SEQUENTIAL TESTING OF SERIES PARALLEL SYSTEMS

by

GÜRKAN IŞIK

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

January 2013

SEQUENTIAL TESTING OF SERIES PARALLEL SYSTEMS

APPROVED BY

Assoc. Prof. Dr. Tonguç Ünlüyurt .....

(Thesis Supervisor)

Assoc. Prof. Dr. Bülent Çatay .....

Assoc. Prof. Dr. Barış Selçuk .....

DATE OF APPROVAL: 13/01/2014

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Tonguç Ünlüyurt for their enthusiasm, inspiration, guidance, patience and motivation. Without their invaluable suggestions it would be impossible to complete this thesis.

I am grateful to my friends for their continuous guidance, emotional and academic support.

ABSTRACT

In this thesis, we study the sequential testing problem of 3-level deep Series Parallel systems (SPS). We assess the performance of depth-first permutation (DFP) algorithm that has been proposed in the literature. DFP is optimal for 1-level deep, 2-level deep SPSs and 3-level deep SPSs that consist of identical components. It can be used to test general SPSs. We report the first computational results regarding the performance of DFP for 3-level deep SPSs by comparing its performance with a dynamic version of DFP and a hybrid simulated annealing-tabu search algorithm that we developed. In order to implement the algorithms, we propose an efficient method to compute the expected cost of a permutation strategy. The results of computational experiments for this algorithm and other algorithms proposed in the literature are reported.

ÖZET

Bu tezde 3 seviyeli seri-paralel sistemlerde (SPS) ardışık test problemi üzerine odaklanılmıştır. Literatürde önerilen derin öncelikli (DFP) algoritmanın performansı değerlendirilmiştir. Bu algoritma 1 seviyeli ve 2 seviyeli sistemler ve 3 seviyeli özdeş bileşenli sistemler için en iyi çözümü vermektedir. DFP algoritması genel SPS'leri test etmek için kullanılabilir. Bu çalışma, DFP algoritmasının performansını değerlendiren ilk hesaplamalı çalışmadır. DFP algoritmasının performansı, dinamik versiyon DFP algoritması ve geliştirdiğimiz melez tavlama benzetimi-tabu araması algoritması ile karşılaştırmalı olarak sunulmuştur. Bu algoritmaların uygulanabilmesi için permütasyon stratejilerin beklenen maliyetini etkin hesaplayan bir metot önerilmiştir. Hesaplamalı deneyler gerçekleştirilmiş ve sonuçları sunulmuştur.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

DFP: Depth-First-Permutation

SA: Simulated Annealing

SAPATS: Simulated Annealing with Post Analysis Tabu Search

SPS: Series-Parallel Systems

TS: Tabu Search

BDT: Binary Decision Tree

LIST OF FIGURES

LIST OF TABLES

NOTATION

c_i: cost of testing component i

p_i: working probability of component i

q_i: failing probability of component i

$\wedge$: logical AND

$\vee$: logical OR

1. INTRODUCTION

1.1. Sequential Testing Problem

Sequential Testing problem involves finding a minimum expected cost strategy to evaluate a Boolean function when learning the values of the variables are costly. The variables assume values independent of each other and the probabilities of each variable taking a value of 1 or 0 are also known.

Sequential testing problem arises in different application areas such as query optimization in databases [9], medical diagnosis [15], project management [4], inspection in manufacturing [11] etc.

In this particular study, we concentrate on a special class of Boolean functions that correspond to Series-Parallel systems. We will define the Sequential Testing problem in a precise manner in the context of Series-Parallel systems in the next section. First we define Series-Parallel systems in this section.

To the best of our knowledge this is the first computational study on 3-level deep SPSs (Level concept is presented in Section 1.2). We show that the expected cost of any permutation strategy can be computed efficiently for 3-level deep SPSs. This efficient cost calculation method provides us to apply metaheuristic methods for sequential testing problem. Our contributions are presented as detailed at the end of Section 2.

1.2. Series-Parallel Systems

A Series-Parallel system (SPS) is a special multi-component system where each of the components can be in working or failing state. The state of the SPS depends on the states of the components via a special Boolean function. We will refer to this Boolean function as the structure function of the SPS. The structure function of an SPS is the Boolean function that maps the states of the components to the state of the system. The variables of the Boolean function correspond to the components of the

system, and the value of the Boolean function will represent the state of the SPS. The simplest SPS is a simple series system or a simple parallel system. A simple series system is in working state if all of its components are working and dually a parallel system is in working state if at least one of its components is working. The structure function of a simple series system and simple parallel system are as follows:

$$f(x_1, x_2, \dots, x_n) = x_1 \wedge x_2 \wedge \dots \wedge x_n \quad \text{(simple series system)}$$

$$f(x_1, x_2, \dots, x_n) = x_1 \vee x_2 \vee \dots \vee x_n \quad \text{(simple parallel system)}$$

where $\wedge$ is the logical AND and $\vee$ is the logical OR operator.

We can depict the simple series and simple parallel systems graphically as shown in Figure 1.

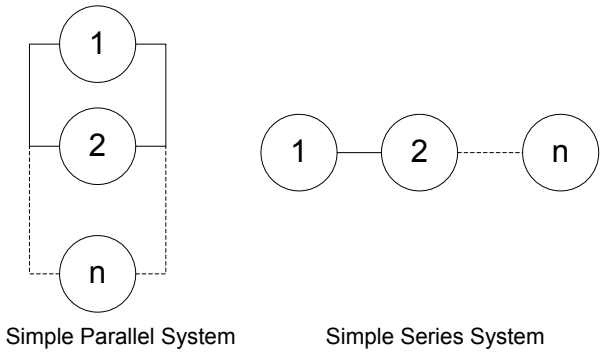


Figure 1. Simple parallel and simple series systems

More complicated SPSs can be constructed by a series or parallel connection of other SPSs. These other SPSs are called the sub-systems of the SPS. The structure function of a series (parallel) connection of some SPSs is the AND (OR) of the structure function of these SPSs.

For instance, the SPS shown in Figure 2 is a series connection of a component and two simple parallel systems whereas the second SPS is a parallel connection of four components.

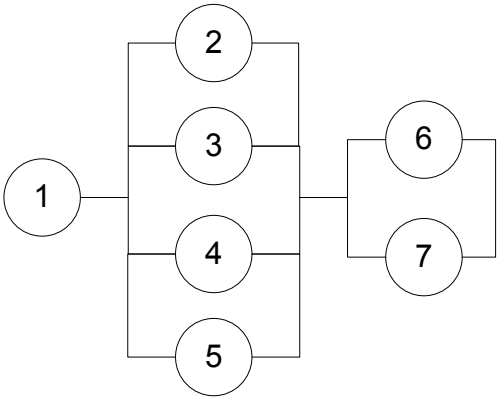


Figure 2. An example series system

The SPS shown in Figure 3 is generated by connecting a simple series system consisting of components 8 and 9, with the SPS given in Figure 2, in parallel. SPS shown in Figure 2 is a "subsystem" of the SPS shown in Figure 3.

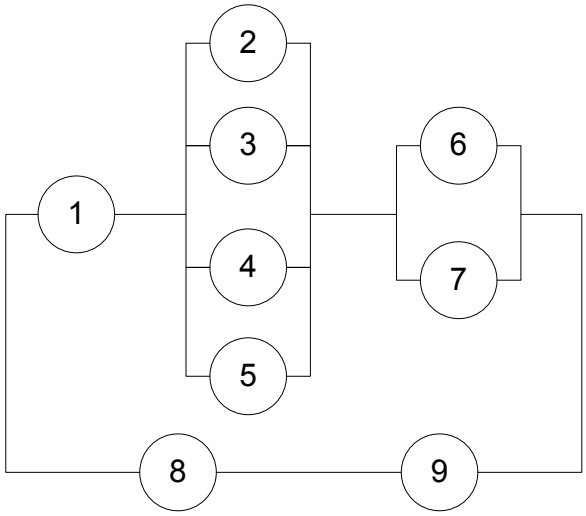


Figure 3. An example parallel system generated from the SPS given in Figure 2

Definition 1: An SPS has "l" level deep if the maximum number of reductions is "$l-1$" to reach a simple series/parallel system.

Figure 4 shows a 3-level deep SPS which consists of parallel connection of a 2-level deep subsystem and component 4. The 2-level deep subsystem consists of a 1-level deep parallel subsystem and component 3.

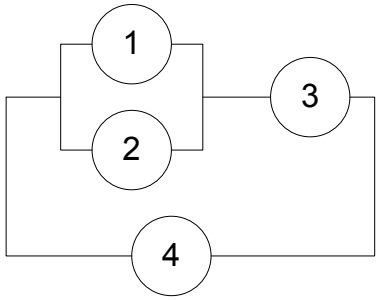


Figure 4. An example 3-level deep SPS

The structure function of the SPS given in Figure 4 in logical form can be written as;

$$f(x_1, x_2, x_3, x_4) = \Big(\big((x_1 \vee x_2) \wedge x_3\big) \vee x_4\Big)$$

Every SPS can be represented as an AND-OR tree. The leaves of the AND-OR tree are indexed by the components of the SPS. The internal nodes describe the type of the connection of its children (series or parallel). The length of the longest path from the root to a leaf node is the depth of SPS. The AND-OR tree representation of the SPS in Figure 4 is given in Figure 5. The simple series and simple parallel systems have deep 1.

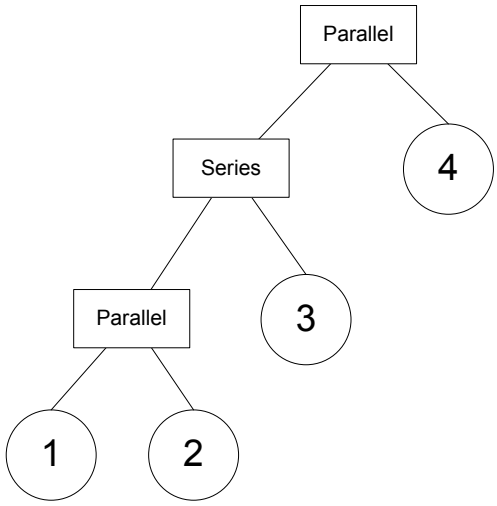


Figure 5. Tree representation of the SPS given in Figure 4

The structure function of any SPS is a special type of Boolean function referred to as Read-Once function. The function is called Read-Once since the Boolean function can be represented in a form where each variable appears exactly once. For instance the structure function of the SPS shown in Figure 3 can be written as;

$$f(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9) = \Big(\big(x_1 \wedge (x_2 \vee x_3 \vee x_4 \vee x_5) \wedge (x_6 \vee x_7)\big) \vee (x_8 \wedge x_9)\Big)$$

Read-Once functions have been extensively studied in the literately for other reasons [9].

