SEQUENTIAL TESTING OF A SERIES SYSTEM IN BATCHES

REBİ DALDAL

Submitted to the Graduate School of Engineering and Natural Sciences

in partial fulfillment of the requirements for the degree of

Master of Science

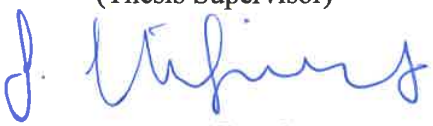
Sabancı University

December, 2015

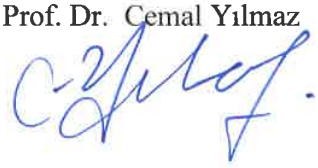
SEQUENTIAL TESTING OF A SERIES SYSTEM IN BATCHES

APPROVED BY:

Assoc. Prof. Dr. Tonguç Ünlüyurt
(Thesis Supervisor)



Asst. Prof. Dr. Cemal Yılmaz



Assoc. Prof. Dr. Oya Ekin Karaşan

DATE OF APPROVAL: 23/12/2015

SEQUENTIAL TESTING OF A SERIES SYSTEM IN BATCHES

Rebi Daldal

Master of Science in Industrial Engineering

Thesis Supervisor: Tonguç Ünlüyurt

Abstract

In this thesis, we study a new extension of the Sequential Testing problem with a modified cost structure that allows performing of some tests in batches. As in the Sequential Testing problem, we assume a certain dependence between the test results and the conclusion. Namely, we stop testing once a positive result is obtained or all tests are negative. Our extension, motivated by health care applications, considers fixed cost associated with executing a batch of tests, with the general notion that the more tests are performed in batches, the smaller the contribution of the total fixed cost of the sequential testing process. The goal is to minimize the expected cost of testing by finding the optimal choice and sequence of the batches available. We separately study two different cases for this problem; one where only some subsets of all tests can be performed together and one with no restrictions over tests. We analyze the problem, develop solution algorithms and evaluate the performance of the algorithms on random problem instances for both both cases of the problem.

Keywords: Combinatorial Optimization, Heuristics, Sequential Testing, Function Evaluation, Batch Testing

SERİ SİSTEMİN GRUPLAR HALİNDE SIRALI SINANMASI

Rebi Daldal

Endüstri Mühendisliği Yüksek Lisansı

Tez Danışmanı: Tonguç Ünlüyurt

Özet

Bu tezde sıralı sınama probleminin değiştirilmiş maliyet yapısına sahip yeni bir uzantısını inceliyoruz. Sıralı sınama probleminde olduğu gibi test sonuçları ve karar arasında belirli bir bağlantı olduğunu kabul varsayıyoruz. Yani, olumlu bir sonuç elde edince veya tüm testler olumsuz sonuç verirse sınamayı durduruyoruz. Bizim incelediğimiz uzantı, sağlık alanındaki uygulamalardan motive olarak, testlerin gruplar halinde yapılması ile ilgili bir sabit maliyetin olduğunu kabul eder. Buradaki genel kanı daha fazla test grup halinde yapılınca test süreci içerisindeki toplam sabit maliyetin azalacağıdır. Amaç birlikte yapılacak test gruplarını ve bunların sıralarını bularak beklenen maliyeti enaza indirmektir. Biz bu tezde tüm testlerin birlikte yapılabildiği ve sadece bazı test kümelerinin birlikte yapılabildiği iki durumun üzerinde ayrı ayrı çalışıyoruz. Biz her iki durum için problemi analiz ediyor, çözüm algoritmaları geliştiriyor ve algoritmaların perfomansını rasgele yaratılmış örnekler üzerinde inceliyoruz.

Anahtar Kelimeler: Kombinatoryal Eniyileme, Rassal Algoritmalar, Sıralı Sınama, Fonksiyon Değerlendirme, Grup Sınama

for xxx

Acknowledgments

I would like to express my sincere gratitude to my thesis supervisor Tonguç Ünlüyurt for his help and encouragement during the course of my master's thesis.

I want to sincerely thank Danny Segev, Iftah Gamzu, Özgür Özlük, Barış Selçuk and Zahed Shahmoradi for their valuable contributions to this research.

I want to thank Cemal Yılmaz and Oya Ekin Karaşan for accepting to be part of the thesis jury.

I gratefully acknowledge the funding received from TÜBİTAK to complete my masters degree.

I also would like to express many thanks to Sabancı University for the scholarships received and for becoming my home for the last six years.

Contents

List of Figures

List of Tables

1 Introduction

Identifying the state of a system with minimum expected cost has been studied in the literature for various applications under different assumptions, as the Sequential Testing problem. In many cases, the problem is to conduct costly tests one by one until the correct state of the system is found. In this thesis, we assume that certain tests can be performed in batches and study this extension of the testing problem where it is possible to perform multiple tests simultaneously in order to gain a cost advantage through reduced total fixed costs.

The Sequential Testing problem is fairly common in health monitoring/diagnosis situations. The costs for medical tests constitute a good portion of health care expenditure, therefore it is important to develop strategies that prescribe how to execute these medical tests in a cost effective manner. Let us consider a set of tests for a specific medical condition. We assume that if at least one of the tests in the set is positive, the patient is likely to be sick and he/she may require an operation, some medication or further tests that might be invasive. If all the tests are negative, we conclude that the patient is fine. In this setting, testing stops as soon as we get a positive result or all the tests are done with negative results. The goal here is to reach a conclusion with the minimum expected cost, assuming that we have probabilistic information as to the results of the tests (these probabilities could be obtained through statistical analysis, since these tests have been administered many times in the past). Any implementable strategy is simply a permutation of the tests. It is a customary assumption that the states of the individual components are independent of each other. When the tests are independent of each other, it is easy to find the permutation corresponding to the minimum cost strategy [1].

When medical tests are executed, it is common that a number of tests are batched and administered together, in order to save time and money. In order to model this situation, we assume that the total cost of each test consists of a fixed and a variable component. When a group of tests are administered together as a batch, the variable cost portion for the batch directly depends on the individual variable costs of the tests in the group, as it is the summation of attribute costs of individual tests. On the other hand, for each batch, the fixed cost which corresponds to the set up costs is incurred only once, regardless of the number or type of individual tests involved. The set up costs include ordering costs, transportation costs, administration costs etc. So typically, batching more tests would result in reduced total costs. In this framework, the main decision is to find out how the tests should be batched together and optimally sequenced. In other words, we would like to find a partition of the individual tests and a sequencing of this partitioning that would give us the minimum expected total cost. Let us note that if the optimal partition is known, the problem becomes trivial and the optimal sequence of the elements of the partition can easily be found as described in [1].

Although we motivate the problem through medical monitoring, the same framework can be utilized in any other application, where one needs to determine whether a complex system is in a good (e.g. healthy, working etc) state or in a bad (e.g. sick, failing) state and batching of at least a portion of the tests is viable. As in other applications of Sequential Testing, this model can also be extended to include situations where the system can be in more than two states. Another possible motivation for this model could be checking whether a query (AND function) over a database is correct or not when the arguments of the query are stored at different locations and it takes a certain time to get the values of the arguments. In this case, the goal is to answer the query with the minimum expected time and one can ask for multiple arguments for some time advantage.

As mentioned before, the special case of the problem where we have only singletons is easy to solve and has been studied in different contexts in the literature. A review on the problem can be found in [2]. The applications mentioned in the literature range from inspection in a manufacturing facility [3] to sequencing potential projects [4] to ordering the offers

to a number of candidates [5]. A generalization of this version where there are precedence constraints among the tests have been studied in [6] where a polynomial time algorithm is provided when the precedence graph is a forest consisting of rooted in-trees and out-trees. Heuristic algorithms for the same problem are proposed in [7, 8] for general precedence graphs. It is shown that the problem is hard under general precedence constraints in [9, 4]. Furthermore, an approximation algorithm is provided for the testing of a series system when tests are dependent is provided in [10].

Sequential testing problem has also been studied in the literature for more complicated systems. That means that the underlying structure function is more general. In this case a feasible solution can be described by a binary decision tree rather than a permutation. For instance, a polynomial time algorithm is provided in [11] for double regular systems generalizing the optimal polynomial time algorithms for k-out-of-n systems provided in [12]. Note that an AND function is an n-out-of-n function. Series-Parallel systems have been studied in [13, 14] and a polynomial time algorithm is provided for 2-level Series Parallel systems and 3-level Series-Parallel systems with identical components. The structure function of the Series-Parallel system is known as the Read-Once formulae. A 3-approximation algorithm for threshold systems is provided in [15]. Threshold systems are also studied in [16]. Evaluation of certain DNF formulas are considered in [17] and approximation algorithms are proposed for certain classes of DNFs. Other discrete functions are considered in [18]. A series system is a special case of double regular systems or k-out-of-n systems or Series-Parallel systems. We also observe that a Series System is a building block for all these systems. In addition, when we look into the details of the algorithms proposed in these studies, one can say that the algorithms are complex adaptations of the optimal solution for a simple Series System. In all of these studies the tests are performed one by one and it is not allowed to batch tests together.To the best of our knowledge, allowing tests in batches has not been considered in the literature before. (except our other works [19, 20])

In this thesis, we study an extension of this problem where multiple tests can be conducted together for a cost advantage. In many practical applications, this is indeed the case. For

instance, if we are considering a medical diagnosis setting and the tests are conducted by laboratories, it is typical that multiple tests are administered simultaneously and depending on the results of these tests other tests may be required if necessary. We assume that the total cost of conducting multiple tests is the sum of a fixed cost and the costs of the individual tests. The fixed cost portion corresponds to administration and order handling costs.

In the first part of this work, we also assume that we have a priori knowledge of which groups of tests are allowed to be executed together. In other words, not every subset of the tests to be performed can be batched and we are provided with the subsets of the tests that can be executed together. However we also assume that if a set of tests can be performed together, as a natural extension of this, any subset of this set can also be performed together. In the medical setting, the subsets may be considered to correspond to collections of tests that can be executed by a single lab or it can be the case that these tests require the same type of setup. The two extreme cases of this batching policy would be: executing each test on its own or executing all tests simultaneously (if allowed). In the former case, the fixed cost is incurred every time a single test is executed, and in the latter, the fixed cost is incurred only once during the whole testing process.

In the second part of this work we assume there are no restrictions on groups of tests that can be performed together. In terms of the problem given in first part, this corresponds to inclusion of the set consisting of all tests in the list of tests that can be performed together. In real life examples, the fixed cost portion corresponds to administration and order handling costs.

Our contributions in this thesis can be summarized into four main points

- We introduce a new model for sequential testing that allows batching of tests.
- We investigate the complexity of this model and provide properties about characteristics of solutions.
- We propose heuristic algorithms and conduct experimental study on randomly generated instances.

- For the case when all subsets are available we implement a constant factor approximation algorithm and an approximate integer programming formulation and evaluate their performance.

In following sections, firstly we formally define our new testing model where multiple tests can be executed at once with the additional restriction where only some group of tests can be performed together. Afterwards we determine the complexity of the model and analyze the model properties. Finally we propose heuristic algorithms for the problem and compare their performances on randomly generated examples. In next chapter we consider the case where all tests can be performed together and remove the restriction on groups of tests that can be performed together from our model. Then we analyze model properties and define a further generalization of this model where at most k batches of tests can be performed. We determine the complexity of this further model. On next sections we propose our heuristic algorithms and provide an implementation for the constant approximation algorithm proposed in [20] and $\epsilon-$approximate integer program formulation. Furthermore we compare the performance of our heuristics and approximation algorithms on randomly generated problem instances. In the final chapter, we summarize our findings and remark the future research areas.

2 Sequential Testing in Batches When Some Subsets of Tests Are Available

2.1 Problem Definition

Let $N = \{1, 2, \ldots, n\}$ denote the set of tests. When we execute a test, we either get a result of 0 or 1 and we assume that the tests are perfect in the sense that the results that we obtain from the tests are always correct. In our motivating example in Section 1, a result of 0 (1) from a test means a positive(negative) test result. Let $P = (p_1, p_2, \ldots, p_n)$ be the vector whose i^{th} component denotes the probability that the result of test i is 1 where $p_i + q_i = 1$ (where $0 < p_i < 1$) and $C = (c_1, c_2, \ldots, c_n)$ be the vector whose i^{th} component denotes the variable cost of executing test i.

We are given a set Γ of subsets of N describing the tests that can be performed simultaneously where the cardinality of Γ is t. We will assume that the elements of Γ are maximal and the set of tests that can be executed simultaneously is closed under taking subsets. In other words, for each element X of Γ, the tests in any subset of X can also be performed together. We will refer to any set of tests that can be executed simultaneously as a meta-test and we will refer to the set of all meta-tests as Ω, where the cardinality of Ω is m. In other words, Ω consists of all subsets of the elements of Γ. In addition, we assume that each $x \in N$ is an element of at least one of maximal meta-test $X \in \Gamma$ for the problem to be feasible.

