number of instances for which algorithms produces best result n=40 k<=6(all strategies)

k	int,minc/p	int,Rand	int,MI	perm-c/p	perm,c	Min merit	c/p LS	c-LS
1	40	40	40	0	0	0	1	0
2	5	14	31	0	0	0	0	1
3	2	5	35	0	0	0	0	0
4	0	8	32	0	0	0	0	0
5	0	11	29	0	0	0	0	0
6	0	9	29	0	0	0	0	2

A-2 Number of instances for which algorithms produces best result n=60 k<=5(all strategies)

k	int,minc/p	int,Rand	int,MI	perm-c/p	perm,c	Min merit	c/p LS	c-LS
1	40	40	40	0	0	0	0	0
2	0	5	34	0	0	0	1	0
3	1	12	27	0	0	0	0	0
4	1	10	29	0	0	0	0	0
5	0	8	32	0	0	0	0	0

A-3 Number of instances for which algorithms produces best result n=80 k<=4(all strategies)

k	int,minc/p	int,Rand	int,MI	perm-c/p	perm,c	Min merit	c/p LS	c-LS
1	40	40	40	1	0	1	1	1
2	1	9	32	0	0	0	0	0
3	0	7	33	0	0	0	0	0
4	0	9	31	0	0	0	0	0

A4.

A-4 Number of instances for which algorithms produces best result n=100 k<=3(all strategies)

k	int,minc/p	int,Rand	int,MI	perm-c/p	perm,c	Min merit	c/p LS	c-LS
1	40	40	40	1	0	1	1	1
2	1	16	25	0	0	0	0	0
3	0	13	27	0	0	0	0	0

A-5. Number of instances for which algorithms produces best result n=40 k>6(permutation strategies)

k	perm-c/p	perm,c	c/p LS	c-LS
7	2	0	6	34
8	5	0	11	29
9	3	0	11	29
10	14	0	21	20
11	8	0	14	26
12	6	0	11	30
13	9	0	16	27
14	4	0	12	28
15	5	0	10	30
16	10	0	14	27
17	4	0	16	24
18	6	0	10	32
19	3	0	13	29
20	4	1	20	20

A-6. Number of instances for which algorithms produces best result n=60 k>5(permutation strategies)

k	perm-c/p	perm,c	c/p LS	c-LS
6	5	0	12	28
7	5	0	14	26
8	10	0	20	20
9	3	0	13	27
10	4	0	8	32
11	5	0	11	29
12	1	0	3	37
13	3	0	10	31
14	6	0	14	26
15	4	0	16	24
16	4	0	17	23
17	3	0	11	29

18	2	0	12	28
19	4	0	12	28
20	6	0	13	27
21	2	0	8	32
22	3	0	8	32
23	2	0	7	33
24	1	0	10	30
25	4	0	11	29
26	4	0	7	33
27	3	0	9	31
28	2	0	10	30
29	0	0	9	31
30	2	0	16	24

A-7. Number of instances for which algorithms produces best result n=80 k>4(permutation strategies)

k	perm-c/p	perm,c	c/p LS	c-LS
5	4	0	12	28
6	4	0	12	29
7	4	0	15	25
8	2	0	7	33
9	2	0	12	28
10	2	0	13	27
11	2	0	13	27
12	6	0	15	25
13	5	0	13	27
14	4	0	13	27
15	4	0	16	24
16	2	0	12	28
17	0	0	13	27
18	1	0	12	28
19	3	0	16	24
20	4	0	10	30
21	3	0	12	28
22	0	0	15	25
23	2	0	17	23
24	2	0	12	28
25	2	0	14	26
26	3	0	15	25
27	1	0	11	29
28	1	0	12	28
29	2	0	8	32
30	6	0	14	26
31	2	0	8	32
32	1	0	9	31
33	3	0	9	31
34	1	0	14	26

35	3	0	10	30
36	1	0	8	32
37	3	0	14	26
38	2	0	11	29
39	6	0	17	23
40	1	0	15	25

A-8. Number of instances for which algorithms produces best result n=100 k>3(permutation strategies)

k	perm-c/p	perm,c	c/p LS	c-LS
4	7	0	16	25
5	2	0	11	29
6	5	0	9	31
7	5	0	11	29
8	2	0	13	27
9	4	0	16	24
10	3	0	12	28
11	2	0	14	26
12	2	0	14	27
13	6	0	13	27
14	3	0	8	32
15	3	0	9	31
16	3	0	10	30
17	1	0	14	26
18	3	0	11	29
19	1	0	14	26
20	2	0	16	24
21	4	0	15	25
22	6	0	17	23
23	1	0	8	32
24	2	0	15	25
25	1	0	14	26
26	2	0	19	21
27	1	0	10	30
28	3	0	18	22
29	1	0	10	30
30	3	0	18	22
31	3	0	12	28
32	1	0	13	27
33	1	0	13	27
34	2	0	14	26
35	4	0	16	24
36	2	0	10	30
37	2	0	13	27
38	2	0	11	29
39	0	0	10	30

40	16	0	7	17
41	1	0	10	30
42	0	0	8	32
43	0	0	10	30
44	3	0	8	32
45	1	0	9	31
46	2	0	13	27
47	3	0	14	26
48	3	0	12	28
49	0	0	11	29
50	1	0	12	28