2. PROBLEM DEFINITION

As indicated in Section 1, Sequential Testing problem for an SPS requires finding a strategy that finds out the correct state of the SPS with the minimum expected cost. Given the structure function of an SPS, a cost vector C, where c_i is the cost of learning the correct state of component i, and a probability vector P where p_i is the probability that components is in working state, a feasible strategy outputs the next component to test given the states of previously tested components. Testing a component is used interchangeable with learning the value of the corresponding variable in the structure function.

For a simple series system a feasible strategy is just a permutation of the components since we stop testing if any component is in failing state or all components are tested. Graphically we can show a feasible solution as in Figure 6.

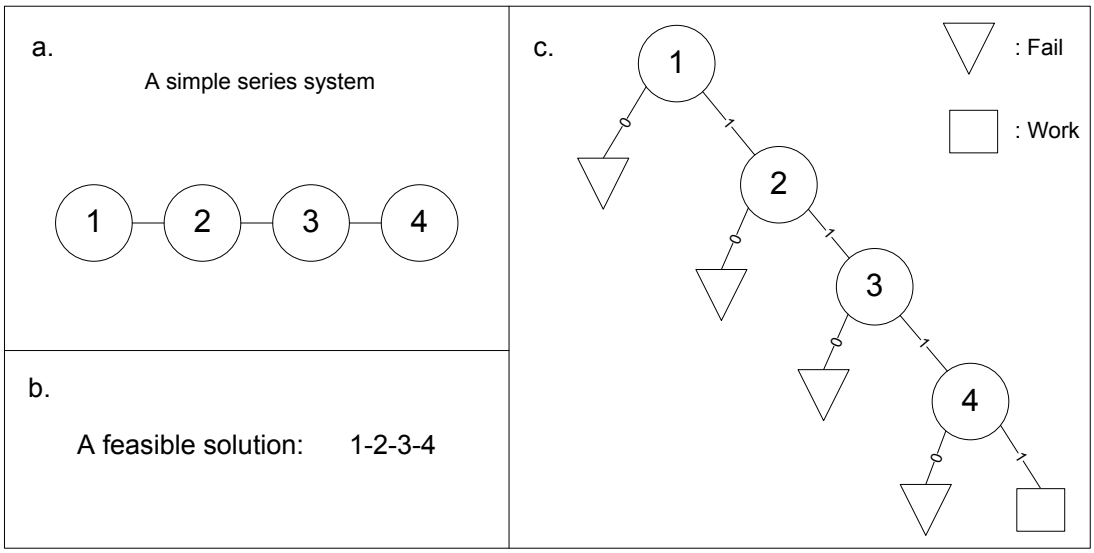


Figure 6. (a) a simple series system (b) A feasible strategy (c) BDT representation of the strategy

Expected cost of the strategy in Figure 6 can be written as follows.

$$E[Cost] = c_1 + (p_1)c_2 + (p_1p_2)c_3 + (p_1p_2p_3)c_4$$

In general, a strategy can be represented as a binary decision tree (BDT). Each internal node corresponds to a variable or component. The right (left) branch of a node corresponds to the case when the variable is 1(0) or the component is in working state. The leaves of the BDT correspond to the state of the SPS or the value of the structure function. For instance, the strategy 3-1-2-4 for the SPS in Figure 4 is shown in Figure 7.

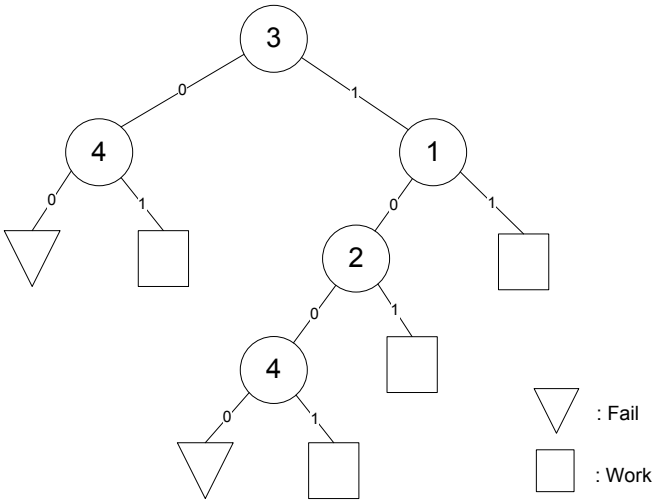


Figure 7. An inspection strategy for the SPS shown in Figure 4

We can compute the expected cost of a strategy in two ways. One is finding the probability of a component being tested and summing up the expected testing costs of all components. The other is to sum up the expected costs of all root-to-leaf paths of the BDT. We illustrate these two methods for the strategy shown in Figure 7 as follows.

$$E[Cost] = (\,p_3)c_1 + (p_3q_1)c_2 + c_3 + \,(q_3 + p_3q_1q_2)c_4$$

We can also write the following recursive equation in order to compute the expected cost of this strategy.

$$E[Cost] = c_3 + (q_3c_4) + \Big(p_3\Big(c_1 + q_1\big(c_2 + (q_2c_4)\big)\Big)\Big)$$

In fact, we don't need to have the whole tree at hand in order to execute the strategy. An algorithm that tells us which component to test next given the states of the previously tested components suffices to execute the strategy.

In general an optimal BDT describing the whole strategy can have exponential size in terms of the input size. Obviously, for simple series and parallel systems this is not the case there are also exceptions to this for systems other than SPSs. For instance, it is shown in [18] that an optimal strategy for testing k out of n systems can be stored in $O(n^2)$ space. Although we don't need to have the whole strategy tree to execute the strategy but we need to know which component to test next given the states of tested components, in order to compute the expected cost of the tree, we may need the whole tree.

On way to avoid this exponential growth is to consider a subset of the strategies that are easy to describe. One example is permutation strategies where we test the components according to a permutation as long as they can affect the state of the system. Although, the whole strategy can be described efficiently in this case, the computation of the expected cost may still be a problem. We will refer to a strategy that cannot be described by a permutation strategy non-permutation (or dynamic) strategy.

In this thesis, distinction between permutation and dynamic strategies will be important so we describe these concepts in detail via examples.

2.1. Solution Strategies

2.1.1. Permutation Strategies

Permutation strategies are static solutions. It means that, the testing order of components does not alter during the inspection. These types of strategies can be described a permutation of the components. So it is easy to store and execute these strategies.

Inspection steps of an example permutation strategy 1-2-3-4 for the SPS in Figure 4 are given in Figure 8. This is a permutation strategy because testing order does not differ during inspection. For example component 3 is tested before component 4 in all paths. If component 1 turns out to be in failing condition, component 2 becomes redundant and has no effect on the state of the SPS. So in this case, we continue testing the next component in the permutation which is component 3.

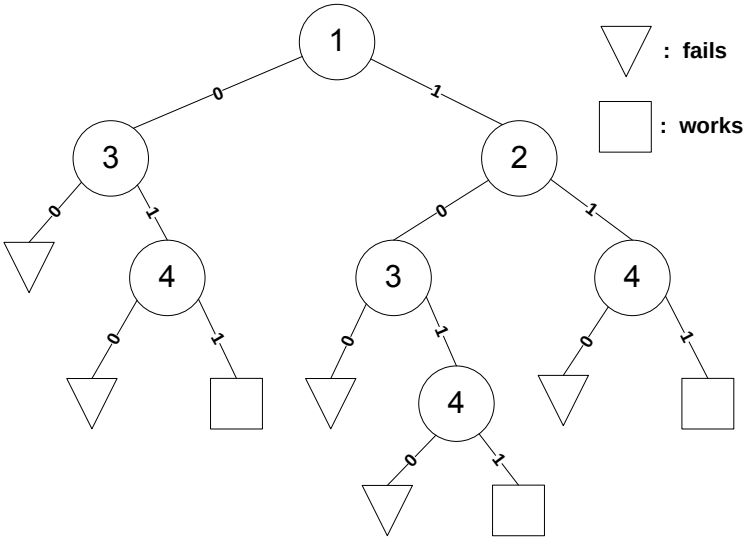


Figure 8. BDT representation of inspection steps of a permutation strategy

2.1.2. Nonpermutation Strategies

Nonpermutation strategies are dynamic solutions. It means that, the testing order of components depends on the results of the tests performed. These types of strategies can be represented as a BDT. Inspection steps of an example nonpermutation strategy for the SPS in Figure 4 are given in Figure 9. This is a nonpermutation strategy because component 4 is tested before component 3 if component 1 fails; on the other hand component 3 is tested before component 4 if component 1 works.

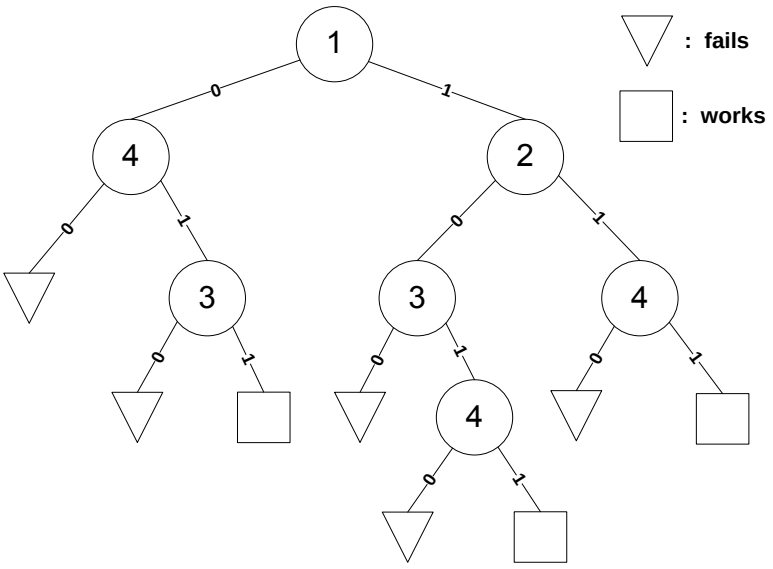


Figure 9. BDT representation of inspection steps of a nonpermutation strategy

2.2. Properties

Observation 1: Combining more than one SPS does not always generate a deeper SPS.