When we conduct a meta-test, that means we execute all tests that are elements of that

meta-test and learn the results of these tests. We assume that the results of the tests are independent of each other. The cost of a meta-test M is defined by:

$$C(M) = \beta + \sum_{i \in M} c_i$$

That is, in order to execute any meta-test there is a fixed cost of β and the sum of the variable costs of the tests that are in the meta-tests. Since we assume that the tests are independent of each other, we can write the probability that we will get a negative result from meta-test $M \in \Omega$, as:

$$P(M) = \prod_{j \in M} p_j$$

We will define $Q(M) = 1 - P(M)$ as the probability of obtaining at least one positive result when meta-test M is executed. (We will alternatively refer to the cost and probability of meta-test k as C_k and P_k.)

Let us define the ratio of a meta-test M, $R(M)$, in the following way:

$$R(M) = \frac{C(M)}{Q(M)}$$

If any test result is 0, we conclude that the system is in state 0. Otherwise, if all test results are 1 then we conclude that the system is in state 1. This corresponds to the well-studied Series System (see, e.g., [1]). Essentially, the goal is to evaluate an AND function with the minimum expected cost. For our problem, where we have meta-tests, we do not allow the repetition of any test in a feasible solution. Then a feasible solution to the problem corresponds to a partition of N, the set of all tests, by subsets in Ω. In other words, a feasible solution is a collection $F \subseteq \Omega$ such that for any $X, Y \in F$, we have $X \cap Y = \emptyset$ and $\bigcup_{X \in F} X = N$. The expected cost of a feasible partition depends on the order in which the meta-tests are executed. If there are h meta-tests in a feasible partition is executed in order $\pi(1), \pi(2), \ldots, \pi(h)$, the expected cost of this ordering of the feasible partition can be written as:

$$EC = \sum_{k=1}^{h} \prod_{j=1}^{k-1} P_{\pi(j)} C_{\pi(k)}$$

Where the product over the empty set is defined as 1. This is because the cost of meta-test in order k is incurred if all tests in meta-tests $1, 2, \ldots, k-1$ give a negative result.

The case when only singletons are available has been studied in the literature in different contexts (see, e.g., [1]). In this case, an optimal permutation is the non-decreasing order of ratios. By the same argument, we obtain the following.

Proposition 2.1. *Given a feasible partition of N, the optimal order of meta-tests in the partition is a non-decreasing order of $R(X)$.*

Proposition 2.2. *For any feasible partition $\{M_{\pi(1)}, M_{\pi(2)}, \ldots, M_{\pi(k)}\}$ of N where*

$$R(M_{\pi(1)}) \leq R(M_{\pi(2)}) \leq \ldots \leq R(M_{\pi(k)})$$

the minimum expected cost is given by

$$\sum_{i=1}^{k} \prod_{j=1}^{i-1} P(M_{\pi(j)}) C(M_{\pi(i)}).$$

So it is easy to compute the expected cost of any given partition in $O(n \log n)$ time. Our goal in this problem is to find the partition with the minimum expected cost.

We can also define our problem as follows. We can represent each meta-test as a binary vector of size n. Let us define $X_k = (X_{1,k}, X_{2,k}, \ldots, X_{n,k})$, to represent the binary vector for the meta-test k, for $k = 1, 2, \ldots, m$. Here $X_{i,k} = 1$ if test i is in meta-test k, and $X_{i,k} = 0$ otherwise. If $\mathbf{Y} \in \{0, 1\}^{(m+n)}$ be an indicator vector of a feasible solution, meaning $Y_k = 1$ iff meta-test k

is in the solution. Let $Cost(\mathbf{Y})$ be the expected cost corresponding to a feasible $\mathbf{Y}$. So one can compute $Cost(\mathbf{Y})$ in the following way. For all meta-tests for which $Y_i = 1$ we compute the C/Q ratio. We order them in non-decreasing order of this ratio. Without loss of generality, assume that this order is $\pi(1), \pi(2), \ldots, \pi(l)$. So we assume that l meta-tests are chosen among k available meta-tests to form a partition of all individual tests. Following this we get:

$$Cost(\mathbf{Y}) = \sum_{i=1}^{l} \prod_{j=1}^{i-1} P_{\pi(j)} C_{\pi(i)}$$

. Then the optimization problem can be stated as:

$$\begin{aligned} \text{(STB) Minimize} \quad & Cost(\mathbf{Y}) && (1) \\ \text{subject to} \quad & \sum_{k=1}^{m} X_{i,k} Y_k = 1 \text{ for all } i && (2) \\ & Y_k \in \{0, 1\} \text{ for all } k \in \{1, 2, \ldots, m\} && (3) \end{aligned}$$

Let us consider the following example with 5 tests where $C = (1, 2, 3, 4, 5)$ and $P = (0.8, 0.7, 0.6, 0.5, 0.4)$ with $\beta = 2$. Let us assume that $\Gamma = \{\{1, 2, 3\}, \{2, 4\}, \{5\}\}$ Then $\Omega = \{\{1, 2, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1\}, \{2\}, \{3\}, \{2, 4\}, \{4\}, \{5\}\}$. The binary vector representation of all meta-tests are: $X_1 = (1, 1, 1, 0, 0)$, $X_2 = (1, 1, 0, 0, 0)$, $X_3 = (1, 0, 1, 0, 0)$, $X_4 = (0, 1, 1, 0, 0)$, $X_5 = (1, 0, 0, 0, 0)$, $X_6 = (0, 1, 0, 0, 0)$, $X_7 = (0, 0, 1, 0, 0)$, $X_8 = (0, 1, 0, 1, 0)$, $X_9 = (0, 0, 0, 1, 0)$ and $X_{10} = (0, 0, 0, 0, 1)$. For instance, meta-test 3 corresponds to having tests 1 and 3 together. The cost of meta-test 3 is 2+1+3=6 and the probability that it will give a negative result is the product of the probabilities that tests 1 and test 3 will give negative results, which is $0.8 \times 0.6 = 0.48$. A subset of meta-tests is feasible for the testing problem if each individual test appears exactly once in the subsets. For instance the set $\{X_3, X_8, X_{10}\}$ is feasible. Once we have a feasible subset the optimal sequence for this subset can be found by sorting the meta-tests in the subset in non-decreasing order of their $C_k/(1 - P_k)$ ratios. In this particular example we will do the meta-tests in the following order (X_3, X_8, X_{10}) giving us an expected testing cost of $C_3 + P_3 C_8 + P_3 P_8 C_{10} = 11.016$.

2.2 Analysis of the Problem

2.2.1 Complexity of the Problem

The problem that we consider seems similar to the well known set partitioning problem which is known to be NP-complete. The set partitioning problem asks whether there exists a partition of the ground set by using the subsets. Yet, our problem is always feasible since all singletons are among the given subsets. On the other hand, it is similar to minimum cost set partitioning problem which asks for a minimum cost set partitioning and appears in applications such as airline crew scheduling, vehicle routing etc. In these models, the set of subsets is not given explicitly but implicitly described. On the other hand, the objective function is simpler than our case, since it is just the sum of the costs of the subsets. It is possible to prove the hardness of our problem by using a reduction from a different problem as follows.

Theorem 2.1. *Problem STB is NP-hard.*

Proof. Let us consider a special case of the problem where the tests are identical in the sense that $c_i = c$ and $p_i = p$ for all tests $i \in N$. In addition, let us assume that all the meta-tests contain exactly 3 tests. So the set of meta-tests consists of singletons, doubles and triples. Let us consider the best solution that contains exactly k meta-tests and denote its objective function value as EC_k^*. We can construct a new feasible solution from the solution by adding the tests in the k^{th} meta-test to the first meta-test. The new solution will consist of $k-1$ meta-tests. If we denote the expected cost of this new solution by EC_{k-1}, we have the following:

$$\begin{aligned} EC_{k-1}^* &\leq EC_{k-1} && (4)\\ &\leq EC_k^* + nC - p^{n-1}\beta && (5) \end{aligned}$$

So if $\beta > \frac{nc}{p^{n-1}}$, we conclude that a solution that consists of the minimum number of meta-tests is an optimal solution. In order to find the solution with the minimum number of meta-tests,

we need to find a solution that maximizes the number of triples. Finding a solution with the maximum the number of triples is exactly 3-dimensional matching problem which is NP-hard [21]. Consequently, Problem STB is NP-hard. □

2.2.2 Properties

Obviously, one expects that as the fixed cost β increases, we would tend to include meta-tests with more tests in the good solutions. Yet, since we are working with a given set of meta-tests and require that we have an exact partition, we may not observe this property for each problem instance. On the other hand, we can show the following under the condition that the set of all tests is among the meta-tests.

Proposition 2.3. *If the set of all tests is among the meta-tests, then for each fixed* $(p_1, p_2, \ldots, p_n)$ *and* $(c_1, c_2, \ldots, c_n)$*, there exists a sufficiently large* β *such that the optimal solution only consists of the meta-test consisting of all tests.*

Proof. This particular solution consists of a single meta-test and it is the only solution that consists of a single meta-test. In order for this solution to be optimal, it should be better than all solutions that contain $h \in \{2, 3, \ldots, n\}$ meta-tests. Without loss of generality, let us consider any other solution that contains h meta-tests and that executes meta-tests in order $(M_1, M_2, \ldots, M_h)$.We can write this condition as follows:

$$(\beta + \sum_{i \in N} c_i) \quad < \quad \sum_{k=1}^{h} \prod_{j=1}^{k-1} P(M_j)(\beta + \sum_{i \in M_k} c_i) \tag{6}$$

In order for 6 to hold, we find the following.

$$\beta \quad > \quad \frac{\sum_{k=2}^{h}(1 - \prod_{j=1}^{k-1} P(M_j) \sum_{i \in M_k} c_i)}{\sum_{j=1}^{h} \prod_{k=1}^{j} P(M_k)} \tag{7}$$

If we let $CV_{max} = \max_{T_k \in T} \sum_{i \in T_k} c_i$ and $P_{min} = \min_{T_k \in T} P_k$, then we see that 7 is always

satisfied if;

$$\beta > \frac{(n-1)CV_{max}}{P_{min}} \tag{8}$$

So if $\beta > \frac{(n-1)CV_{max}}{P_{min}}$ then the optimal solution will consist of a single meta-test containing all tests. □

This number could be very large depending on the problem instance. On the other hand, it is a finite computable bound for β that will make the meta-test with all tests optimal.

When $\beta = 0$ the optimal solution consists of all singletons since it is possible to improve any solution that includes a batch of tests by separating the batches into singletons. We can show the following positive upper bound for β that will also make all singletons solution optimal. Without loss of generality let us assume that meta-test X consists of tests $1, 2, \ldots, h$ where this order gives the minimum expected cost if they were executed one by one. The cost of this meta-test is $C(X) = \beta + \sum_{i=1}^{h} c_i$. Let S_X be the policy that executes these tests one by one and $C(S_X)$ be the cost of this strategy.

$$C(S_X) = \sum_{k=1}^{h} \prod_{j=1}^{k-1} p_j(\beta + c_k), \text{ where } \frac{c_1}{q_1} \leq \frac{c_2}{q_2} \leq \ldots \leq \frac{c_h}{q_h}.$$

Then we have the following.

Proposition 2.4. *If $C(S_X) < C(X)$, then meta-test X will not be part of any optimal solution.*

Proof. Let us consider any feasible solution containing meta-test X. We can obtain another feasible solution by replacing meta-test X by the singletons in X. This solution will always be better than the solution containing only X. □

Proposition 2.5. *For each fixed $(p_1, p_2, \ldots, p_n)$ and $(c_1, c_2, \ldots, c_n)$, there is a sufficiently small positive β such that the optimal solution consists of all singletons.*

Proof. By using proposition 2.4, we can claim that if $C(S_X) < C(X)$ for all meta-tests X, then

no meta-test will be part of an optimal solution. We can write these conditions as follows.

$$\sum_{i=1}^{h} \prod_{j=1}^{i-1} p_j(\beta + c_i) \quad < \quad \beta + \sum_{i=1}^{h} c_i \tag{9}$$

$$\beta \quad < \quad \frac{\sum_{i=2}^{h}((1 - \prod_{j=1}^{i-1} p_j))c_i}{\sum_{j=1}^{h-1} \prod_{k=1}^{j} p_k} \tag{10}$$

We will refer to the right hand side of the final inequality as β_X. Then we have that if $\beta < min_X \beta_X$ then there will not be any meta-tests in any optimal solution. This means that the optimal solution will consist of the singletons. □

We can also use proposition 2.4 to eliminate meta-tests that will not be in any optimal solution and decrease the size of the problem if such meta-tests exist in an instance.

A lower bound for the minimum expected cost can be computed as follows. For any problem instance let $(n_1, n_2, \ldots, n_m)$ be the vector whose i^{th} component is the number of tests in the corresponding meta-test where $n_1 \geq n_2 \geq \ldots \geq n_m$. Then

$$h = argmin_j\{\sum_{k=1}^{j} n_k \geq n\}$$

will be a lower bound on the number of meta-tests in any solution. Let C' be the cost vector C and the and P' be the probability vector P of tests sorted in descending order. Now construct the binary vector B in following fashion: Set $B[1] = 1$, then using the vector $(n_1, n_2, \ldots, n_m)$ and h which is determined previously set $B[i] = 1$ if $i = \sum_{k=1}^{s} n_k$ for $1 \leq s \leq h$ and $B[i] = 0$ otherwise. Then the lower bound is simply:

$$LB = \sum_{i=1}^{n} \prod_{j=1}^{i-1} P'[j] * (C'[i] + \beta B[i])$$

We will use this bound when we compare the performances of the heuristics.

2.3 Algorithms

In this section, we assess the performance of three heuristic algorithms on randomly generated problem instances. We will describe the algorithms one by one, discussing the pros and cons of each. The first algorithm, "Ratio Algorithm" is an adaptation of an optimal greedy algorithm for the case when batching is not allowed to our problem. The second one, Branch, is a search algorithm that we have developed. The third algorithm is a GA algorithm, that is proposed for the set partitioning problem in the literature. We also adapt this algorithm to our problem.

2.3.1 Enumeration

We have implemented a brute force enumeration algorithm to find optimal solutions for relatively small problems for benchmarking purposes. It is a recursive implementation and we use proposition 2.4 to eliminate meta-tests that will not appear in any optimal solution. The algorithm uses a partial solution C that consists of some non-intersecting meta-tests from Ω. In addition, we use proposition 2.4, to construct a set of candidate meta-tests, Ω^C, that may be part of an optimal solution that contains the meta-tests in C. Then $Enumeration(\emptyset, \Omega')$ will output an optimal solution where Ω' is obtained from Ω by deleting the meta-tests that will never appear in an optimal solution by proposition 2.4. The pseudocode of the enumeration algorithm can be seen as Algorithm 1. In addition, in order to speed up the algorithm, we start the enumeration by inserting meta-tests one by one starting with the ones with the highest number of individual tests. In order to be efficient, we also avoid checking some combinations of meta-tests multiple times. Since each test is an element of at least one maximal meta-test, any meta-test that consists of a single test is an element of Ω. We will refer to these as *SingleTests* in pseudo-codes of the Algorithms.

Algorithm 1 Enumeration(C , Ω^C)

Input: A collection C of meta-tests in Ω; and Ω^C all meta-tests in Ω that may be a part of an optimal solution with C, and sorted by number of components in descending order.

Output: A collection F of sets in Ω which give a partition of N with lowest Expected Cost.

1: **if** $C = \emptyset$ **then**
2: $BestSolution \leftarrow \emptyset$
3: $BestCost \leftarrow \infty$
4: **end if**
5: $CurrentSolution \leftarrow \emptyset$
6: $RemainingMetatests \leftarrow \Omega^C$
7: **for all** $MetatestsM \in \Omega^C$ **do**
8: **if** $RemainingMetatests = \emptyset$ **then**
9: Remove last $Metatest$ from $CurrentSolution$
10: **end if**
11: $CurrentTest \leftarrow M$
12: $CurrentSolution \leftarrow CurrentSolution \cup CurrentTest$
13: $FeasibleSolution \leftarrow CurrentSolution$
14: **for all** $SingleTests\ S$ **do**
15: **if** $MetatestsIn(CurrentSolution) \cap \{S\} = \emptyset$ **then** # $MetatestsIn(X) = \bigcup x : x \in X$
16: $FeasibleSolution \leftarrow FeasibleSolution \cup S$
17: **end if**
18: **end for**
19: **if** $ExpectedCost(FeasibleSolution) < BestCost$ **then**
20: $BestSolution \leftarrow FeasibleSolution$
21: $BestCost \leftarrow ExpectedCost(FeasibleSolution)$
22: **end if**
23: $RemainingMetatests \leftarrow \emptyset$
24: **for all** $M \in \Omega^C$ **do**
25: **if** $\{M\} \cap CurrentSolution = \emptyset$ **then**
26: $RemainingMetatests \leftarrow RemainingMetatests \cup M$
27: **end if**
28: **end for**
29: **if** $RemainingMetatests \neq \emptyset$ **then**
30: $Enumeration(CurrentSolution, RemainingMetatests)$
31: **end if**
32: **end for**
33: **return** $BestSolution$

2.3.2 Ratio Heuristic

Our first heuristic is called the Ratio Heuristic. Since the non-decreasing order of ratios gives an optimal strategy when batching of tests is not allowed (i.e. when there are only singletons), one may expect that may be a good starting point for a fast algorithm. We conducted some initial experiments by finding the optimal solutions for small sized problems. In these experiments, we observed that meta-tests or singletons with small ratios tend to be in the optimal solutions. This is not always the case mainly due to the problem of maintaining feasibility. The pseudocode of the Ratio Heuristic is shown as Algorithm 2. Mainly it always adds the meta-test with the minimum ratio among all feasible meta-tests. (That means they do not intersect with the current set of meta-tests in the solution.) At each point that a meta-test is added to the solution, we complete that partial solution with singletons and output the best solution at the end. Consequently, we pick the best solution among as many as the number of meta-tests inserted different solutions in this algorithm. Since all singletons are among the meta-tests, this approach will always provide a feasible solution for our problem. One disadvantage of this algorithm is that the input is the set of all meta-tests. We need to generate all meta-tests using the maximal meta-tests. In the worst case, this may lead to an exponential increase in the size of the input. On the other hand, in many practical cases, this would not be the case. It is also possible to follow a similar approach by using maximal meta-tests and singletons only but as expected, this version of the algorithm does not produce satisfactory results.

2.3.3 Branch

Secondly, we propose an algorithm called Branch that starts with the solution consisting of all singletons. Then, the algorithm randomly splits each maximal meta-test for a certain number of (l) times to create a set of meta-tests to be used in the next step. Then, it inserts meta-tests one by one to the solution consisting of singletons to obtain new solutions. And we split the meta-test that is just inserted in a random manner to improve that solution. We keep the best k solutions out of all solutions obtained in this manner. After the initial pool of k solutions

Algorithm 2 Ratio Heuristic(Ω)

Input: Set of all meta-tests, Ω

Output: A collection F of sets in Ω which give a partition of N.

$BestSolution \leftarrow \emptyset$
$BestCost \leftarrow \infty$
$Solution \leftarrow \emptyset$
$MetatestsSet \leftarrow \Omega$
while $MetatestsSet \neq \emptyset$ **do**
  $NewTest \leftarrow$ Metatest with lowest ratio in $MetatestsSet$
  $Solution \leftarrow MetatestsIn(Solution) \cup NewTest$ # $MetatestsIn(X) = \bigcup x : x \in X$
  for all $S \in SingleTests$ **do**
    if $MetatestsIn(Solution) \cap S = \emptyset$ **then**
      $Solution \leftarrow Solution \cup S$
    end if
  end for
  if $ExpectedCost(Solution) < BestCost$ **then**
    $BestSolution \leftarrow Solution$
    $BestCost \leftarrow ExpectedCost(Solution)$
  end if
  $RemainingMetatests \leftarrow \emptyset$
  for all $M \in MetatestsSet$ **do**
    if $M \cap MetatestsIn(Solution) = \emptyset$ **then**
      $RemainingMetatests \leftarrow RemainingMetatests \cup M$
    end if
  end for
  $MetatestsSet \leftarrow RemainingMetatests$
end while
return $BestSolution$

are formed, we try to insert feasible meta-tests to each solution to obtain new solutions. Here, feasible means the inserted meta-test should not intersect the non-singleton meta-tests in that solution. After a new meta-test is inserted to a solution, similar to what we have done before, all meta-tests in that solution are splitted randomly to improve the solution. We always keep k best solutions that have been created. We continue as long as we find better solutions. The pseudocode of Branch is shown as Algorithm 3.

2.3.4 Genetic Algorithm

As we discussed previously, the structure of our problem is very similar to the minimum cost Set Partitioning Problem (SPP) except that we are dealing with a nonlinear objective function. This fact motivated us to look for solution methods proposed for SPP and apply them to our problem. Among those solution strategies, we should find the one which is able to handle a nonlinear objective function efficiently. A Genetic Algorithm (GA) applied to SPP by [22] is a heuristic algorithm in which both good feasible solutions can be obtained and leaves no difficulty in dealing with a complex objective function. Since SPP is a highly constrained problem, it is not easy to develop a GA algorithm that produces feasible solutions most of the time. The techniques used in [22] ensure that we have a feasible solution most of the time.

The Genetic Algorithm (GA) is a suitable approach for solving variety of optimization problems. It is a simulation of evolutionary process of biological organisms in nature, which starts with an initial population (gene pool) and tries to eliminate less fit genes from population and substitute them with more qualified ones. The new produced genes, which are results of crossover and mutation operators on high fit members of population, will supersede the less qualified members of population. Therefore after a number of iterations, the population converges to an optimal (best fit) solution.

We represent the solution (gene) as a binary vector in which $S[j]$ is 1 if meta-test j contributes to the solution and 0 otherwise. Fitness value equals to objective function value and since we are dealing with a minimization problem, the lower the fitness value, the more qualified the solution is. We apply uniform crossover and mutation operators to our problem as follows.

Algorithm 3 Branch(Γ, k, l)

Input: Set of all maximal meta-tests Γ, and 2 positive integers k and l
Output: A collection F of sets in Ω which give a partition of N.

```
1:  InitialSolution ← I = ⋃ S : S ∈ SingleTests
2:  MetatestsSet ← Γ
3:  for all Metatests M ∈ Γ do
4:     while NumberSplitted < l do
5:        MetatestsSet ← MetatestsSet ∪ SplitRandomly(M)
6:        NumberSplitted ← NumberSplittted + 1
7:     end while
8:  end for
9:  CandidateSolutions ← ∅
10: BestSolutions ← InitialSolution
11: BestCost ← ∞
12: repeat
13:    Improved ← False
14:    for all Solution ∈ BestSolutions do
15:       for all Metatests M ∈ MetatestsSet : M ∩ MetatestsIn(Solution) = ∅ do
16:          NewSolution ← MetatestsIn(Solution) ∪ M
17:          for all S ∈ SingleTests do
18:             if MetatestsIn(NewSolution) ∩ S = ∅ then
19:                NewSolution ← NewSolution ∪ S
20:             end if
21:          end for
22:          CandidateSolutions ← CandidateSolutions ∪ NewSolution
23:          if Cost(NewSolution) < BestCost then
24:             Improved ← True
25:             BestCost ← Cost(NewSolution)
26:          end if
27:       end for
28:    end for
29:    SplittedSolutions ← CandidateSolutions
30:    for all Solution ∈ CandidateSolutions do
31:       NumberSplitted ← 0
32:       while NumberSplitted < l do
33:          Metatest ← SelectRandomMetatest(Solution)
34:          NewSolution ← Solution {Metatest} ∪ SplitRandomly(Metatest)
35:          NumberSplitted ← NumberSplittted + 1
36:          SplittedSolutions ← SplittedSolutions ∪ NewSolution
37:          if Cost(NewSolution) < BestCost then
38:             Improved ← True
39:             BestCost ← Cost(NewSolution)
40:          end if
41:       end while
42:    end for
43:    BestSolutions ← SolutionsWithBestCost(SplittedSolutions, k)
44: until Improved
45: return SolutionsWithBestCost(BestSolutions, 1)
```

First uniform crossover operator selects two best solutions in the gene pool and combines them to produce new solution vector. With equal probability of 0.5, bit i of the new child will get the value of i^{th} bit of Parent(1) or Parent(2). When a new child solution is obtained, mutation operator converts x randomly selected bits where x is the adaptive mutation parameter. Adaptive mutation prevents some infeasible solutions to be dominated in the gene pool and therefore expands the search space. Here in our work we apply adaptive mutation whenever infeasibility factor (number of times a row is over-covered) of a solution exceeds a threshold. Still after applying adaptive mutation not all the rows are covered only once and there exist some rows in the child solution which are under or over covered. A heuristic improvement operator is applied to reduce the number of times a row is covered and along this it tries to cover as many as possible under-covered rows. Heuristic improvement performs this by removing a meta-test from the set of columns covering an over covered row without letting other rows to get under-covered. On the other hand, it adds meta-tests to the set of columns covering an under-covered row without covering any other rows more than once. Even after using heuristic improvement operator the child solution might remain infeasible. Therefore this somehow should affect the fitness value of a solution. By this we mean that among a set of solutions with same fitness value the infeasible solutions should be less interesting. In order to take this into account, we define Unfitness value which is a number showing how infeasible a solution is. It is obtained through defining a function over number of times the rows in a solution are under or over covered. Therefore, we represent the eligibility of a solution using both Fitness and Unfitness values. Another challenge which indeed needs to be considered is the selection of leaving solution from population after a new child solution is entered to the solution pool. For this we divide the population into four mutually exclusive subgroups with respect to new child solution. Leaving member of the population is selected from the first non-empty subgroup to be replaced by the new child solution.