Let's explain Observation 1 with an example. Assume that we have two SPSs having structure functions $f_A(x_1, x_2, x_3) = \big((x_1 \wedge x_2) \vee x_3\big)$ and $f_B(x_4, x_5) = (x_4 \wedge x_5)$.

If we create a new parallel system by combining A and B, the new system C will have 2-level deep (same with system A). The function will be $f_C(x_1, x_2, x_3, x_4, x_5) = \big((x_1 \wedge x_2) \vee x_3 \vee (x_4 \wedge x_5)\big)$. Alternatively, if we create a new series system by combining A and B, the new system C will have 3-level deep (larger than A and B). The function will be $f_C(x_1, x_2, x_3, x_4, x_5) = \Big(\big((x_1 \wedge x_2) \vee x_3\big) \wedge x_4 \wedge x_5\Big)$.

An SPS is a connection of smaller SPSs and can be represented with Read Once structure functions [16]. For instance, the system given in Figure 10 is not an SPS [6].

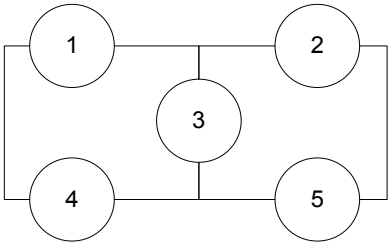


Figure 10. A non series-parallel system

Duality concept is described in [6]. Dual system can be derived by switching the parallel and series signs in any representation. If the SPS is parallel (series) then the dual system is series (parallel). The SPS and the dual system are same level. Let's show the dual of the SPS in Figure 4. Dual system will be obtained as $f(x_1, x_2, x_3, x_4) = \Big(\big((x_1 \wedge x_2) \vee x_3\big) \wedge x_4\Big)$. Figure 11 shows dual system.

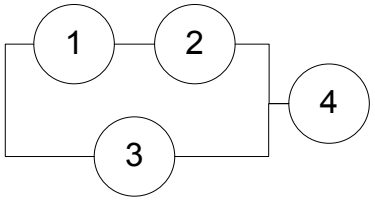


Figure 11. Dual of the SPS given in Figure 4

Duality provides us a very important feature; algorithms and results of a sequential testing problem for an SPS can be easily translated for its dual system. It means that when we solved a sequential testing problem of an SPS, we have already solved the sequential testing problem of dual system [6].

In this work, we concentrate on general 3-level deep SPSs and try to analyze the performance of various approaches from a computational point of view.

Our contributions can be summarized as follows:

- To the best of our knowledge this is the first computational study on 3-level deep SPSs.
- We show that the expected cost of any permutation strategy can be computed efficiently for 3-level deep SPSs.
- We compare the performance of DFP by an extension of DFP that is dynamic in nature and never produces strategies that are worse than DFP in terms of expected cost.
- We develop a special simulated annealing-tabu search based algorithm by using properties of 3-level deep SPSs and analyze how much improvement can be made starting at a DFP solution.

3. LITERATURE REVIEW

The sequential testing problem have a wide area of applications including healthcare (testing patients against some dangerous disease), telecommunication (testing stability systems, connectivity of networks), artificial intelligence (finding optimal derivation strategies in knowledge bases, testing search algorithms), manufacturing (testing machines before delivery, testing for replacement in technical service centers), design of screening procedures,. The inspection of the system is usually repeated many times in real life so it is important to minimize the total cost in the long run [18].

In this literature review, we don't intend to provide a complete review of Sequential Testing applications and solution algorithms. Rather, after describing a couple of examples, we will review the results for SPS systems in detail.

Doctors determine whether their patients has disease by making some tests, each test has an associated cost and confidence level. They can diagnose a disease by making one test or making some combinations of several tests. Minimum expected test cost can be found by solving a sequential testing problem of [15]. Although many articles on sequential testing, motivate their problems by using medical diagnosis examples (see e.g. Greiner [15]), medical diagnosis problem has many different aspects. Still a diagnosis strategy can be described by a tree.

It is important to protect the functionality of complex systems such as electricity distribution systems, nuclear power plants etc. from adaptive threats. Attackers can adapt their strategies by analyzing the defense of possible targets. The defense levels of possible targets change the expended effort and success probability of attacks. Investment can increase the defense levels of these points. Investments can be planned by willing to make the attacks as costly as possible. The defender wants to increase the minimum expected effort of an attack. Finding the attack having minimum expected effort can be another motivation for sequential testing problem [12].

In the literature, polynomial time algorithms to find optimal strategies for 1-level, 2-level deep SPSs and 3-level deep SPSs having identical (testing costs and working probabilities are the same for all) components are provided. There is no study which focuses to 3-level deep or more complex SPSs optimally. Solution methods are proposed for general SPSs and they don't guarantee high quality solutions.

In literature, precedence constraints are considered as extension of sequential testing problem. The researchers have examined different classes of testing policies for 1-level systems under general precedence constraints. Dynamic programming and branch-and-bound algorithms are suggested for solution. The dynamic programming has memory limitation, branch-and-bound algorithm does not have the limitation of memory issues, but it is limited in the size of the instances [20].

Most results in the literature are for the case when we have a simple series or parallel system. Chiu et al. [17] provide an optimal algorithm for parallel precedence constraints, for series or parallel systems. Garey in [13] gives a polynomial time optimal algorithm that works for the series case under forest-type precedence constraints. Berend et al. [5] also present similar result with Garey for 1-level systems with general type of precedence by using object detection and acceptance testing as motivation. They argue the runtime as cost, introduce mathematical models and give complexity of solution methods. There is no further study to solve 2-level deep or more complex SPSs under precedence constraints.

There is limited number of studies about sequential testing problem in literature. Generally theoretical studies have made and there is no extensive computational benchmark study. Some solution methods are proposed but the solution qualities are not analyzed. In this study we have focused to 3-level deep SPSs and implemented some solution strategies which are offered for general SPSs in the literature. We have also developed a new method and compared the results on randomly generated instances.

4. SOLUTION APPROACHES

4.1. Depth First Permutation (DFP)

The DFP strategy is an intuitive algorithm generalizing the optimal strategies for simple series and simple parallel systems and it is proposed by several studies (see [6][18][15][12][21]) and [10]). DFP produces a strategy for any SPS. The strategy produced by DFP is a permutation strategy and it is optimal for 1-level, 2-level deep SPSs and 3-level deep SPSs with identical components. A strategy for a 1-level deep SPS is simply a permutation of the components since for instance for a series system testing stops as soon as a failing component has been found. The optimal permutation is the non-decreasing order of c_i/q_i which is quite intuitive. We would like to test component that are more likely to fail and cheap to test first. That is quantified by the ratio c_i/q_i. This can be proved by a simple exchange argument. (Similarly an optimal permutation for a parallel system is the non-decreasing order of c_i/p_i) DFP is an intuitive generalization of this strategy for more general SPSs. Mainly, DFP recursively replaces the subsystem at the lowest level of the SPS (which is a simple series or simple parallel system) by a single component whose testing cost is the optimal expected cost of testing that subsystem and whose probability of functioning is the probability that the subsystem functions. When a subsystem is replaced by a component, this means that the components of that subsystem will be tested one after another. At the end of this recursive process, we end up with a simple parallel system or simple series system whose components correspond to some subsystems of the SPS. Then the DFP strategy is to test these subsystems one by one in the corresponding optimal order.

Theorem 1: DFP is optimal for 1-level deep SPSs, 2-level deep SPS and 3-level deep SPSs that consist of identical (testing costs and working probabilities are the same for all) components. See [6][18][15][21] for the proof of the theorem.

DFP solution is not optimal for general 3-level deep SPSs and 4-level deep SPSs with identical components [6]. There are SPS instances where this algorithm can

behave very badly. For instance, [19] reports a construct where the algorithm misses the optimal solution by any constant. A similar result is presented in [15] for 3-level deep small sized SPSs. In this strategy, once we start testing a subsystem, we never switch to another subsystem before we determine the former subsystem is working or failing. We refer to this algorithm Depth first Permutation (DFP) since it starts testing the subsystem with the best ratio and switches to the next subsystem after determining the state of the current subsystem.

Figure 8 shows a depth-first permutation strategy (1-2-3-4) because it tests the component 3 after determining the state of 1-level deep parallel system $(1 \vee 2)$ and tests component 4 after determining the state of 2-level deep series system $((1 \vee 2) \wedge 3)$. On the other hand the strategy shown in Figure 7 is permutation because it can be represented as 3-4-2-1 but not depth-first. It starts with component 3 but tests component 4 while the state of 2-level deep series system $((1 \vee 2) \wedge 3)$ is not determined.

Some properties of the strategy produced by DFP can be summarized as follows:

- The DFP algorithm produces a permutation strategy [15].
- DFP produces a strategy that has the lowest cost among all depth-first strategies [15].
- These strategies do not switch from one subsystem to another before the current subsystem has been resolved.
- It is very easy to obtain these strategies and to compute their expected cost.

In the literature different pseudo-codes can be found for DFP (see [6][15][12]), we present the one which we implemented for 3-level deep SPSs, since this study concentrates on 3-level deep SPSs. The pseudo-code of DFP is as follows.

Definitions

$E[C]$: Expected cost of testing given 3-level SPS

P: Working probability of given SPS

S: permutation solution found by algorithm

c: cost vector

p: working probability vector

q: failing probability vector (q = 1-p)

Algorithm

partial sequence(*) of component i is $\pi_i = \{i\}$

L = 1

WHILE L <= *3*

  label L level deep subsystems with the index j

  REPEAT

    IF subsystem j is a parallel system

    THEN

      label all elements of system j with the index i

      sort components in non-decreasing order of c_i/p_i

      relabel components according to order with the index $i' \in \{k_1, k_2, \dots, k_t\}$

      calculate $c_j = \sum_{a=1}^{t}(q_{k_1} \dots q_{k_{a-1}} c_{k_a})$ and $p_j = 1 - q_j = q_{k_1} q_{k_2} \dots q_{k_t}$

      find sequence $\pi_j = (\pi_{k_1}, \pi_{k_2}, \dots, \pi_{k_t})$

    ELSE

      label all elements of system j with the index i

      sort components in non-decreasing order of c_i/q_i

      relabel components according to order with the index $i' \in \{k_1, k_2, \dots, k_t\}$

      calculate $c_j = \sum_{a=1}^{t}(p_{k_1} \dots p_{k_{a-1}} c_{k_a})$ and $p_j = p_{k_1} p_{k_2} \dots p_{k_t}$

      find sequence $\pi_j = (\pi_{k_1}, \pi_{k_2}, \dots, \pi_{k_t})$

    ENDIF

    convert system j to the equivalent component i'' having cost $c_{i''} = c_j$, working probability $p_{i''} = p_j$ and partial sequence $\pi_{i''} = \pi_j$

  UNTIL all the L level deep subsystem j's are examined

  IF *L = 3*

  THEN

    $E[C] = c_j$, $P = p_j$ and S = π_j

  ENDIF

  increment L

ENDWHILE

PRINT $E[C]$, P and S.