We implemented the GA proposed in [22] from the description in the article to the best of our understanding. We changed the definition of fitness and unfitness functions for our problem according to our objective function. The detailed pseudocodes and more information regarding the GA can be found in [22].

2.4 Computational Results

2.4.1 Random Instance Generation

We compare the performance of our algorithms on randomly generated instances. Firstly, we generate instances where we can find an optimal solution in a reasonable time so that we have some idea on the observed optimality gap of the proposed heuristic algorithms. We conducted some initial experiments to determine appropriate values for the parameters of the random instances. We made sure that the majority of optimal solutions are interesting in the sense that they do not just contain all singletons or they are not entirely composed of maximal meta-tests. We, in particular, had to be careful in determining how to set the probabilities, the fixed cost (β) and the density. For instance, if the fixed cost value is too small, then the optimal solution may consist of only singletons in most instances.

The density of an instance is the probability that a test appears in a meta-test. So if the density is d where $0 < d < 1$, the expected number of tests in a maximal meta-test is nd. After conducting some initial runs, the parameters of the experimental design are fixed as follows:

Table 2.1: Parameters of Experimental Design for Smaller Instances

Factors	Treatments
n	15, 20
t	10,12,15
Prob.	Uniform(0,1), Uniform(0.9,1)
Density	15%, 20%
β	n,$n/2$,$n/4$
c	Uniform(1,10)

In total the number of parameter settings is $2 \times 3 \times 2 \times 3 \times 2 = 72$. For each parameter setting, we independently created 10 instances, so in total we created 720 instances of this type.

Then we created larger instances where we cannot find the optimal solution in reasonable time by our brute force enumeration algorithm. The parameters of the larger instances are the same

as the smaller instances except for n, t, d. We used $n = 20, 30$, $t = 20, 30$ and $d = 20\%, 30\%$. For the larger instances, we generated 480 large instances.

2.4.2 Comparison of Algorithms

In this section, we present the results that we obtained from our experiments. All runs were performed on a server with Intel i7 4770k processor with a speed of 3.5 GHz and 16 GB RAM. The main comparison is with respect to expected costs. In addition, since we can compute a lower bound for the expected cost of each instance (see Section 2.2.2), we also compared the performances of the algorithms by comparing the additional expected costs incurred on top of the lower bound. In other words, for each instance we subtracted the lower bound from the objective function value of the solution found by each algorithm and compared the algorithms also according to this measure.

Firstly, we report the results of relatively small instances for which the optimal solution can be found easily. In particular, for these smaller instances it took the enumeration algorithm to find an optimal solution in just above 7 minutes on average (the maximum was 162 minutes). In the following tables, we will refer to the algorithms as Ratio, Branch and GA. For the 720 instances for which we can find the optimal solutions by our enumeration algorithm, we report the average optimality gaps of the algorithms and the number of times the algorithms find the optimal solutions. The optimality gap is defined as the percentage deviation of the value of the solution obtained by the algorithms over the the optimal value. The first four columns in Tables 2.2 and 2.3 describe the parameters of the instances. Table 2.2 (2.3) shows the results for cases when the probabilities are drawn from Uniform(0,1) (Uniform(0.9,1)). For each algorithm, we report the optimality gap (Opt. Gap.) and the number of times the algorithms find the optimal solutions (# Opt). For each parameter set, we report the average optimality gap over 10 runs. The table shows results with respect to all parameters but β. Each line corresponds to the accumulated results. For instance, the first line shows the results over all instances where the probabilities are drawn from (0,1) (a total of 360 instances) whereas the second line shows the results where the probabilities are drawn from uniform (0,1) and density is 0.2 (a total of 180 instances), and so on. For each line, we also report the number of

instances that are averaged for that line under the (# of ins.) column.

The Ratio Heuristic is non-parametric. For the Branch algorithm, we took $k = 20$ and $l = 10$ after conducting some initial runs for different values of k and l. These value provide a good trade-off between the running time of the algorithm and quality of the solution. For the GA, we have three parameters to decide. Population size, is the number of solutions which we should have in our solution pool. These solutions are the parents from which genetic operators produce new solutions. The more we increase the population size, the more the chance of finding new solutions would increase. We use 25 as the population size for running small and large instances. Adaptive mutation, as we explained before, is an operator that prevents converging the population toward infeasible solutions by altering some bits of the child solution. The first adaptive mutation parameter is the number of bits to be changed and we can determine it in our run configuration. We decided to choose 5 for all instances as the first adaptive mutation parameter. Another adaptive mutation parameter that takes values between 0 and 1, we set the value 0.5 in all our runs. Here we should note that adaptive mutation will be applied whenever infeasibility factor exceeds the threshold which is defined as the product of the two adaptive mutation parameters. The values for adaptive mutation parameters are the numbers we determined after observing results from running with different values.

We should note that the Ratio Algorithm and the GA algorithm take the set of all meta-tests as the input, whereas the input of the Branch algorithm is the set of maximal meta-tests. During the execution of the Branch Algorithm, the maximal meta-tests are divided into parts if that helps to improve the objective function. We have also conducted experiments by providing the set of all meta-tests to Branch and set of maximal meta-tests to Ratio Algorithm. We observed that, the Ratio Algorithm provides very poor solutions in this case. On the other hand, the Branch Algorithm outperforms all the others when the set of all meta-tests is provided as the input. The size of the set of all meta-tests is exponential in the size of the maximal meta-tests in the worst case. So here we report the performance of the Branch Algorithm when the input is the set of maximal meta-tests. So as we solve much larger problems, the running time of the Branch Algorithm will scale well by tuning the parameters while the other algorithms will

suffer in terms of running times. We do not report running times of the algorithms for these instances. The Ratio heuristic takes almost no time. Branch Algorithm takes 13 seconds on average. We give 120 seconds for the GA by allowing multiple starts.

When we look at the results, we see that especially when the probabilities are drawn from uniformly between 0 and 1 (see Table 2.2), all of the heuristics perform very well. This could be due to the fact that in these cases, once the first few components of the solution are chosen correctly, the additional costs contributed by other meta-tests become small since the cost of those meta-tests are multiplied by the product of the probabilities of the previous meta-tests, which becomes very small when probabilities are drawn from uniformly between 0 and 1. Although the optimality gaps are very small for all algorithms, Branch algorithm is able to find the exact optimal solution in the most number of times. When the probabilities are drawn from (0.9,1) since the cost contributions of meta-tests performed later become significant, the results become more interesting and we begin to observe the differences in the performances of the algorithms (see Table 2.3). In these cases, the Ratio heuristic does not perform well with an average optimality gap of almost 7.43%. The other algorithms achieve an average optimality gap of 1 to 3%, with GA performing better than Branch. In Table 2.4, we show the optimality gaps after the lower bound is subtracted from each objective function value. In other words, for each problem instance, the lower bound for the optimal objective function value is subtracted from optimal value and objective function values obtained by the heuristic algorithms. Then the gaps are calculated according to these numbers. Essentially, these are the possible percentage improvements on that part of the objective function that exceeds the lower bound. We just report these when the probabilities are drawn from uniform (0.9,1) since these are the challenging instances for the heuristic algorithms.

For large instances, we run the Ratio heuristic, Branch with $k = 20$ and $l = 10$, and GA. It is not possible to find an optimal for any instance in this test suit even if we 4 hours of computing time. We run the GA for 2 minutes for each instance allowing restarts. The average computation time required by the Branch algorithm with $k = 20$ and $l = 10$ is 18 seconds per instance. The Ratio heuristic runs in almost no time. Since we observe that the more challenging problems are those where the probabilities are drawn from uniform(0.9,1) we

Table 2.2: Results for cases where probabilities are drawn from uniform(0,1)

					Ratio		GA		Branch	
Prob.	Density	n	t	# of Instances	Opt. Gap.	# Opt.	Opt. Gap.	# Opt.	Opt. Gap.	# Opt.
Unif(0,1)	All			360	0.04	65	0.15	3	0.01	213
	0.15			180	0.05	40	0.10	2	0.00	121
		15		90	0.07	21	0.07	2	0.00	73
			10	30	0.01	8	0.04	1	0.00	24
			12	30	0.15	7	0.11	0	0.00	25
			15	30	0.05	6	0.05	1	0.00	24
		20		90	0.03	19	0.14	0	0.00	48
			10	30	0.02	6	0.06	0	0.00	15
			12	30	0.00	10	0.32	0	0.01	17
			15	30	0.06	3	0.04	0	0.00	16
	0.2			180	0.02	25	0.20	1	0.01	92
		15		90	0.02	15	0.29	1	0.00	59
			10	30	0.00	3	0.14	0	0.00	20
			12	30	0.04	6	0.60	1	0.00	22
			15	30	0.01	6	0.13	0	0.01	17
		20		90	0.03	10	0.10	0	0.02	33
			10	30	0.08	5	0.23	0	0.00	14
			12	30	0.00	1	0.06	0	0.00	9
			15	30	0.00	4	0.02	0	0.05	10

Table 2.3: Results for cases where the probabilities are drawn from uniform(0.9,1)

					Ratio		GA		Branch	
Prob.	Density	n	t	# of Instances	Opt. Gap	# Opt.	Opt. Gap	# Opt.	Opt. Gap	# Opt.
Unif(0.9,1)	All			360	7.43	1	1.07	128	3.06	27
	0.15			180	7.09	0	0.86	80	2.19	20
		15		90	6.46	0	0.04	76	1.47	19
			10	30	6.99	0	0.02	27	1.60	8
			12	30	5.90	0	0.06	27	1.32	6
			15	30	6.49	0	0.05	22	1.48	5
		20		90	7.72	0	1.68	4	2.91	1
			10	30	7.57	0	1.50	1	3.00	0
			12	30	7.56	0	1.79	2	2.37	0
			15	30	8.04	0	1.76	1	3.37	1
	0.2			180	7.78	1	1.28	48	3.94	7
		15		90	7.10	1	0.23	46	2.54	7
			10	30	7.17	1	0.21	15	2.11	3
			12	30	6.80	0	0.19	14	2.47	2
			15	30	7.32	0	0.29	17	3.04	2
		20		90	8.46	0	2.33	2	5.33	0
			10	30	8.22	0	1.91	0	5.25	0
			12	30	9.06	0	2.43	2	5.64	0
			15	30	8.09	0	2.65	0	5.10	0

Table 2.4: Optimality gaps when the lower bound is subtracted from all values

				Opt. Gap by using LB		
Prob.	Density	n	m	Branch	GA	Ratio
Unif(0.9,1)				10.00	3.68	26.87
	0.15			7.53	3.09	26.30
		15		4.96	0.19	25.97
			10	3.90	0.07	26.55
			12	5.11	0.29	22.82
			15	5.86	0.21	28.54
		20		10.11	5.98	26.62
			10	7.49	5.27	26.42
			12	10.45	6.44	25.64
			15	12.38	6.23	27.82
	0.2			12.46	4.27	27.44
		15		9.04	0.92	28.08
			10	8.10	0.95	30.06
			12	9.25	0.76	26.25
			15	9.79	1.05	27.91
		20		15.88	7.61	26.81
			10	17.33	5.96	24.99
			12	16.91	7.60	27.74
			15	13.38	9.29	27.69

only created instances by using this distribution. Since for the large instances, we do not have the optimal solutions, we compare Branch and GA against the solution found by Ratio Heuristic. We report the average percentage improvements over the ratio heuristic in Table 2.5. It turns out Branch algorithm is quite effective for larger problems. The average performance of the GA algorithm seems to be better than Branch Algorithm on average, but GA cannot find feasible solutions in 18 instances. When the problem size and density increase, the performance of the Branch algorithm deteriorates. This is essentially due to the fact that the input of this algorithm is the set of maximal meta-tests.

In general, we can say that one can choose the right algorithm depending on the size of the problem. All algorithms perform well when the input is the set of all meta-tests. In addition, for relatively smaller problems for which we can compute an optimal solution, we observe that the optimality gaps are reasonable. On the other hand, when problem size gets larger, one can use the Branch Algorithm to obtain good solutions in reasonable times. One problem with the GA algorithm is that it does not guarantee a feasible solution.