(*) Partial sequence: It implies the scheduled part of the solution. For example if we decided to test the components i and j in the order i-j, the partial sequence is {i,j}. The final solution consists of partial sequences. In order to initialize the algorithm, partial sequences are defined for all components.

The time complexity of DFP algorithm is polynomial in number of components and number of subsystems so it can solve big instances in reasonable time. We have tried to solve randomly generated instances having 100 components and having subsystems between 35 and 49. The DFP algorithm solves an instance in less than one second.

Example 1: Let's find the DFP solution for the SPS given in Figure 12.

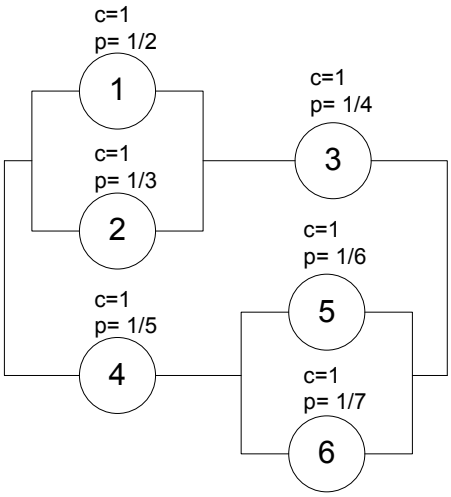


Figure 12. An example 3-level deep parallel system

Iteration 1: Let's label the 1-level subsystems. Label the parallel subsystem as A which includes components 1 and 2 ($A = (1 \vee 2)$). Label the parallel subsystem as B which includes components 5 and 6 ($B = (5 \vee 6)$).

Subsytem A:

$c_1/p_1 = 2$ and $c_2/p_2 = 3$ so non-decreasing c_i/p_i order is: B-3. Hence,

$\pi_A = \{1,2\}$, $c_A = 1 + 1/2 \times 1 = 3 / 2$ and $p_A = 1 - 2 / 3 \times 1 / 2 = 2 / 3$

Subsytem B:

$c_5/p_5 = 6$ and $c_6/p_6 = 7$ so non-decreasing c_i/p_i order is: 5-6. Hence,

$\pi_B = \{5 , 6\}$, $c_B = 1 + 5 / 6 \times 1 = 11 / 6$ and $p_B = 1 - 5 / 6 \times 6 / 7 = 2 / 7$

Iteration 2: All 1-level subsystems was evaluated so continue with 2-level subsystems. Label the series subsystem as C which includes subsystem A and

component 3 ($C = (A \wedge 3)$). Label the series subsystem as D which includes component 4 and subsystem B ($D = (4 \wedge B)$).

Subsytem C:

$c_A/q_A = 9/2$ and $c_3/q_3 = 4/3$ so non-decreasing c_i/q_i order is: 3-A. Hence,

$\pi_C = \{3\,, 1-2\}\,, c_C = 1 + 1\ /\ 4 \times 3\ /\ 2 = 11\ /\ 8$ and $p_C = 2\ /\ 3 \times 1\ /\ 4 = 1\ /\ 6$

Subsytem D:

$c_4/q_4 = 5/4$ and $c_B/q_B = 77/30$ so non-decreasing c_i/q_i order is: 4-B. Hence, $\pi_D = \{4\,, 5-6\}$, $c_D = 1 + 1\ /\ 5 \times 11\ /\ 6 = 41\ /\ 30$ and $p_D = 1\ /\ 5 \times 2\ /\ 7 = 2\ /\ 35$

Iteration 3: All 2-level subsystems was evaluated so continue with 3-level system and label as E ($E = (C \vee D)$).

System E:

$c_C\ /\ p_C = 66/8$ and $c_D/p_D = 1435/60$ so non-decreasing c_i/p_i order is: C-D. Hence, $\pi_E = \{3\,, 1-2\,, 4\,, 5-6\}$, $c_E = 11/8 + 5/6 \times 41/30 = 2.51$ and $p_E = 1 - \ 5\ /\ 6 \times 32\ /\ 35 = 0.24$

Solution: The DFP solution is 3-1-2-4-5-6 and the expected cost of this strategy is 2.51. The BDT representation of the steps of this solution is given in Figure 13.

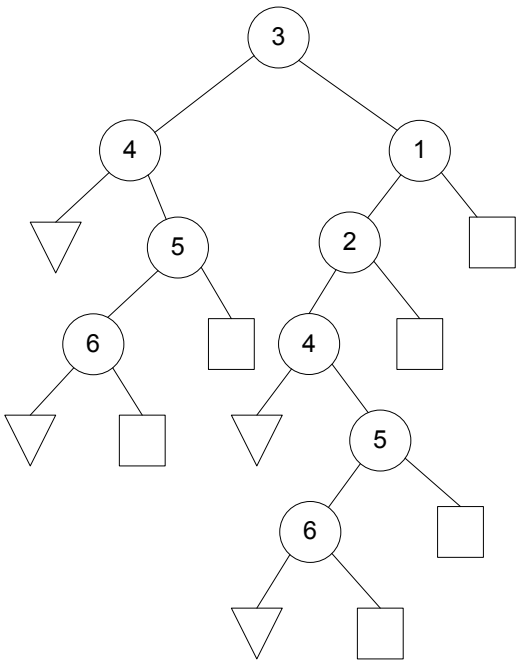


Figure 13. BDT representation of DFP solution for the SPS shown in Figure 12

4.2. Depth first Dynamic (DF-D)

Depth first Dynamic (DF-D) algorithm is the dynamic version of DFP algorithm. As a matter of fact, one could improve the DFP algorithm by recomputing all ratios after determining the next component to test [19]. It can give the same strategy and result with DFP for some instances but it has a potential to give better results. For the cases that DF-D will improve on DFP, we know that the strategy produced by DF-D will not be a permutation startegy. This is because in order for DF-D to produce a different strategy than DFP, it should be the case that we switch from one subsystem to another after recomputing all ratios in some step of the algorithm.

The results of DF-D algorithm may not be given as permutation so they should be represented in BDT representation. This algorithm updates the current SPS at each node, it calculates the ratios and tests the component having smallest ratio.

The DF-D algorithm can give better results than DFP but it needs more computing effort and time than DFP. Moreover, executing the strategy found by DF-D is less convenient than executing the strategy found by DFP.

Observation 2: DF-D finds same solution with DFP for 1-level, 2-level SPSs and 3-level SPSs having identical components.

Observation 3: DF-D finds permutation solutions for 1-level, 2-level SPSs and 3-level SPSs having identical components.

The pseudo-code of implemented algorithm is as follows:

Definitions

$E[C]$: Expected cost of testing given SPS

P: Working probability of given SPS

c: cost vector

p: working probability vector

q: failing probability vector (q = 1-p)

TREE: list of nodes which are not examined

Node: a solution element which consist of an "SPS", "cost", "probability", and "tested" fields

Algorithm

```
Create node N0
Assign given SPS to N0.SPS
Run DFP for N0.SPS
N0.tested = id of component which is tested first by DFA
N0.cost = tested component's cost
N0.probability = 1
Add N0 to TREE
E[C] = N0.cost
P=0
WHILE TREE is not empty
  Label the first node of TREE as N0
  BLOCK
    Create a node and label it as N1 for N0.tested fails
    N1.probability = N0.probability * q_{N0.tested}
    Update N0.SPS when N0.tested fails and assign this SPS to N1.SPS
    IF N1.SPS is not empty
    THEN
      Run DFP for N1.SPS
      IF level of N1.SPS is less than or equal to 2
      THEN
        N1.cost = N1.probability * cost of DFP solution
        N1.probability = N1.probability * working probability of DFP
                         solution
        P = P + N1.probability
      ELSE
        N1.tested = The component's id which is tested first by DFA
        N1.cost = N1.probability * N1.tested
        Insert N1 to TREE at just behind of N0
      ENDIF
      E[C] = E[C] + N1.cost
    ENDIF
  ENDBLOCK
  BLOCK
    Create a node and label it as N2 for N0.tested works
    N2.probability = N0.probability * p_{N0.tested}
    Update N0.SPS when N0.tested works and assign this SPS to N2.SPS
    IF N1.SPS is empty
    THEN
```

```
      P = P + N2.probability
    ELSE
      Run DFP for N2.SPS
      N2.tested = The component's id which is tested first by DFA
      IF level of N2.SPS is less than or equal to 2
      THEN
        N2.cost = N2.probability * cost of DFP solution
        N2.probability = N2.probability * working probability of DFP
                         solution
        P = P + N2.probability
      ELSE
        N2.tested = The component's id which is tested first by DFA
        N2.cost = N2.probability * N2.tested
        Insert N2 to TREE at just behind of N1
      ENDIF
      E[C] = E[C] + N2.cost
    ENDIF
  ENDBLOCK
  Delete N0 from TREE
ENDWHILE
PRINT E[C] and P
```

The DF-D algorithm requires too much space and time. In order to implement a more efficient algorithm, we utililize some properties of the problem. We are able to solve the instances having 50-60 components rather than 10-12 components by utilizing these properties. Some of these properties/observations are given below;

- DF-D needs to run DFP and update current SPS for each node of the BDT. We have checked the updated SPSs level for each node and if the depth of SPS is less than or equal to 2 then new node is not created. DFP's solution is accepted as cost of this node, since DFP produces optimal solutions for 1 and 2-level deep SPSs.
- In order to reduce memory requirement, we used depth first search in BDT. We have erased each node after branched on.
- In order to reduce memory requirement and accelerate the algorithm we have calculated the cost cumulatively. When a node is created, global cost and probability variables are updated by using this node's cost and probability.