Table 2.5: Results for large instances

Prob.	Density	n	t	GA	Branch
Unif(0.9,1)				1.12	1.90
	0.2			-0.79	-0.30
		20		-3.76	-2.65
			20	-4.24	-2.78
			30	-3.29	-2.52
		30		2.19	2.04
			20	0.88	0.82
			30	3.50	3.27
	0.3			3.05	4.11
		20		-2.30	-0.30
			20	-3.32	-0.92
			30	-1.27	0.32
		30		8.50	8.52
			20	7.80	8.21
			30	9.21	8.84
Unif(0,1)				-0.23	-1.23
	0.2			-0.54	-0.91
		20		-1.02	-1.09
			20	-1.01	-1.05
			30	-1.04	-1.13
		30		-0.06	-0.72
			20	0.28	-0.18
			30	-0.40	-1.26
	0.3			0.13	-1.56
		20		-2.10	-2.55
			20	-0.99	-1.20
			30	-3.20	-3.89
		30		3.24	-0.57
			20	2.64	-0.22
			30	4.00	-0.93

3 Sequential Testing in Batches When All Subsets of Tests Are Available

An interesting special case of the problem presented in 2.12 is when all subsets of tests are available for testing. This is equivalent to having the meta-test that consists of all tests $[n]$ included in Γ, or simply having $\Omega = 2^n$, the powerset of all subsets of $[n]$. For this part we collaborated with Danny Segev and Iftah Gamzu and implemented a constant factor approximation algorithm and and an approximate integer programming formulation proposed by them. A description of implemented algorithms are in Sections 3.3.1 and 3.3.2. Detailed proofs can be found in [20].

A redefinition of the problem presented in Section 2.1 in a different manner, without including the constraint on meta-tests that can be performed together is given below.

3.1 Problem Definition

Let $X_1, \ldots, X_n$ be a collection of n independent Bernoulli random variables, with $\Pr[X_i = 1] = p_i$. We define a *testing scheme* to be an ordered partition of $[n]$, that is, a sequence of pairwise-disjoint subsets $\mathcal{S} = (S_1, \ldots, S_T)$ whose union is exactly $[n]$. Any testing scheme $\mathcal{S}$ corresponds to a sequential procedure for determining whether $\prod_{i=1}^{n} X_i = 1$ or not. In step t of this procedure, the values of $\{X_i : i \in S_t\}$ are inspected. If at least one of these variables evaluates to 0, we have just discovered that $\prod_{i=1}^{n} X_i = 0$; otherwise, we proceed to step $t + 1$.

The cost $C(S_t)$ of inspecting a subset S_t is comprised of a fixed set-up cost β and an additive testing cost of $\sum_{i \in S_t} c_i$. Here, we use c_i to denote the individual testing cost of X_i. With respect to our testing procedure, the cost $C(S_t)$ is incurred only when we did not detect a zero value in any of the preceding tests $S_1, \ldots, S_{t-1}$, an event that occurs with probability $\phi(\mathcal{S}, t) = \prod_{\tau=1}^{t-1} \prod_{i \in S_\tau} p_i$. Therefore, the expected cost of a testing scheme $\mathcal{S} = (S_1, \ldots, S_T)$ is given by

$$\mathcal{E}(\mathcal{S}) = \sum_{t=1}^{T} \phi(\mathcal{S}, t) \cdot C(S_t) \, .$$

The objective is to compute a testing scheme of minimum expected cost.

3.2 Analysis of the Problem

3.2.1 Complexity of the Problem

Definition 3.1. *k-Batch Testing* is a further restricted version of the batch testing problem where the solution should contain at most k subsets. The corresponding decision problem checks that given β, c_i, p_i and a threshold T whether there exists a solution for the k-Batch testing problem with its expected cost $\leq T$.

Note that when $k \geq N$, the k-Batch Testing problem is equivalent to our original problem.

Lemma 3.2. *There exist an unique minimizer for the k-Batch Testing problem for 2 subsets with general additive costs c_i and $p_i = e^{-c_i}$.*

Proof. A solution corresponds to two subsets of the tests, say S_1 and S_2, where $S_1 \bigcup S_2 = N$. The expected cost for the testing scheme is:

$$z = \beta + \sum_{i \in S_1} c_i + e^{-\sum_{i \in S_1} c_i} (\beta + \sum_{i \in S_2} c_i)$$

Let $x = \sum_{i \in S_1} c_i$, the cost of all tests in the first set and $C = \sum_{i \in S} c_i$, the cost of all tests. In

order to find the minimum value that this expression can assume, we need:

$$\begin{aligned} \frac{dz}{dx} &= e^{-x}(-\beta - C + e^x + x - 1) = 0 \\ e^x + x &= \beta + C + 1 \\ x &= -W(e^{\beta+C+1}) + \beta + C + 1 \end{aligned}$$

This value is unique, and it is the global minimum of the expected cost function on domain $(0, C]$. Therefore we need the sum of costs of tests in set 1 to be exactly $-W(e^{\beta+C+1}) + \beta + C + 1$ to minimize expected cost. W is the upper branch W_0 of the Lambert W function, or more commonly known as product logarithm, which is single-valued for real $x \geq -1$. □

Theorem 3.3. *2-Batch Testing is NP-complete.*

To use in our proof, let us first remind the *Subset Sum* problem:

Definition 3.4. Given a set of integers $S = \{s_1, s_2, ..., s_n\}$ and another integer k, the Subset Sum problem asks if is there a non-empty subset of S whose elements sum to k.

Proof. Given an instance of the Subset Sum problem, we can construct a 2-Batch Testing instance as following:

For each integer in S, create a test for the 2-Batch Testing problem with cost $c_i = s_i$ and probability $p_i = e^{-s_i}$. Fix $\beta = e^k + k - C - 1$, where C is the sum of all integers in S. Set $T = e^k - e^{-k} + 2k - C$.

Suppose the original subset sum problem has a solution X, $\sum_{i \in X} s_i = k$. Then the corresponding testing problem has a solution where tests generated from elements of X are in the first subset and others are in the second subset.

$$\begin{aligned} z &= \beta + k + e^{-k}(\beta + C - k) \\ &= e^k + 2k - C - 1 + e^{-k}(e^k - 1) = T \end{aligned}$$

Due to 3.2, if the subset sum problem does not have a solution, then the constructed 2-Batch

Testing won't have a solution since k is the value that gives lowest possible expected cost.

Suppose the 2-Batch Testing problem has a solution $\{S_1, S_2\}$ with expected cost of this testing scheme $\leq T$. Since an expected cost of T is only achievable when $\sum_{i \in S_1} c_i = k$. The equivalent subset sum problem has a solution X, where elements of X are exactly the tests in S_1.

□

3.2.2 Properties

It is possible to optimally solve the problem in polynomial time when the additive costs (c_i) or the probabilities (p_i) are the same by modeling the problem as a shortest path problem an an acyclic directed network. Without loss of generality, we will assume that the additive costs (c_i) are the same.

Proposition 3.1. *When all additive costs c_i (probabilities p_i) are equal, in any optimal solution the tests are in non-increasing order of their probabilities p_i (additive costs c_i).*

Proof. Let us consider any feasible solution where test i is executed before test j but $p_i > p_j$. By simply switching positions of i and j the expected cost of testing scheme can be decreased. Therefore this solution cannot be an optimal solution. □

Proposition 3.2. *When all additive costs c_i (probabilities p_i) are equal, the Sequential Testing in Batches problem can be optimally solved in polynomial time.*

Proof. We construct a directed network with $n + 1$ nodes, node 0 to node n. In this network node 0 is the special starting node and nodes 1 to n corresponds to tests in the original problem, ordered using Proposition 3.1. In the network, we include all arcs (i, j) such that $j > i$. We define the cost of arc (i, j) as

$$c_{ij} = (\prod_{k=1}^{i} p_{\pi(k)})(\beta + \sum_{k=i+1}^{j} c_{\pi(k)})$$

By these definitions, any path from node 0 to node n corresponds to a feasible solution for

the original problem. In addition, the length of any path from node 0 to node n is equal to the expected cost of the solution for the original problem that corresponds to this path. So we can solve our problem by simply solving a shortest path problem from node 0 to node n on this network. □

3.3 Algorithms

3.3.1 Constant Factor Approximation Algorithm

Structural Modifications Rather than focusing attention on the optimal testing scheme, it is instructive to consider a sequence of alterations, during which we gain much needed structural properties at the expense of slightly compromising on optimality.

For this purpose, let $\mathcal{S} = (S_1, \ldots, S_T)$ be some testing scheme, and consider the non-increasing sequence of probabilities

$$\phi(\mathcal{S}, 1) \geq \phi(\mathcal{S}, 2) \geq \cdots \geq \phi(\mathcal{S}, T) \,.$$

As shown in Figure 3.1, we can draw this sequence as a *product chart*, along with the corresponding subsets. Here, the horizontal axis displays the indices $1, \ldots, T$, while the vertical axis displays the function value $\phi(\mathcal{S}, t)$. It is worth mentioning that, by definition, $\phi(\mathcal{S}, t)$ is the product of all variable probabilities within the sets $S_1, \ldots, S_{t-1}$, i.e., does not include those in S_t.

For a parameter $\Delta > 1$, whose value will be optimized later on, we proceed by partitioning the sequence of subsets $S_1, \ldots, S_T$ by powers of Δ into (potentially empty) buckets $B_1, \ldots, B_L, B_{L+1}$. This partition is schematically illustrated in Figure 3.2. The first bucket B_1 consists of subsets for which $\phi(\mathcal{S}, t) \in (1/\Delta, 1]$, the second bucket B_2 consists of subsets with $\phi(\mathcal{S}, t) \in (1/\Delta^2, 1/\Delta]$, so forth and so on. We pick the value of L such that the next-to-last bucket B_L consists of subsets for which $\phi(\mathcal{S}, t) \in (1/\Delta^L, 1/\Delta^{L-1}]$, where L is the minimal integer such that $1/\Delta^L \leq \epsilon/n$, implying that $L = O(\frac{1}{\Delta-1} \log \frac{n}{\epsilon})$. Finally, the last bucket B_{L+1} consists of subsets with $\phi(\mathcal{S}, t) \in [0, 1/\Delta^L]$.

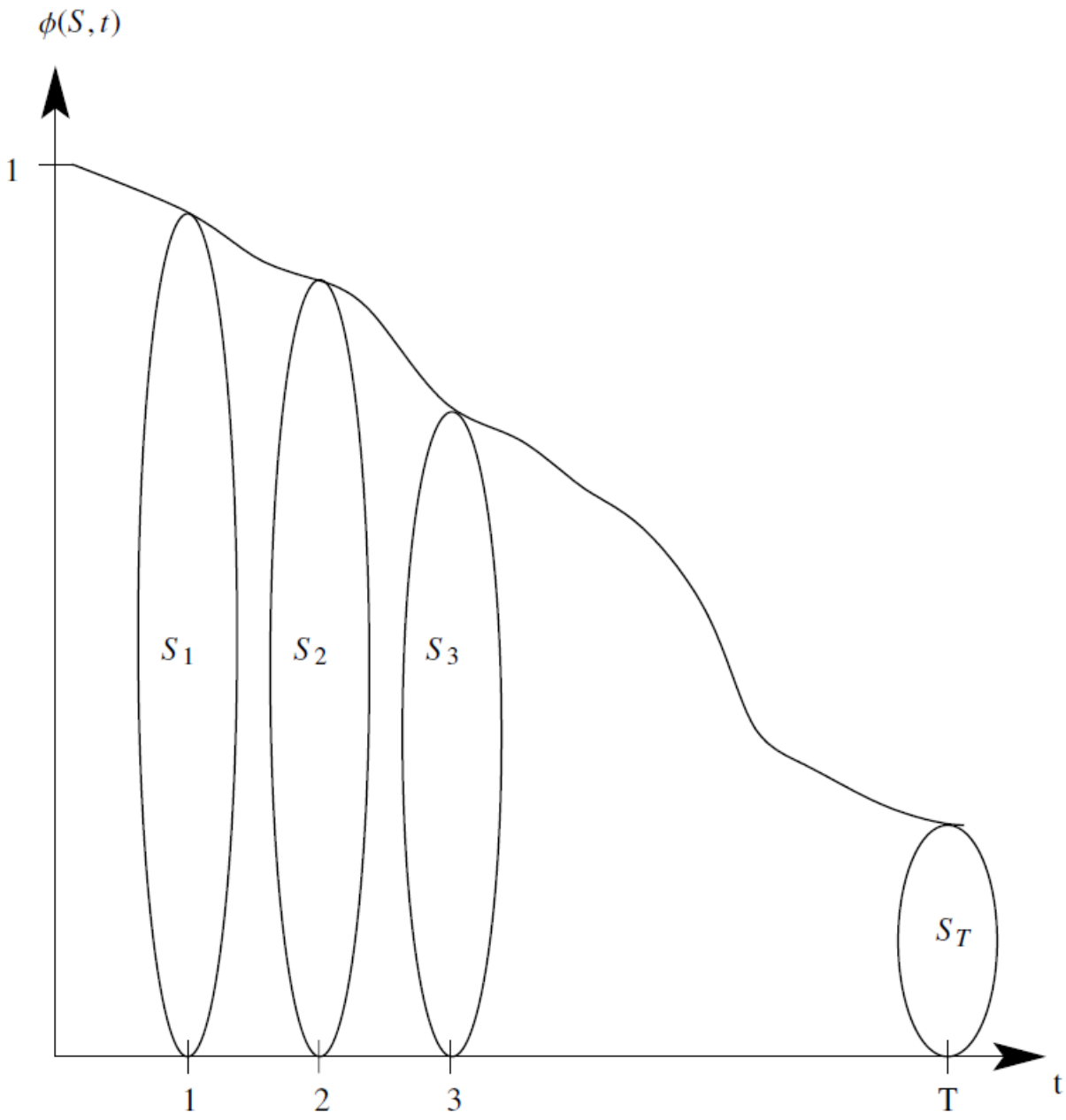


Figure 3.1: The product chart of $\mathcal{S} = (S_1, \ldots, S_T)$.