Example 2: (See [15] for similar examples)

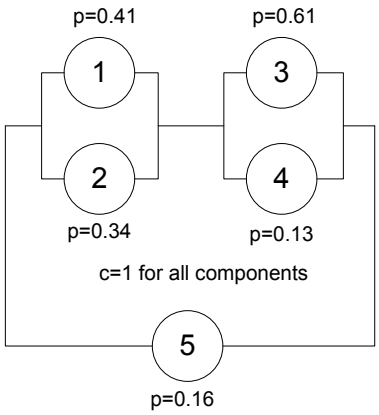


Figure 14. An example 3-level deep parallel system

DFP solution of the SPS shown in Figure 14 is 1-2-3-4-5 and expected cost of this strategy is 3.035. Calculation steps;

Iteration 1: Evaluate 1-level SPSs.

$$c_1/p_1 = 2.44 \ \ and \ \ c_2/p_2 = 2.94 \ \ increasing\ ratio\ order\ 1-2$$
$$c_{12} = 1 + 0.59 = 1.59 \ \ and\ p_{12} = 1 - 0.59 \times 0.66 = 0.6106$$
$$c_3/p_3 = 1.64 \ \ and \ \ c_4/p_4 = 7.69 \ \ increasing\ ratio\ order\ 3-4$$
$$c_{34} = 1 + 0.39 = 1.39 \ \ and\ p_{34} = 1 - 0.39 \times 0.87 = 0.6607$$

Iteration 2: Evaluate 2-level SPSs.

$$c_{12}/q_{12} = 4.083 \ \ and \ \ c_{34}/q_{34} = 4.088 \ \ increasing\ ratio\ order\ 1{,}2-3{,}4$$
$$c_{1234} = 1.59 + 0.6106 \times 1.39 = 2.4387 \ \ and\ p_{1234} = 0.6106 \times 0.6607 = 0.4034$$

Iteration 3: Evaluate 3-level SPSs.

$$c_{1234}/p_{1234} = 6.0453\ and \ \ c_5/p_5 = 6.25\ increasing\ ratio\ order\ 1{,}2{,}3{,}4-5$$
$$c_{12345} = 2.4387 + 0.5966 = 3.0353$$

Figure 15 shows BDT representation of this strategy.

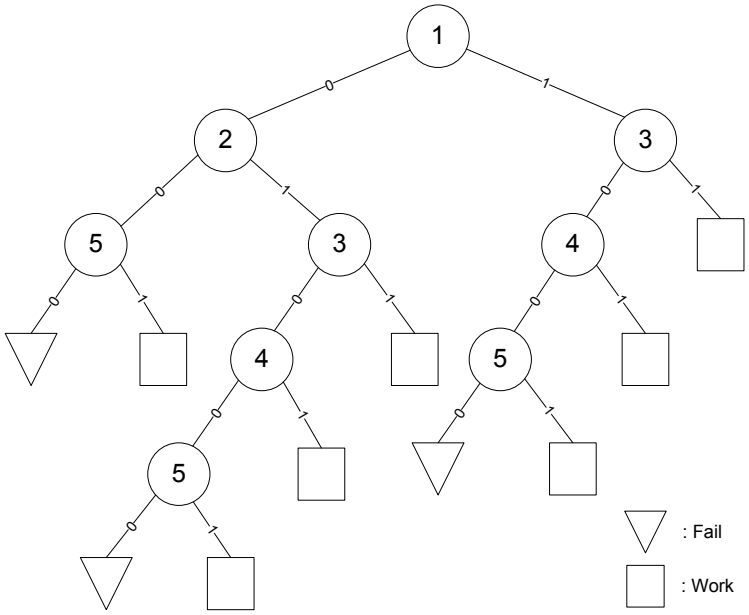


Figure 15. BDT representation of DFP solution for the SPS shown in Figure 14

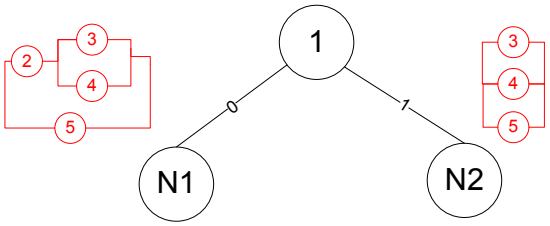
The BDT representation can be used to verify the expected cost of this strategy. The expected cost is as follows.

$$E[C_{DFP}] = c_1 + q_1\left(c_2 + q_2(c_5) + p_2\left(c_3 + q_3(c_4 + q_4 c_5)\right)\right) + p_1\left(c_3 + q_3(c_4 + q_4 c_5)\right)$$
$$= 1 + 0.59\left(1 + 0.66 + 0.34\left(1 + 0.39(1 + 0.87)\right)\right)$$
$$+ 0.41\left(1 + 0.39(1 + 0.87)\right) \quad = 3.0353$$

Let's find DF-D solution for the SPS given in Figure 14.

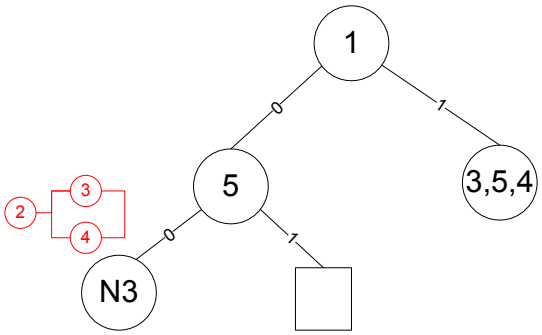
Iteration 1:

- Find DFP solution for the current SPS. DFP solution is found as 1-2-3-4-5 above. Component 1 will be tested.
- Create two nodes for failing and working states of component 1. Update the SPSs of these nodes.



Iteration 2:

- Find DFP solution for the N1's SPS.
  - $c_{34}/q_{34} = 4.088\ and\ c_2/q_2 = 1.515\ ratio\ order\ 2-3,4$
  - $c_{234}/p_{234} = 6.562\ and\ c_5/p_5 = 6.25\ ratio\ order\ 5-2,3,4$
  - Node N1 tests component 5.
- Create two nodes for failing and working states of component 1. Update the SPSs of these nodes.
- Find DFP solution for the N2's SPS. N2 has 1-level deep so do not create a new node and use DFP cost.
  - $c_3/p_3 = 1.64\ , c_4/p_4 = 7.69\ and\ c_5/p_5 = 6.25$
  - $c_{354} = 1.7176$



Iteration 3:

- Find DFP solution for the N3's SPS. N3 has 1-level deep so do not create a new node and use DFP cost.
  - $c_{34}/q_{34} = 4.088\ and\ c_2/q_2 = 1.515\ ratio\ order\ 2-3,4$
  - $c_{234} = 1.4726$

$$E[C_{DF-D}] = c_1 + q_1(c_5 + q_5 c_{234}) + p_1 c_{354}$$
$$= 1 + 0.59(1 + 0.84 \times 1.4726) + 0.41 \times 1.7176 = 3.024$$

Figure 16 shows BDT representation of DF-D strategy. Dashed lines show DFP solution for the SPSs having less than or equal to 2-level deep.

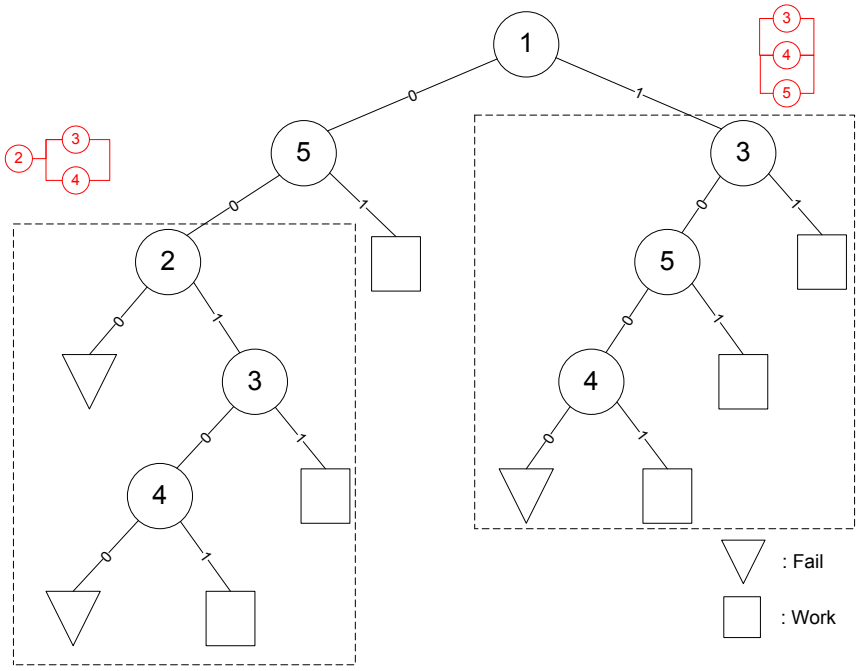


Figure 16. BDT representation of DF-D solution for the SPS shown in Figure 14

Example 2 shows that DF-D can find better solutions than DFP. However DF-D is not necessarily find optimal solution for 3-level deep SPSs. Figure 17 shows a better strategy than DF-D having expected cost 2.993. (See [15])

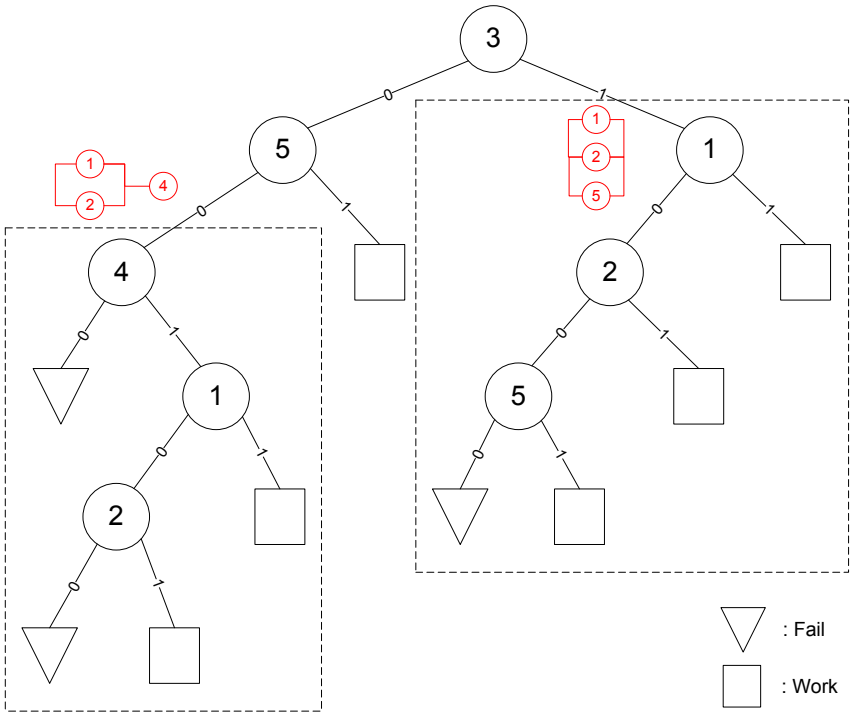


Figure 17. BDT representation of a nonpermutation solution for the SPS shown in Figure 14

4.3. Dynamic Programming Algorithm (DYNPROG)

DYNPROG is a dynamic programming method which is developed by Greiner et al. [15]. This algorithm can solve 3-level or deeper SPSs optimally but time complexity and memory requirement of this algorithm is high.