Now suppose we create a new testing scheme $\mathcal{B}$ as follows:

1. For every $1 \leq \ell \leq L$, all subsets within bucket B_ℓ are unified into a single subset. We overload notation and denote the resulting subset by B_ℓ.

2. For the last bucket, B_{L+1}, each subset is broken into singletons. We still keep the original order between subsets, whereas within the singletons of any subset, the order is arbitrary.

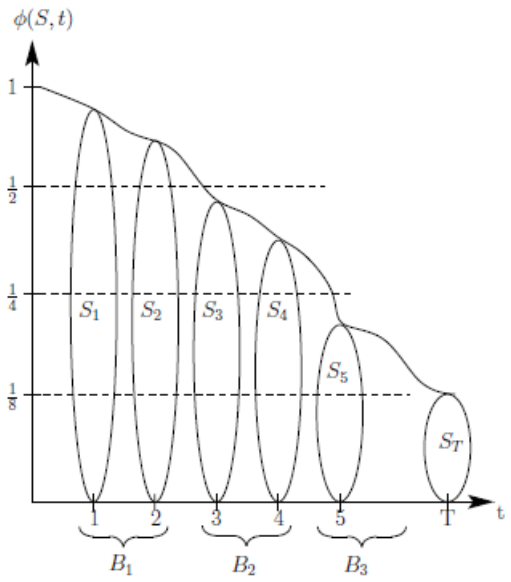


Figure 3.2: Partitioning the subsets $S_1, \ldots, S_T$ into buckets (for $\Delta = 2$).

The guessing procedure Let $\mathcal{S}^*$ be some fixed optimal testing scheme, and let $\mathcal{B}^*$ be the testing scheme that results from the structural modifications described in 3.3.1.

Guessing step 1: Non-empty buckets. We begin by guessing, for every $1 \leq \ell \leq L+1$, whether the bucket B^*_ℓ is empty or not, by means of exhaustive enumeration. The number of required guesses is $2^{O(L)} = O((n/\epsilon)^{O(1/(\Delta-1))})$.

Guessing step 2: Bucket probabilities. In addition, for every $1 \leq \ell \leq L+1$, we also guess an ϵ-estimate φ_ℓ for the probability $\phi(\mathcal{B}^*, \ell)$, i.e., a value that satisfies $\varphi_\ell \in [\phi(\mathcal{B}^*, \ell), (1+\epsilon)\cdot\phi(\mathcal{B}^*, \ell)]$. Note that we can indeed guess this value over all buckets in polynomial time, since:

- When $1 \leq \ell \leq L$: By definition, $\phi(\mathcal{B}^*, \ell) \in [1/\Delta^\ell, 1/\Delta^{\ell-1})$, and the number of guesses within this interval is $O(\Delta/\epsilon)$.
- When $\ell = L+1$: By exploiting the trivial lower bound $\phi(\mathcal{B}^*, L+1) \geq \prod_{i=1}^n p_i \geq p_{\min}^n$, the number of guesses is $O(\frac{n}{\epsilon} \cdot \log \frac{1}{p_{\min}})$. Here, $p_{\min}$ stands for the minimum probability of any variable.

Consequently, the total number of guesses is

$$O\left(\left(\frac{\Delta}{\epsilon}\right)^{L} \cdot \frac{n}{\epsilon} \cdot \log \frac{1}{p_{\min}}\right) = O\left(\left(\frac{n}{\epsilon}\right)^{O\left(\frac{\log(\Delta/\epsilon)}{\Delta-1}\right)} \cdot \log \frac{1}{p_{\min}}\right).$$

The main procedure In what follows, we use $\mathcal{A} \subseteq [n]$ to denote the collection of indices for which the corresponding random variables are still active, meaning that they have not been inspected yet, where initially $\mathcal{A} = [n]$. For $\ell = 1, \ldots, L+1$, in this order, we proceed as follows. First, if bucket B^*_ℓ is empty, we simply skip to the next step, $\ell + 1$. Otherwise, B^*_ℓ is not empty, and there are two cases, depending on whether it is the last non-empty bucket or not.

Case 1: B^*_ℓ is not the last non-empty bucket

Let $B^*_{\nu(\ell)}$ be the first non-empty bucket appearing after B^*_ℓ, meaning that $B^*_{\ell+1}, \ldots, B^*_{\nu(\ell)-1}$ are all empty. Due to our initial guessing steps, the index $\nu(\ell)$ is known, and so is the probability estimate $\varphi_{\nu(\ell)}$.

The algorithm Consider the following optimization problem:

$$\begin{aligned} \min_{S_\ell \subseteq \mathcal{A}} \quad & C(S_\ell) \\ \text{s.t.} \quad & \prod_{i \in S_\ell} p_i \leq \frac{\varphi_{\nu(\ell)}}{\prod_{i \in [n] \setminus \mathcal{A}} p_i} \end{aligned} \tag{11}$$

In other words, we wish to identify a minimum cost subset S_ℓ of active variables, such that multiplying their product $\prod_{i \in S_\ell} p_i$ with that of already-tested variables, $\prod_{i \in [n] \setminus \mathcal{A}} p_i$, results in a probability of at most $\varphi_{\nu(\ell)}$.

From a computational perspective, the above problem can be equivalently written as:

$$\begin{aligned} \beta + \min \quad & \sum_{i \in \mathcal{A}} c_i x_i \\ \text{s.t.} \quad & \sum_{i \in \mathcal{A}} (-\log p_i) x_i \geq -\log\left(\frac{\varphi_{\nu(\ell)}}{\prod_{i \in [n] \setminus \mathcal{A}} p_i}\right) \\ & x_i \in \{0, 1\} \qquad\qquad \forall\, i \in \mathcal{A} \end{aligned}$$

This is precisely an instance of the minimum knapsack problem, which is known to admit an FPTAS, through simple adaptations of existing algorithms for maximum knapsack (see, for instance, [23, 24]). Therefore, for any $\epsilon > 0$, we can compute in $\text{poly}(n, 1/\epsilon)$ time a subset $S_\ell \subseteq \mathcal{A}$ satisfying $\prod_{i \in S_\ell} p_i \leq \frac{\varphi_V(\ell)}{\prod_{i \in [n] \setminus \mathcal{A}} p_i}$ and $C(S_\ell) \leq (1+\epsilon) \cdot C(S^*)$, where S^* stands for the optimal subset here. Having determined S_ℓ, this subset of variables is the next to be inspected, we update $\mathcal{A}$ by eliminating S_ℓ, and move on to step $\ell + 1$.

Case 2: B^*_ℓ is the last non-empty bucket

Here, we simply define $S_\ell = \mathcal{A}$ and terminate the algorithm, meaning that all active variables are inspected as a single subset.

The approximation ratio is $\mathcal{E}(\mathcal{S}) \leq (1+\epsilon)^2 \cdot \frac{2\Delta-1}{\Delta-1} \cdot \mathcal{E}(\mathcal{B}^*)$. This ratio is minimized for $\Delta = 1 + 1/\sqrt{2}$.

3.3.2 ϵ -Approximate Integer Program

Due to the highly non-linear nature of its objective function, it is not entirely clear whether the batch testing problem can be expressed as an integer program of polynomial size. In what follows, we argue that this goal can indeed be obtained, by slightly setting on optimality. Specifically, we show how to formulate any batch testing instance as an integer program with $O(\frac{n^3}{\epsilon} \cdot \log \frac{1}{p_{\min}})$ variables and constraints, at the cost of blowing up its cost by a factor of at most $1 + \epsilon$.

Preliminaries For purposes of analysis, consider some fixed optimal testing scheme $\mathcal{S}^* = (S^*_1, \ldots, S^*_T)$. Any of the subsets S^*_t is naturally characterized by three attributes:

1. The index t, which refers to the position of S^*_t within the testing scheme $\mathcal{S}^*$.
2. The cost $C(S^*_t) = \beta + \sum_{i \in S^*_t} c_i$, which is uniquely determined by the variables in S^*_t.
3. The product of probabilities $\phi(\mathcal{S}^*, t) = \prod_{\tau=1}^{t-1} \prod_{i \in S^*_\tau} p_i$, which determines the coefficient of $C(S^*_t)$ in the objective function.

We begin by defining a collection of approximate configurations $\mathcal{K}$. Each configuration is a pair (t, ϕ), where t is some index in $[n]$ and ϕ is some power of $1 + \epsilon$ in $[p_{\min}^n, 1]$, implying that $|\mathcal{K}| = O(\frac{n^2}{\epsilon} \cdot \log \frac{1}{p_{\min}})$. By the discussion above, any subset S_t^* can be mapped to a unique configuration $(t, \phi) \in \mathcal{K}$, with precisely the same index t, and with $\phi(\mathcal{S}^*, t) \leq \phi \leq (1 + \epsilon) \cdot \phi(\mathcal{S}^*, t)$.

The assignment formulation For this reason, we can view the batch testing problem as that of computing a minimum-cost assignment of the variables $X_1, \ldots, X_n$ to the set of configurations $\mathcal{K}$. Specifically, for each variable X_i and configuration (t, ϕ), we introduce a binary decision variable $y_{i,(t,\phi)}$, indicating whether X_i is assigned to (t, ϕ). Also, for each configuration (t, ϕ), there is a corresponding binary variable $z_{(t,\phi)}$, indicating whether at least one of $X_1, \ldots, X_n$ is assigned to this configuration. Note that the number of variables is $O(n \cdot |\mathcal{K}|) = O(\frac{n^3}{\epsilon} \cdot \log \frac{1}{p_{\min}})$. With this notation, the objective function can be written as

$$\min \sum_{(t,\phi)\in\mathcal{K}} \phi \cdot \left(\beta \cdot z_{(t,\phi)} + \sum_{i=1}^{n} c_i y_{i,(t,\phi)} \right) ,$$

and we have three types of linear constraints:

1. Each variable X_i is assigned to exactly one configuration:

$$\sum_{(t,\phi)\in\mathcal{K}} y_{i,(t,\phi)} = 1 \qquad \forall\, i \in [n] \,.$$

2. For each configuration (t, ϕ) in use, the product of probabilities over all variables assigned to lower-index configurations is at most ϕ:

$$\prod_{\substack{(\tau,\varphi)\in\mathcal{K}:\\ \tau<t}} \prod_{i=1}^{n} p_i^{y_{i,(\tau,\varphi)}} \leq \phi^{z_{(t,\phi)}} \qquad \forall\, (t, \phi) \in \mathcal{K} \,.$$

This constraint can easily be rephrased in linear form using a logarithmic transformation:

$$\sum_{\substack{(\tau,\varphi)\in\mathcal{K}:\\ \tau<t}} \sum_{i=1}^{n} (-\log p_i) \cdot y_{i,(\tau,\varphi)} \geq (-\log \phi) \cdot z_{(t,\phi)} \qquad \forall\, (t,\phi) \in \mathcal{K}\ .$$

3. Consistency between y and z variables:

$$z_{(t,\phi)} \geq y_{i,(t,\phi)} \qquad \forall\, i \in [n],\ (t,\phi) \in \mathcal{K}\ .$$

3.3.3 Heuristics

Our first heuristic is called the Ratio Candidate Heuristic. Again using the fact that non-decreasing order of ratios gives an optimal strategy when batching of tests is not allowed (i.e. when there are only singletons), one may expect that may be a good starting point for a fast algorithm. We conducted some initial experiments by finding the optimal solutions for small sized problems. In these experiments, we observed that meta-tests or singletons with small ratios tend to be in the optimal solutions. However simply ordering tests according to their ratios does not always give the best solution. Furthermore finding the meta-test with best ratio is also a very hard problem with no better solution option than exhaustive enumeration. The pseudocode of the Ratio Candidate Heuristic can be seen as Algorithm 4. This heuristic tries to create a meta-test with good enough ratio to become a part of the solution quickly, without inspecting all possible meta-tests. The algorithm takes the N tests as input and firstly it calculates the ratios of N single tests and $N * (N-1)/2$ meta-tests that contain 2 tests. From this pool the algorithm keeps k meta-tests with best ratios as a starting point. Using this pool, the algorithm tries to insert each single test into meta-tests in the pool to create candidate meta-tests. If a candidate meta test has better ratio than meta-tests in the pool, the newly created candidate meta-test is added to the pool and the pool is updated. This step continues until no candidate has a better ratio than candidates in the pool. When improvement stops the meta-test with best ratio is added to the solution and tests in this meta-test is removed from the set of N tests. Algorithm then starts over with the reduced set and continues until all tests are used. This algorithms terminates very quickly and scales efficiently since it only inspects

at most kN meta-tests in each step. Calculating the ratio of a meta-test after the addition of another test is also a very fast operation when aggregate costs and probability of tests are kept.

Algorithm 4 RatioCandidate(N,k)

Input: Set of all tests N and a positive integer k
Output: A partition of N.

```
1:  Cand ← ∅
2:  Cand′ ← ∅
3:  Improved ← true
4:  while N ≠ ∅ do
5:     Cand ← set of meta-tests with k smallest ratios among all meta-tests consisting of at
       most 2 tests in N
6:     R_max ← ratio of the meta-test with the worst ratio in Cand
7:     Cand′ ← Cand
8:     while Improved do
9:        Improved ← false
10:       for all X ∈ Cand′ do
11:          for all y ∈ N − X do
12:             if Ratio(X ∪ {y}) < R_max then
13:                Add X ∪ {y} to Cand
14:                Delete the meta-test with the worst ratio from Cand
15:                Update R_max
16:                Improved ← true
17:             end if
18:          end for
19:       end for
20:       Add best solution B of Cand to the Solution
21:       N ← N − B
22:    end while
23: end while
24: return Solution
```

The pseudocode of our second heuristic, the Merge heuristic, can be seen as Algorithm 4. This algorithm starts with an initial solution consisting of only singleton meta-tests. Then the algorithm tries to merge every pair of meta-tests in the solution to create a new candidate solution. The meta-tests to be merged may be singleton meta-tests or meta-tests that contain multiple tests which were created in subsequent merge operations. After a candidate solution is created it is inserted into the global pool of k solutions with the best cost if the candidate is good enough. If an improvement to the best cost is made the algorithm continues with best k

solutions and tries to create new candidate solutions by merging meta-tests in these solutions. The algorithm stops and returns the solution with the lowest cost in the pool if no improvement is made. At each step of the algorithm at most kN^2 candidate solutions are inspected. Since the cost of calculating the excepted cost of a solution is much higher than calculating the ratio and more alternatives are inspected at each iteration this algorithm runs slower than Ratio Candidate Heuristic. Detailed comparisons of running times and quality of the solutions are presented in next section.

Algorithm 5 Merge(N, k)

Input: Set of all tests N and a positive integer k
Output: A partition of N.

$InitialSolution \leftarrow I = \bigcup S : S \in N$
$CandidateSolutions \leftarrow \emptyset$
$BestSolutions \leftarrow InitialSolution$
$BestCost \leftarrow \infty$
repeat
  $Improved \leftarrow False$
  for all $Solution \in BestSolutions$ **do**
    for all $Metatests\ M, N \in Solution : M \neq N$ **do**
      $Merged \leftarrow M \cup N$
      $NewSolution \leftarrow Solution - M - N$
      $NewSolution \leftarrow CandidateSolution \cup Merged$
      $CandidateSolutions \leftarrow CandidateSolutions \cup NewSolution$
      if $Cost(NewSolution) < BestCost$ **then**
        $Improved \leftarrow True$
        $BestCost \leftarrow Cost(NewSolution)$
      end if
    end for
  end for
  $BestSolutions \leftarrow SolutionsWithBestCost(SplittedSolutions, k)$
until $Improved$
return $SolutionsWithBestCost(BestSolutions, 1)$

3.4 Computational Results

We conducted numerical experiments on randomly generated instances in order to assess the quality of the solutions obtained by the Approximation Algorithm and by solving the IP model.

3.4.1 The General Setting

We first summarize the implementation framework.

a) All experiments are performed on 3.3 GHz workstation with 64 GB of RAM.

b) The IP model is implemented and solved by using OPL CPLEX 12.6.2 by the default settings.

c) The Approximation Algorithm and heuristics are implemented in C++.

d) We use the FPTAS of [25] in order to solve the knapsack problem in the Approximation Algorithm. The algorithm was proposed for the max knapsack problem. It is adapted to min knapsack for our purposes. We used the implementation of this algorithm provided in [26] and we used $\epsilon = 0.25$ as the parameter of this algorithm.

In order to have an efficient trade-off between the quality of the solutions and the solution times, the parameters of both the Approximation Algorithm and the IP model should be adjusted accordingly. Especially the size of the Integer Programming model is very dependent on the probability distribution and the ϵ value.

We also implemented a brute force enumeration algorithm similar to the Algorithm 1 that finds an optimal solution for the problem instances. The brute force enumeration algorithm evaluates all feasible solutions in a recursive manner.The brute force enumeration algorithm generates all partitions with k subsets, for $k = 1, 2, \ldots, n$. For a particular k, we apply the following procedure. Suppose we have x tests and y empty subsets. We either insert a test to an empty subset or not. If we insert a test to an empty subset, we have a problem with $x - 1$ tests and $y - 1$ empty subsets. Otherwise, we have a problem $x - 1$ tests and y empty subsets. The main difference of this algorithm from the algorithm used in Section 2 is that it does not need to keep the set of all remaining meta-tests together with each solution since the remaining meta-tests to be inserted can be found efficiently from the current solution.

3.4.2 Small-scale Instances

Firstly, we generated relatively small problem instances for which we can compute the optimal value by the brute force enumeration algorithm in reasonable times.

The parameters of the small instances are summarized below.

a) n, the number of tests is taken as 10, 11, 12, 13, 14, 15.

b) The probability distribution is chosen either Uniform(0,1) or Uniform(0.9,1).

c) The additive costs are generated randomly from Uniform(1,10)

d) The fixed cost, β is take as n, $n/2$ and $n/4$ for each setting.

The β values are determined after some initial runs. These β values ensure that the optimal solution does not typically consist of a single subsets or n subsets.

For each setting, we have created 20 independent instances. So, in total, we have 720 small sized instances. We have used the brute force algorithm to find optimal solutions for all these instances. The average computation times of the optimal algorithm can be seen in table 3.1, with respect to different n and probability distributions. The number of feasible solutions is equal to the number of partitions of a n element set which is equal to n^{th} Bell Number.Even for $n = 15$, number of solutions turns out to be 1,382,958,545. So the brute force enumeration algorithm starts becomes impractical starting at $n = 15$. For larger values of n, the solution times of the enumeration algorithm increases rapidly.

Table 3.1: Average computation times of the brute force enumeration algorithm

n	**UNIF(0,1)**	**UNIF(0.9,1)**
10	0.2	0.1
11	1.0	0.9
12	6.3	5.5
13	40.2	36.9
14	284.8	258.0
15	2050.6	1776.8

For these instances, we have tried the Approximation Algorithm for different values of ϵ ranging from 0.25 to 1. For the small instances, it turns out that the quality of the solutions do

not deteriorate even for $\epsilon = 1$. So we report the results of the Approximation Algorithm for $\epsilon = 1$. The running times for $\epsilon = 1$ are all under 1 sec. (Even for $\epsilon = 0.25$, the running times of the algorithm for these instances do not exceed 33 seconds.)

For the IP model, we report the best solution found in 10 minutes. Typically, for the small instances, OPL CPLEX can find a relatively good solution in 10 minutes. Due to the way the IP model is constructed, especially when the probabilities are drawn from (0,1), the size of the IP becomes excessive and it may take more than 24 hours to obtain an optimal solution for some instances. For the solutions of the IP model, we report the real objective function computed from the solution obtained from the IP solution. (the objective function of the IP model is approximate) For the IP, we tried 0.1, 0.2 and 0.3 for ϵ. Since we are giving the solver 10 minutes for each instance, it was more important to have a small size problem and the results were the best for $\epsilon = 0.3$. In general, depending on the time and memory available, one should adjust the ϵ value accordingly. When there is no time limit, we typically obtain better solutions for smaller ϵ and the quality of the solutions are much better than the theoretical bound.

We have tried running both heuristics with different branching parameters k. Increasing the number of solutions kept rapidly increases the running time and but stopped affecting the quality of solutions after certain thresholds for both heuristics. However since Ratio Candidate heuristic is significantly faster and uses much less resources than Merge heuristic we keep 20 best solutions for Ratio Candidate and 5 best solutions for Merge heuristic to keep running times similar. Both heuristics run faster than approximation algorithm in these small instances.

The results obtained are summarized in Tables 3.2 and 3.3. In these tables, we show the average and maximum percentage deviation from the optimal solution for each value of n and β. We observe that especially when the probabilities are drawn from 0.9-1, the solutions obtained by the IP model are better than those obtained by the Approximation Algorithm. On the other hand, the optimality gap of the Approximation Algorithm is much better than the theoretical bound. For this set of parameters, the solutions obtained cannot be worse than 14.4416 times the optimal value. Merge Heuristic performs very well on both settings,

and it founds exactly the optimal solution most of the time when probabilities are uniformly distributed. Ratio Candidate heuristics performance is better than both approximation algorithms under uniform probabilities. However Approximate IP formulation performs better than RC when probabilities are drawn from (0.9,1). In general high probability instances decrease performance of all algorithms since the effect of selecting the first meta-test to be tested well is decreased in these instances.However the IP models performance increases under this setting.To summarize, these results show that merge heuristic is better than RC heuristic in small instances and Approximation Algorithm performs better than IP model when probabilities are from (0,1) and worse when they are from (0.9,1).

Table 3.2: Average and maximum percentage gaps for small instances when probabilities are drawn from (0.9,1)

		Average Gap				**Maximum Gap**			
n	**Beta**	**Approx.**	**IP**	**RC**	**Merge**	**Approx.**	**IP**	**RC**	**Merge**
All		21.59	1.54	10.46	0.92	121.65	5.70	28.40	5.34
10		21.59	2.05	11.57	0.69	115.63	5.31	22.87	2.48
	2.5	21.67	1.63	7.80	0.40	33.69	5.06	11.85	1.33
	5	21.72	1.91	12.12	0.89	66.75	5.31	18.15	2.48
	10	21.38	2.62	14.79	0.77	115.63	5.01	22.87	2.02
11		17.79	1.54	9.62	0.77	121.65	5.70	23.78	3.96
	2.75	21.59	1.23	6.25	0.64	34.21	3.17	9.82	2.32
	5.5	18.53	1.41	9.06	0.72	68.66	2.98	17.56	2.52
	11	13.24	1.98	13.54	0.96	121.65	5.70	23.78	3.96
12		20.75	1.64	10.65	0.72	118.79	4.82	20.74	3.61
	3	25.77	1.26	7.00	0.55	37.77	3.21	10.52	1.12
	6	21.33	1.76	10.67	0.55	65.76	4.32	17.64	2.54
	12	15.14	1.91	14.29	1.05	118.79	4.82	20.74	3.61
13		19.42	1.24	11.00	0.99	36.09	3.90	28.30	3.57
	3.25	27.62	1.01	7.37	0.62	36.09	2.74	12.62	1.43
	6.5	19.87	1.45	10.29	0.77	28.76	3.90	15.74	2.40
	13	10.77	1.25	15.33	1.58	16.59	3.59	28.30	3.57
14		23.03	1.33	8.96	1.16	40.40	2.93	17.64	5.34
	3.5	30.59	1.42	5.66	0.62	39.11	2.93	10.82	1.26
	7	24.46	1.56	9.72	0.92	40.40	2.72	13.93	2.74
	14	14.03	1.01	11.50	1.93	25.19	2.90	17.64	5.34
15		26.97	1.42	10.94	1.19	47.36	3.42	28.40	4.18
	3.75	36.20	1.72	7.46	0.73	47.36	2.93	12.16	2.33
	7.5	27.97	1.65	10.40	0.83	38.73	3.42	14.90	3.49
	15	16.74	0.87	14.97	2.00	25.43	2.73	28.40	4.18