Theorem 2: In any optimal strategy the components that are in the same 1 level deep sub-system should be in the correct order on any path from root to leaf in the strategy tree, meaning that components that belong to the same parallel sub-system should be in non-decreasing order of c_i/p_i in all paths. (They do not need to be one after another though) [15]

DYNPROG uses Theorem 2 so the time complexity is a function of number of subsystems and number of components. Hence it is a dynamic programming method, it makes enumeration; it cannot be used for large instances in a computational study.

5. IMPROVED SOLUTION METHOD

5.1. Cost of Permutation Strategies

We mentioned about the expected cost calculation methods in Section 2. We need to create BDT to calculate cost of any random permutation or dynamic/nonpermutation strategy. Only exception is depth first permutation strategies; we don't need to create BDT for calculating the cost of these strategies. Creating a BDT for a given SPS having "n" components have a time and space complexity $O(2^n)$.

Cost calculation is a time consuming operation so it is hard to solve the SPSs having more than 10-20 components. Moreover, metaheuristic methods cannot be applied to Sequential Testing problem because they have to search for a solution many times as subroutine. And the expected costs of many solutions need to be evaluated.

In this study we focused 3-level deep parallel SPSs and developed an algorithm for calculating the expected cost faster for permutation strategies. This new method enables to apply metaheuristics for 3-level deep SPSs. It also increases the solvable instance size. When we solve parallel systems, we can find solution for series systems by using duality.

Input: A permutation of the components δ
Output: The expected cost of testing with respect to the permutation.
Algorithm
c_i: cost of testing component i
P_j: working probability of series subsystem j
Q_{jk}: failing probability of subsystem k of series system j
Initially $P_j, Q_{jk} = 1$ for all j and k , TotalCost = 0
Renumber components as i' according to permutation δ.
FOR *i'*=1 to number of components
  Let *j'* is the index of the series system including component *i'*
  J is the set of series systems which ***can give result***$^{(*)}$
  K_j *is the set of parallel subsystems of series system j*

$$P = 1 - \left(\prod_{j \in J',\ j \neq j'} P_j \left(\prod_{k \in K_j} (1 - Q_{jk}) \right) \right)$$

IF *i′* is element of a series system
THEN
  TotalCost = TotalCost + $c_{i'} \times P_{j'} \times P$
  $P_{j'} = P_{j'} \times p_{i'}$
ELSE
  Let *k′* is the index of 1-level parallel system including component *i′*
  TotalCost = TotalCost + $c_{i'} \times P_{j'} \times Q_{j'k'} \times P$
  $Q_{j'k'} = Q_{j'k'} \times q_{k'}$
  IF all tests are realized in system *k′*
  THEN
    $P_{j'} = P_{j'} \times (1 - Q_{j'k'})$
  ENDIF
ENDIF
ENDFOR

(*) A 3-level deep parallel SPS functions if we have a series system that functions. In other words, if all of the single components and individually at least one component of each parallel subsystem of a series system are tested and working then this system is in working state. For example: In Figure 12, if we test component 1 and 3 and they are working, we can declare that the SPS is working state without testing any other component.

Example 3. Let's calculate the cost of permutation strategies 3-1-2-4-5-6 and 1-3-5-4-2-6 for the SPS shown in Figure 12.

Label the first series system ((1∨2) ∧3) as A and parallel subsystem (1∨2) as Aa

Label the second series system (4∧ (5∨6)) as B and parallel subsystem (5∨6) as Ba

Permutation: 3-1-2-4-5-6

Initialization: $C = 0$, $P_A, P_B, Q_{Aa}, Q_{Ba} = 1$

Iteration 1: Test 3

$P = 1$
$C = C + c_3 = 1$
$P_A = p_3$

Iteration 2: Test 1

$P = 1$
$C = C + c_1 P_A = C + c_1 p_3 = 1.25$
$Q_{Aa} = q_1$

Iteration 3: Test 2

$P = 1$
$C = C + c_2 P_A Q_{Aa} = C + c_2 p_3 q_1 = 1.375$
$Q_{Aa} = q_1 q_2$
$P_A = p_3(1 - q_1 q_2)$

Iteration 4: Test 4

$P = q_3 + p_3 q_1 q_2$
$C = C + c_4 P = C + c_4(q_3 + p_3 q_1 q_2) = 2.208$
$P_B = p_4$

Iteration 5: Test 5

$P = q_3 + p_3 q_1 q_2$
$C = C + c_5 P_B = C + c_5 p_4 (q_3 + p_3 q_1 q_2) = 2.375$
$Q_{Ba} = q_5$
$P_B = p_4$

Iteration 6: Test 6

$P = q_3 + p_3 q_1 q_2$
$C = C + c_6 P_B Q_{Ba} = C + c_6 p_4 q_5 (q_3 + p_3 q_1 q_2) = 2.514$

Permutation: 1-3-5-4-2-6

Initialization: $C = 0, P_A, P_B, Q_{Aa}, Q_{Ba} = 1$

Iteration 1: Test 1

$P = 1$
$C = C + c_1 = 1$
$Q_{Aa} = q_1$

Iteration 2: Test 3

$P = 1$
$C = C + c_3 = 2$
$P_A = p_3$

Iteration 3: Test 5

$$P = 1 - p_1 p_3$$
$$C = C + c_5 P = C + c_5(1 - p_1 p_3) = 2.875$$
$$Q_{Ba} = q_5$$

Iteration 4: Test 4

$$P = 1 - p_1 p_3$$
$$C = C + c_4 P = C + c_4(1 - p_1 p_3) = 3.750$$
$$P_B = p_4$$

Iteration 5: Test 2

$$P = 1 - p_4 p_5$$
$$C = C + c_2 P_A Q_{Aa} P = C + c_2 p_3 q_1 (1 - p_4 p_5) = 3.871$$
$$Q_{Aa} = q_1 q_2$$
$$P_A = p_3(1 - q_1 q_2)$$

Iteration 6: Test 6

$$P = q_3 + p_3 q_1 q_2$$
$$C = C + c_6 P_B Q_{Ba} = C + c_6 p_4 q_5 \, (q_3 + p_3 q_1 q_2) = 4.01$$

5.2. SAPATS Algorithm

5.2.1. Simulated Annealing Algorithm

The Simulated Annealing algorithm simulates the heating and cooling process of solids. Annealing is a physical process where a solid heated to high temperature, cools slowly and tends to state with least internal energy. The SA begins with some initial solution and temperature and operates until the temperature reaches critical value. If the cooling process is slow, particles of the solid will be close to each other, and the solid have high resistance. If the cooling process is fast, the solid will be hard but fragile. Because some particles will be close to each other but some particles will not. If we heat a solid and refrigerate too fast than the particles of this solid select the first good position as destination point. They don't have enough chance to search better points. These are local optimal points. If they find enough time to search, they can find better destination points. The simulated annealing heuristic is based on this fact [2][8][14].

There exist different variations of SA in literature but the main procedure is as follows [14];

- Start with an initial solution and an initial temperature
- Find a neighbor of this solution
- If the new solution improves the objective function value then accept this solution
- If the new solution does not improve the objective function value then accept this solution according to a probability (which depends on the current temperature and the difference between current solution and best solution's objective function value.)
- If a solution is accepted then update/decrease the temperature (cooling)
- Repeat this procedure (continue with finding a neighbor of accepted solution) until termination conditions.

The acceptance probability of bad solutions is calculated as follows;

$$p = e^{-\Delta C\ T}$$

ΔC shows the difference between current solution and best solution's objective function value and T shows the current temperature. If the temperature decreases or ΔC increases then acceptance probability decreases. The cooling process provides that the algorithm converges to a local optimum with the passing of iterations.

5.2.2. Tabu Search Algorithm

The tabu search employs restrictions to block certain moves, and aspiration criteria to allow very good solutions to overcome any tabu status. Tabu restrictions are used to prevent moving back to previously analyzed solutions. The aspiration criteria determines when a move produces a solution better than the best known solution it is accepted as new solution even if tabu [7]. This structure is used to prevent cycling and search for good solutions and reach a local optimal.

5.2.3. Improved Algorithm

The SA can find good solutions quickly but it converges to a local (or global) optimum in a short time period so it may not improve the solution in a long time. On

the other hand, the Tabu Search may not find a good solution quickly but it can improve this solution in a long time and can find better solutions than SA. This means that, SA can find better solutions than TS in short time limits [8]. The results which are reached by Hussin and Stützle [14] confirm this situation. They compares the different SA and TS algorithms' performances.

Thanks to the method given in section 3.2 a metaheuristic method can be applied to this problem. In the light of the above comparison we decided to develop a hybrid metaheuristic method to solve sequential testing problem. We want to combine the advantages and reduce the disadvantages of these two algorithms. This hybrid algorithm works faster than TS and it finds better solutions than SA [2][8].

We use the fast cost calculation method presented in section 3.2 and Theorem 2 to develop a simulated annealing with post analysis tabu search (SAPATS) algorithm. A similiar structure is proposed by Misevicius [2] and it is compared with SA, TS and different hybrid SA-TS algorithms. This algorithm performs better than other algorithms both in terms of solving time and solution quality.

The SAPATS algorithm starts with a DFP solution and simulating annealing algorithm finds an initial solution for TS to improve. Since SA provides a good initial solution for TS, diversification is not used. At each step, we use our efficient method described above to compute the expected cost of neighbor permutation strategies.

Basic flow of the SAPATS algorithm is given in Figure 8.

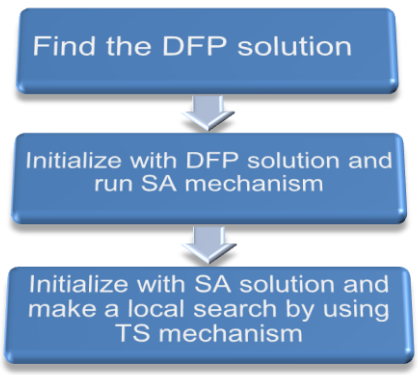


Figure 18. Basic flow of SAPATS algorithm

The pseudo code of SAPATS is as follows.