Table 3.3: Average and maximum percentage gaps for small instances when probabilities are drawn from (0,1)

		Average Gap				Maximum Gap			
n	**Beta**	**Approx.**	**IP**	**RC**	**Merge**	**Approx.**	**IP**	**RC**	**Merge**
All		3.37	8.36	0.14	0.00	28.90	241.32	5.50	0.84
10		2.26	0.39	0.31	0.00	17.55	5.30	5.50	0.04
	2.5	1.30	0.75	0.53	0.00	10.79	5.30	5.50	0.00
	5	2.05	0.25	0.29	0.00	12.98	2.50	3.09	0.00
	10	3.43	0.18	0.11	0.00	17.55	2.88	1.40	0.04
11		3.28	0.67	0.20	0.00	21.93	38.06	4.92	0.27
	2.75	1.65	1.29	0.43	0.00	18.02	9.60	4.92	0.00
	5.5	3.13	0.11	0.16	0.00	15.92	0.98	2.58	0.00
	11	5.06	0.61	0.01	0.00	21.93	38.06	0.27	0.27
12		3.54	1.81	0.03	0.00	19.54	42.21	0.94	0.02
	3	1.92	3.80	0.02	0.00	11.20	42.21	0.20	0.00
	6	3.47	1.17	0.06	0.00	17.04	19.07	0.94	0.02
	12	5.24	0.46	0.01	0.00	19.54	3.85	0.08	0.02
13		3.31	4.29	0.08	0.00	19.74	55.48	2.40	0.00
	3.25	1.45	6.61	0.07	0.00	10.14	55.48	1.29	0.00
	6.5	3.35	4.21	0.16	0.00	13.38	41.91	2.40	0.00
	13	5.13	2.06	0.01	0.00	19.74	14.59	0.11	0.00
14		3.05	12.63	0.15	0.00	14.78	114.88	4.95	0.00
	3.5	2.65	9.60	0.45	0.00	11.40	64.24	4.95	0.00
	7	3.31	18.43	0.01	0.00	14.78	114.88	0.16	0.00
	14	3.21	9.87	0.00	0.00	11.70	83.49	0.01	0.00
15		4.80	30.38	0.08	0.02	28.90	241.32	1.10	0.84
	3.75	4.49	38.06	0.03	0.00	28.90	144.42	0.37	0.00
	7.5	4.42	35.87	0.12	0.00	17.99	241.32	1.10	0.00
	15	5.48	17.20	0.07	0.06	24.03	91.18	0.76	0.84

3.4.3 Large-scale Instances

In order to compare the performance of the approximation algorithm for larger instances, we have created instances for $n = 50, 100, 150$ and 200, where all the variable costs are equal to 1 ($c_i = 1, \forall i \in N$). It is possible to solve these types of problems optimally using Proposition 3.2. In this case, in order to obtain some solutions from the IP, we only generated instances by generating probabilities from (0.9,1). For the IP, we took $\epsilon = 2$ for $n = 50$ and $\epsilon = 4$ for $n = 100$. Even for these settings, it was not possible to obtain a solution from the IP for $n = 150$ or $n = 200$. The optimality gaps are shown in Table 3.4. Whenever IP can deliver a solution, the optimality gap is much better than the Approximation Algorithm. The quality of solutions obtained by the Approximation Algorithm deteriorates as n gets large and they are much worse than the optimality gaps for the small instances. Yet again the optimality gaps are much better than the theoretical bound.

For these large instances both heuristics perform very well when probabilities are uniformly selected from(0,1). However when probabilities are selected from (0.9,1) the optimality gaps become more significant and RC heuristic becomes significantly better than Merge Heuristic. Unlike the smaller instances, Ratio Candidate Heuristic is overall better than Merge Heuristic.

Next, in Table 3.5, we show the average percentage gaps for the case when the probabilities are drawn from (0,1). Let us recall that in these cases, we cannot obtain a solution from the IP. The percentage gaps are surprisingly good for these cases and much better than the theoretical bounds.

The running times of the Approximation Algorithm for these cases are shown in Table 3.6. These are results for $\epsilon = 1$. Let us note that not only the running time but also the memory requirement of the Approximation algorithm quite depends on the value of ϵ. For instance, for $n = 100$ and $\epsilon = 0.25$, the Approximation Algorithm becomes impractical due to resource requirements. Note that Merge Heuristic become unpractical very fast and needs more computation time than Approximation Algorithm in larger instances. Ratio Candidate Heuristic is the fastest by a significant margin.

Table 3.4: Average and maximum percentage gaps for large instances when probabilities are drawn from (0.9,1)

		Average Gap				Maximum Gap			
n	**Beta**	**Approx.**	**IP**	**RC**	**Merge**	**Approx.**	**IP**	**RC**	**Merge**
All		88.50	2.07	0.63	2.23	201.56	6.62	4.91	9.76
50		41.18	2.03	2.34	1.60	69.84	6.62	4.91	3.90
	12.5	65.94	0.82	1.74	2.46	69.84	2.49	2.63	3.90
	25	38.39	2.51	2.29	1.54	42.80	3.61	4.41	3.37
	50	19.20	2.76	2.98	0.79	22.11	6.62	4.91	1.10
100		80.52	2.10	0.18	0.46	131.94	5.88	0.42	1.24
	25	125.79	0.85	0.15	0.73	131.94	3.31	0.34	1.24
	50	74.62	2.53	0.16	0.42	77.84	5.88	0.29	0.64
	100	41.16	2.93	0.22	0.23	42.58	4.11	0.42	0.36
150		106.68		0.01	0.18	171.10		0.05	0.58
	37.5	168.87		0.02	0.29	171.10		0.05	0.58
	75	97.99		0.01	0.17	99.33		0.02	0.37
	150	53.17		0.01	0.06	53.83		0.03	0.11
200		125.64		0.00	6.70	201.56		0.04	9.76
	50	200.31		0.00	6.34	201.56		0.04	8.72
	100	114.72		0.00	6.36	115.73		0.01	8.82
	200	61.87		0.00	7.39	62.61		0.01	9.76

Table 3.5: Average and maximum percentage gaps for large instances when probabilities are drawn from (0,1)

		Average Gap			Maximum Gap		
n	**Beta**	**Approx.**	**RC**	**Merge**	**Approx.**	**RC**	**Merge**
All		0.35	0.00	0.00	6.91	0.00	0.00
50		0.59	0.00	0.00	6.91	0.00	0.00
	12.5	0.08	0.00	0.00	1.49	0.00	0.00
	25	0.54	0.00	0.00	4.97	0.00	0.00
	50	1.16	0.00	0.00	6.91	0.00	0.00
100		0.31	0.00	0.00	4.97	0.00	0.00
	25	0.11	0.00	0.00	2.08	0.00	0.00
	50	0.26	0.00	0.00	3.97	0.00	0.00
	100	0.56	0.00	0.00	4.97	0.00	0.00
150		0.25	0.00	0.00	1.33	0.00	0.00
	37.5	0.00	0.00	0.00	0.01	0.00	0.00
	75	0.17	0.00	0.00	0.68	0.00	0.00
	150	0.58	0.00	0.00	1.33	0.00	0.00
200		0.24	0.00	0.00	1.50	0.00	0.00
	50	0.00	0.00	0.00	0.04	0.00	0.00
	100	0.11	0.00	0.00	1.00	0.00	0.00
	200	0.60	0.00	0.00	1.50	0.00	0.00

Table 3.6: Average computation times of the algorithms

n	**Approx.**	**RC**	**Merge**
50	0.57	0.03	2.06
100	7.86	0.16	28.22
150	43.63	0.41	149.73
200	104.33	0.83	343.96

4 Conclusion & Future Research

To the best of our knowledge this is the first attempt to model the case when certain subsets of tests can be administered together in the Sequential Testing or Function Evaluation literature. We provide and study a model for this case that has interesting applications. We show that the problem is NP-hard and we propose some heuristic algorithms and demonstrate the effectiveness of the algorithms through an experimental study. All our heuristics and proofs are also applicable when subsets of available meta-tests cannot be surely tested together. For example if the meta-test 3,4,5 is available then we do not know if 3,4 is also available unless explicitly stated. A major disadvantage of this model is it requires an exhaustive list of all meta-tests and this lead to impractical input sizes that may be exponential in some cases. A very interesting problem for future research is a new testing model where the tested subsets can intersect. In this case our problem changes from a set partition problem into a set covering problem. In this case the generally accepted rule of ordering tests with respect to ratios may not generally hold. Therefore after finding the optimal subsets to be tested, finding the optimal ordering will also be a problem.

References

[1] T. Ünlüyurt. Sequential testing of complex systems: a review. *Discrete Applied Mathematics*, 142(1-3):189–205, 2004.

[2] T. Ünlüyurt. Sequential testing of complex systems: a review. *Discrete Applied Mathematics*, 142(1-3):189–205, 2004.

[3] S.O. Duffuaa and A. Raouf. An optimal sequence in multicharacteristics inspection. *Journal of Optimization Theory and Applications*, 67(1):79–87, 1990.

[4] B. De Reyck and R. Leus. R&d-project scheduling when activities may fail. *IIE Transactions*, 40(4):367–384, 2008.

[5] B. Alidaee. Optimal ordering policy of a sequential model. *Journal of Optimization Theory and Applications*, 83:199–205, 1994.

[6] M.R. Garey. Optimal task sequencing with precedence constraints. *Discrete Mathematics*, 4:37–56, 1973.

[7] Bülent Çatay, Özgür Özlük, and Tonguç Ünlüyurt. Testant: An ant colony system approach to sequential testing under precedence constraints. *Expert Systems with Applications*, 38(12):14945–14951, 2011.

[8] Wenchao Wei, Kris Coolen, and Roel Leus. Sequential testing policies for complex systems under precedence constraints. *Expert Systems with Applications*, 40(2):611–620, 2013.

[9] D. Berend, R. Brafman, S. Cohen, S.E. Shimony, and S. Zucker. Optimal ordering of independent tests with precedence constraints. *Discrete Applied Mathematics*, 162:115 – 127, 2014.

[10] H. Kaplan, E. Kushilevitz, and Y. Mansour. Learning with attribute costs. In *STOC*, pages 356–365, 2005.

[11] Endre Boros and Tonguç Ünlüyurt. Diagnosing double regular systems. *Annals of Mathematics and Artificial Intelligence*, 26(1-4):171–191, 1999.

[12] M. Chang, W. Shi, and W. K. Fuchs. Optimal diagnosis procedures for k-out-of-n structures. *IEEE Transactions on Computers*, 39(4):559–564, 1990.

[13] R. Greiner, R. Hayward, M. Jankowska, and M. Molloy. Finding optimal satisficing strategies for and-or trees. *Artif. Intell.*, 170(1):19–58, 2006.

[14] E. Boros and T. Ünlüyurt. *Computing Tools for Modeling, Optimization and Simulation*, volume 12 of *Operations Research/Computer Science Interfaces Series*, chapter Sequential Testing of Series-Parallel Systems of Small Depth, pages 39–73. Springer, 2000.

[15] Amol Deshpande, Lisa Hellerstein, and Devorah Kletenik. Approximation algorithms for stochastic boolean function evaluation and stochastic submodular set cover. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'14, pages 1453–1467. SIAM, 2014.

[16] L. Cox, Y. Qiu, and W. Kuehner. Heuristic least-cost computation of discrete classification functions with uncertain argument values. *Annals of Operations Research*, 21:1–29, 1989.

[17] Sarah R. Allen, Lisa Hellerstein, Devorah Kletenik, and Ünlüyurt Tonguç. Evaluation of monotone dnf formulas. *Algorithmica*, pages 1–25, 2015.

[18] M. Shakeri, V. Raghavan, K.R. Pattipati, and A. Patterson-Hine. Sequential testing algorithms for multiple fault diagnosis. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 30(1):1–14, Jan 2000.

[19] R. Daldal, O. Ozluk, Selcuk B., Shahmoradi Z., and T. Unluyurt. Sequential testing in batches. *Annals of Operations Research*, 2015 (Under Revison).

[20] R. Daldal, I. Gamzu, D. Segev, and T. Unluyurt. Unpublished manuscript. 2015.

[21] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[22] P.C. Chu and J.E. Beasley. Constraint handling in genetic algorithms: The set partitioning problem. *Journal of Heuristics*, 11:323–357, 1998.

[23] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[24] Eugene L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4(4):339–356, 1979.

[25] H. Kellerer and U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3(1):59–71, 1999.

[26] http://paal.mimuw.wsu.pl/.