Algorithm

```
Xnew: an initial solution found by DFP algorithm
Xbest=Xnew, Xcurrent=Xnew
Timelim=Initial_time_limit
T=K*Cost(Xbest)
WHILE time<Timelim
  n=0
  WHILE n<Num_of_accepted AND time<Timelim
    make a random single element move exchange on Xcurrent
    and find an Xnew satisfying Theorem 2
    IF Cost(Xnew)<Cost(Xbest)
    THEN
      Xbest=Xnew
      Xcurrent=Xnew
      increment n
      IF time≥Timelim-Time_limit_step/2
        AND Timelim<Global_time_limit
      THEN
        Timelim=Timelim+Time_limit_step
      ENDIF
    ELSE
      z=exp(-((Cost(Xnew)-Cost(Xbest))/T))
      accept Xnew as Xcurrent with the probability z
    ENDIF
  ENDWHILE
  T=T*α;
ENDWHILE
Timelim=Timelim+ Time_limit_step
Xcurrent=Xbest
WHILE time<Timelim
  decrease all positive tabu_list entries 1 unit
  examine all possible single element move exchanges
  (which satisfies Theorem 2) of the Xcurrent
  save the best Candidate_list_size solutions as ordered in
  candidate_list
  index=0
  REPEAT
    rename candidate_list [index] as Xcandidate
    IF the move creates Xcandidate is not tabu
```

```
      OR cost(Xcandidate)<Cost(Xbest)
    THEN
      Xnew=Xcandidate
    ENDIF
    increment index
  UNTIL updating the Xnew
  make the selected move's tabu_list entry equal to tabu_size
  ΔCost = Cost(Xnew) - Cost(Xbest)
  IF ΔCost<0
  THEN
    Xbest=Xnew
    IF time≥Timelim-Time_lim_step/2
       AND Timelim<Global_time_limit
    THEN
      Timelim=Timelim+Time_limit_step
    ENDIF
  ELSE
    Xcurrent=Xnew
  ENDIF
ENDWHILE
print Xbest and Cost(Xbest)
```

5.2.4. Parameter Selection

In this study, we have used non-deterministic run time strategy for all algorithms and all instances. This means that there exist dynamic "time-limit"s which are determined by the convergence of the solution. A deterministic maximum "time_limit" is also determined in order to prevent too extended runs. This mechanism works in this way;

- A restricted "time_limit" is assigned initially.
- If the algorithm continues improving the solution when the current solving time is close to "time_limit", the time limit is increased.
- If the algorithm converges to a local optimal before "time_limit" then the "time_limit" will not be increased.

This mechanism has two advantages. Firstly, the "time_limit" does not restrict the algorithm too many so it can perform better. Secondly, if the algorithm converges to a local optimum fast, the algorithm does not consume unnecessary time.

We decided on the values of the time parameters as follows.

- Initial_time_limit = num_of_components seconds
- Time_limit_step = num_of_components/5 seconds
- Global_time_limit = 600 seconds
- Candidate_list_size = Tabu_size +1

The initial_time_limit and Time_limit_step are the function of component number. The values of these two parameters do not alter the solution quality because the time_limit is increased as dynamically.

The parameters used in the SAPATS algorithm are as follows:

- **K :** a constant to decide the initial temperature
- **α :** a constant to decide the cooling speed
- **num_of_accepted:** number of accepted solutions in each iteration
- **tabu_size:** size of the short-term tabu list

We have realized some experiments to decide the values of these parameters. The candidate values are given in Table 1. Totally 81 designs are tested on 10 randomly selected instances.

Table 1. Candidate values of algorithm parameters

K	Alpha	Num of Accepted	Tabu Size
0.05	0.95	5	N
0.01	0.9	3	N/2
0.005	0.85	1	N/4

We have solved all instances by using all combinations of the values given in Table 1. We have ordered the objective function values in non-decreasing order for each instance. Best five solutions are scored by using the rating. For example best design earns 5 point, second best design earns 4 points, third best design earns 3 points etc. All other solutions earn 0 point. Each design is scored for all instances and total scores of all design are calculated. Overall scores of best 5 designs are given in Table 2. N is the component number in Table 2.

Table 2. Scores of best designs

Design No	K	Alpha	Num of Accepted	Tabu Size	Score
66	0.005	0.95	3	N/4	23
52	0.05	0.85	1	N/2	21
48	0.005	0.95	1	N/2	21
65	0.01	0.95	3	N/4	21
23	0.01	0.9	1	N	20
2	0.01	0.95	5	N	20

We have decided to use the design 66 because it has the largest score. The selected parameters are as follows:

Parameters

- K= 0.005
- α=0.95
- Num_of_accepted= 3
- Tabu_size = num_of_components/4

6. APPLICATION

6.1. Experimental Design

In our experimental design, we decided to generate 3-level deep SPS instances with certain number of components. This is not a straightforward task. As one forms the subsystems the number of remaining components decrease and the generated SPS could be biased in terms of the sizes of the subsystems. An alternative method would be to fix the number of subsystems and randomly determine the size of the sub-subsystems. If the random instances are generated in this way, there would be a wide range for the total number of components in each instance and it would be difficult to analyze the results with respect to the number of components.

We generated random instances with 10, 20, 30, 40, 50 and 100 components of 3-level parallel systems. We only work with parallel systems because algorithms and results of a sequential testing problem for an SPS can be easily translated for its dual system [6]. We randomly determine the number of subsystems and the number of parallel systems for each subsystem for each value of the number of components. We use different parameters for different values of the number of components. Then we try to assign the corresponding number of components to the parallel systems such that the whole SPS has the required number of components. We have some steps to avoid extreme cases and we also have a mechanism to determine the appropriate parameters for different values of the number of components.

At the end we obtain 200 random instances for each value of number of components so we have 1200 instances in total. 120 of 200 instances are created by Instance generator-1 in three clusters and 80 of instances are created by Instance generator-2 in two clusters. For each instance, we run DFP, DF-D, and SAPATS and compute the expected cost of the strategy produced by these algorithms. We have used two different structures to create random SPSs.

6.1.1. Instance generator-1

This structure generates parallel systems having known number of components (n) and 3-level deep. Here, single components are not allowed as an element of main system; see Figure 9.

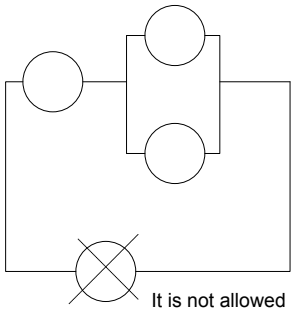


Figure 19. An example SPS having a single component of main system

We group the parameters in three different clusters. First class is named as Strategy 1 creates SPSs having a few number of subsystems. Each subsystem has many components. Strategy 3 creates SPSs having many subsystems but the subsystems have fewer components. Strategy 2 creates SPSs having subsystems more than Strategy 1's and less than Strategy 3.

The above clustering method is used to create different instance groups having different hardness levels.

Parameters:

a: number of level 2 systems

- a can take value in the interval $\left[2, \left\lfloor \frac{n}{2} - \epsilon \right\rfloor \right]$ ϵ is a very small number.

b_i: number of series subsystems in system i (system i is a 2-level system)

- for $b_0 = 0$ and $i = 1, \dots, a$
  b_i can take value in interval $\left[2, \left((n-1) - \sum_{j=0}^{i-1} b_j - 2(a-i)\right)\right]$

a_{min} : it is a parameter which limits the minimum value of a is decided by user.

a_{max} : it is a parameter which limits the maximum value of a is decided by user.

b_{max} : it is a parameter which limits the maximum value of b is decided by user.

Different a_{max} and b_{max} values are decided for creating different shaped level 3 systems. These values are chosen as follows:

Table 3. Instance generator-1 parameters

Component Number	Strategy 1			Strategy2			Strategy 3		
	a_{min}	a_{max}	b_{max}	a_{min}	a_{max}	b_{max}	a_{min}	a_{max}	b_{max}
10	2	3	4	3	3	5	3	4	Not limited
20	2	4	6	4	7	7	7	9	
30	2	5	8	5	10	10	10	14	
40	2	6	9	6	14	12	14	19	
50	2	7	10	7	18	14	18	24	
100	2	12	16	12	35	22	35	49	

Algorithm

get input parameters n, a_{min} , a_{max} and b_{max} , p_{min} , p_{max} , c_{min} , c_{max}, from user

generate a random "a" in interval $[a_{min}$, $a_{max}]$

create a two-dimensional "Array" having "a" rows

$b_0=0$; $b_{sum}=0$; $i=0$;

WHILE $i<$ size of *Array*

generate a random "b_i" in interval $\left[2,\left(\min\left(b_{max}\,,(n-1-b_{sum}-2a+2i)\right)\right)\right]$

create "b_i" columns in row i of "*Array*" and write 1 in every cells.

$b_{sum} = b_{sum} + bi$;

ENDWHILE

calculate the number of remaining components "*remaining*"=$(n - b_{sum})$

give id to all systems from 1 to b_{sum}

REPEAT

select a system randomly and add one component to this system.

UNTIL all components are assigned

REPEAT

give id for component

assign p in interval [p_{min} , p_{max}]

assign c in interval [c_{min} , c_{max}]

UNTIL "n" components are finished

Example: A sample output for the above algorithm when component number is 10 can be such that:

Array=(1,2,2)(3,2) and this array represents the system given in Figure 10.

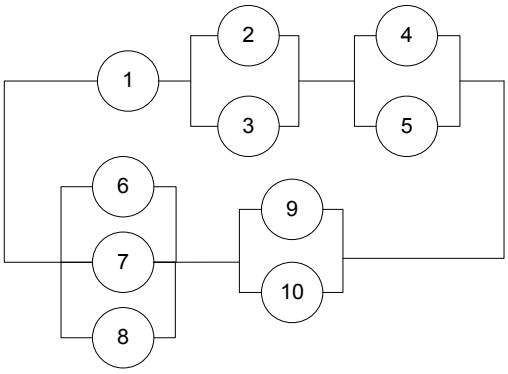


Figure 20. An example randomly generated SPS

6.1.2. Instance generator-2

This structure allows single components as an element of main system. It means that it creates 3-level SPSs which may consist of some 2-level systems and components. For example the SPS given in Figure 9 can be created by this generator. It generally creates SPSs having more than one single components as element of main system. Hence, it generally creates SPSs which are easier than generator-1's instances to be solved by DFP and DF-D. We created instances in two clusters; the number of single components in main system is reduced in second cluster. The pseudo code for Instance generator-2 is presented associate for two clusters as follows:

```
Algorithm
get input parameters n, a_min , a_max  and b_max , p_min , p_max , c_min , c_max,
REPEAT
  give id for component
  assign p in interval [p_min , p_max]
  assign c in interval [c_min , c_max]
UNTIL "n" components are finished
add the id of all components to "candidate_list"
REPEAT
  create a new system as "created_system"
  assign a system id for "created_system"
  select status of "created_system" ramdomly (series or parallel)
  select two elements from "candidate_list", add to "created_system"
```

```
  delete these two elements from "candidate_list"
  calculate the level of "created_system" by using elements and status
  add the "created_system" to "systems_list"
  add the id of "created_system" to "candidate_list"
UNTIL  size of  "candidate_list"=0 OR level of "created_system"=3
IF size of "candidate_list">0
THEN
  REPEAT
    select a random element "a" from "candidate_list"
    IF " a" is component
    THEN
      select a system "b" from "systems_list" randomly (for first
        cluster)
      select a system "b" randomly from "systems_list" which
        excludes main system (for second cluster)
      add "a" to elements of "b"
      delete "a" from "candidate_list"
    ELSE
      select a system "b" from "systems_list"
      IF "a" and "b" have same status and level
         OR "b" has higher level than "a"
      THEN
        add "a" to elements of "b"
        delete "a" from "candidate_list"
      ENDIF
    ENDIF
  UNTIL size of "candidate_list"=0
ENDIF
REPEAT
  IF a system and an element of this system have same status and level
  THEN
    merge these two systems
    revise the "systems_list"
  ENDIF
UNTIL all systems in "systems_list" are checked
```

6.2. Results

We have implemented the algorithms in C++ and solve the instances which are generated by Instance generator-1 and Instance generator-2. We analyze the improvement in the expected cost with respect to DFP since the solution obtained by DFP is used as an initial solution for SAPATS and we know that DF-D can only be better than DFP. We analyze the improvements by the number of components and by some properties of the random instances.

Tables 2 and 3 compare SAPATS and DF-D with DFP with respect to the number of components. Max % imp column shows the maximum % improvement with respect to DFP solution whereas Mean % imp column shows the average % improvement over 200 instances with the same number of components. We observe that the improvements are largest for moderate size problem instances. DF-D could be run for 8 instances for 100 components. For others, DF–D seems to improve the DFP solution better than SAPATS. Yet, DF-D does not provide a permutation strategy.

Table 4. SAPATS results based on component numbers

SAPATS				
No of Comps	Max % imp.	Mean % imp.	Number of Improved	Number of Solved
10	3,4%	1,0%	14	200
20	2,0%	0,5%	8	200
30	25,3%	3,0%	9	200
40	12,4%	1,6%	13	200
50	1,1%	0,4%	9	200
100	6,8%	0,5%	20	200

Table 5. DF-D results based on component numbers

DF-D				
No of Comps	Max % imp.	Mean % imp.	Number of Improved	Number of Solved
10	5,1%	1,1%	18	200
20	6,8%	0,7%	27	200
30	12,2%	1,1%	22	200
40	12,4%	1,5%	24	200
50	4,9%	0,7%	24	200
100	0,4%	0,1%	5	80

Tables 5 and 6 provide the same information for different classes of instances that we refer as scenarios. Here scenarios correspond to some properties of the instances. Scenario 1 consists of instances where single components are allowed in subsystems, scenario 2 consists of instances where single components are allowed in subsystems but their number is low. The scenarios 3,4 and 5 correspond to instances with no single component as a subsystem and the number of subsystems is low, medium and high respectively. Scenarios are summarized in Table 4.

Table 6. Scenario Summary

Scenario	Properties	Generated By
1	Single components as subsystem	Instance generator-2
2	Single components as subsystem but number of them is reduced	Instance generator-2
3	No single components as subsystem and number of subsystems is low	Instance generator-1
4	No single components as subsystem and number of subsystems is medium	Instance generator-1
5	No single components as subsystem and number of subsystems is high	Instance generator-1

The improvements seem robust among different groups here and as before DF-D seems to improve the DFP solutions better than SAPATS. When we examine the results in detail, it is not easy to observe what conditions favor each algorithm.

Table 7. SAPATS results based on scenarios

SAPATS				
Scenario	Max % imp.	Mean % imp.	Number of improved	Number of solved
1	3,4%	0,8%	17	240
2	25,8%	1,2%	25	240
3	0,1%	0,0%	3	240
4	12,4%	3,5%	5	240
5	6,8%	0,8%	23	240

Table 8. DF-D results based on scenarios

DF-D				
Scenario	Max % imp.	Mean % imp	Number of improved	Number of solved
1	4,9%	0,4%	23	240
2	12,2%	1,0%	23	240
3	8,7%	1,1%	27	200
4	12,4%	2,0%	15	200
5	3,0%	0,6%	32	200

7. CONCLUSION AND FUTURE RESEARCH

DFP algorithm was proposed in the literature for sequential testing of SPSs in different studies. There are also articles that show that there exist instances where DFP performs arbitrarily badly. In this study, we conducted a numerical study to compare the performance of DFP with algorithms that we develop to obtain better solutions than provided by DFP. Although it is possible to improve to the solution of DFP by up to 25% on some instances, on average the % improvements were not that large.

DFP reaches the same solutions with other algorithms for nearly 90% percent of all instances. We also observed that permutation strategies (such as one that is produced by DFP or SAPATS) that are very easy to represent and implement perform very satisfactorily.

Finding a new solution and calculating cost in each iteration is polynomial time operations for SAPATS so the solvable instance size is high. On the other hand DF-D cannot solve big instances especially the instances created by Instance generator-1. The solution quality of SAPATS algorithm is not affected negative from instance size. Moreover, solution quality of SAPATS increases when the number of subsystems increased.

The hardness of the sequential testing problem of SPSs and in particular 3-level SPSs are open problems. One question is whether there is an efficient algorithm for computing the optimal permutation strategy for 3-level SPS. A second question is whether there is an efficient algorithm for computing the optimal strategy of 3-level SPS. Another direction of research would be to develop and analyze different heuristic approaches for more general SPSs rather than 3-level SPSs. It is also an open problem to find optimal solution for 4-level SPSs having identical components.

Precedence constraints can be incorporated as in [3]. It is known that the testing problem is NP-complete when we have precedence constraints even for 1-level deep SPSs. Approximation algorithms can be developed for special cases as in [9] and [1]. Literature has solved 1-level SPSs optimally but 2-level SPSs under even line-precedence is also waits to be solved.

BIBLIOGRAPHY

[1] **A. Deshpande, L. Hellerstein and D. Kletenik.** Approximation algorithms for stochastic boolean function evaluation and stochastic submodular set cover. *ArXiv.* 2013, 1303.0726.

[2] **A. Misevicius.** An improved hybrid optimization algortihm for the quadratic assignment problem. *Mathematical Modelling and Analysis.* 2004, Vol. 9, pp. 149-168.

[3] **B. Çatay, Ö. Özlük and T. Ünlüyurt.** TestAnt: an ant colony system approach to sequential testing under precedence constraints. *Expert Systems with Applications.* November 2011, Vol. 38, 12, pp. 14945-14951.

[4] **B. De Reyck and R. Leus.** R&D-project scheduling when activities may fail. *IIE Transactions.* 2008, Vol. 40, 4, pp. 367-384.

[5] **D. Berend, R. Brafman, S. Cohen, S.E. Shimony and S. Zucker.** Optimal ordering of independent tests with precedence constraints. *Discrete Applied Mathematics.* 2013, 162, pp. 115–127.

[6] **E. Boros and T. Ünlüyurt.** Sequential testing of series-parallel systems of small depth. *Computing Tools for Modeling, Optimization and Simulation.* Springer US, 2000, pp. 39–73.

[7] **E. Rolland, D. A. Schilling and J. R. Current.** An efficient tabu search procedure for the p-Median Problem. *European Journal of Operational Research.* 1996, 96, pp. 329-342.

[8] **G. Paul.** Comparative performance of tabu search and simulated annealing heuristics for the quadratic assignment problem. *Operations Research Letters.* 2010, Vol. 38, 6, pp. 577-581.

[9] **H. Kaplan, E. Kushilevitz and Y. Mansour.** Learning with attribute costs. *In STOC.* 2005, pp. 356-365.

[10] **K. S. Natarajan.** Optimizing depth-first search of AND-OR trees. Yorktown Heights, NY : *IBM T.J.Watson Research Center*, 1986. Technical Report. 10598.

[11] **L. A. Cox Jr., Y. Qui and W. Kuehner.** Heuristic least-cost computation of discrete classsification functions with uncertain argument values. *Annals of Operations Research.* 1989, Vol. 21, pp. 1-30.

[12] **M. N. Azaiez and V. M. Bier.** Optimal resource alocation for security in reliability systems. *European Journal of Operational Research.* 2007, pp. 773-786.

[13] **M. R. Garey.** Optimal task sequencing with precedence constraints. *Discrete Mathematics.* 1973, 4, pp. 37-56.

[14] **M. S. Hussin and T. Stützle.** Tabu search vs. simulated annealing for solving large quadratic assignment instances. s.l. : *IRIDIA – Technical Report Series*, 2010. TR/IRIDIA/2010-020.

[15] **R. Greiner, R. Hayward, M. Jankowska and M. Molloy.** Finding optimal satisficing strategies for and-or trees. *Artificial Intelligence.* 2006, pp. 19-58.

[16] **S. Allen, L. Hellerstein, D. Kletenik and T. Ünlüyurt.** Evaluation of DNF formulas. *http://arxiv.org/abs/1310.3673.* [Online] 2013.

[17] **S. Y. Chiu, L.A. Cox Jr. and X. Sun.** Optimal sequential inspections of reliability systems subject to parallel chain precedence constraints. *Discrete Applied Mathematics.* 1999, pp. 327-336.

[18] **T. Ünlüyurt.** Sequential testing of complex systems: a review. *Applied Mathematics.* 2004, pp. 189-205.

[19] **T. Ünlüyurt and E. Boros.** A note on optimal resource allocation for security in reliability systems. *European Journal of Operational Research.* 2009, pp. 601-603.

[20] **W. Wei, K. Coolen and R. Leus.** Sequential testing policies for complex systems under precedence constraints. *Expert Systems with Applications.* 2013, 40, pp. 611-620.

[21] **Y. Ben-Dov.** Optimal testing procedures for special structures of coherent systems. *Management Science.* 1981, 27(12), pp. 1410-1420.