Error Locating Arrays, Adaptive Software Testing, and Combinatorial Group Testing

Jacob Chodoriwsky

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of Master of Science in
Mathematics [1]

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

[1]The M.Sc. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

Combinatorial Group Testing (CGT) is a process of identifying faulty interactions ("errors") within a particular set of items. Error Locating Arrays (ELAs) are combinatorial designs that can be built from Covering Arrays (CAs) to not only cover all errors in a system (each involving up to a certain number of items), but to locate and identify the errors as well. In this thesis, we survey known results for CGT, as well as CAs, ELAs, and some other types of related arrays. More importantly, we give several new results.

First, we give a new algorithm that can be used to test a system in which each component (factor) has two options (values), and at most two errors are present. We show that, for systems with at most two errors, our algorithm improves upon a related algorithm by Martínez et al. [32] in terms of both robustness and efficiency.

Second, we give the first adaptive CGT algorithm that can identify, among a given set of k items, all faulty interactions involving up to three items. We then compare it, performance-wise, to current-best nonadaptive method that can identify faulty interactions involving up to three items. We also give the first adaptive ELA-building algorithm that can identify all faulty interactions involving up to three items when safe values are known. Both of our new algorithms are generalizations of ones previously given by Martínez et al. [32] for identifying all faulty interactions involving up to two items.

Acknowledgements

As I reflect upon the trials I've faced during the time I have spent at the University of Ottawa, I am vividly reminded of the well-known, longstanding metaphor about standing on the shoulders of giants. I count myself blessed as having had many metaphorical giants to stand and lean upon.

Mom, Dad, thank you for teaching me perseverance. Without it I would surely not have accomplished this. Adrian, Sonya, and Dave, thank you for always believing in me, even when I didn't.

I thank Fabrice Colin for his generous support of my pre-master's research in graph theory and algorithms. My work with him was an irreplaceable first step in this tremendous odyssey.

I humbly thank the University of Ottawa and the Department of Mathematics and Statistics, for their excellent financial support during my time in Ottawa. Notable in the department are Suzanne Vézina, Michelle Lukaszczyk, and Carolynne Roy, who have always ensured that taking care of paperwork was more pleasant than it had to be. Very special thanks to Chantal Giroux for consistently assigning me absolutely splendid teaching assistantships, and to Steven Desjardins, for being such a pleasure to work for. Benoit Dionne, your tremendous efforts at keeping everything running smoothly are definitely noticed.

Without the wisdom of my office mates and fellow discrete mathematicians, I would be lost. Amy, you have shown me that a windowless office need not be dull.

Maryam, I find inspiration in your take-no-prisoners attitude. Patrick, I sincerely thank you for helping me to keep things in perspective. Elizabeth, your energy and encouragement have often helped to push me forward. Sebastian, your wit always keeps me on my toes, even to this day. Cate, your friendship made my first year at this university particularly memorable.

The help and support of the staff at the Office of Graduate Studies, Faculty of Science has been absolutely stellar; I sincerely appreciate the top-notch professionalism of Elvira Evangelista, Lorraine Houle, Diane Perras, and Manon Gauvreau.

Mike Newman, I may not have survived my first term of graduate school without your helpful guidance and always-approachable attitude. I also appreciate the time and effort you and Brett Stevens invested in my thesis examination. I heartily thank Paul Mezo for his endless patience, and Daniel Panario for ensuring that I will never, ever forget about Catalan numbers.

I owe very deep gratitude to my supervisors, Lucia Moura and Mateja Šajna, for both their financial support and the mentorship they blessed me with. Mateja, your combination of sharp editing skills and attention to fine detail has been a life saver; most of what I know about technical writing, I owe to you. Lucia, before I met you I would never have thought that a meeting regarding a thesis or course project could be so truly stirring. Our collaborative brainstorming sessions have absolutely made this thesis a memorable endeavour.

Most of all, I thank my wife Rebecca. Her love carries me through the days when I am at my weakest.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

Real, applied mathematical problems often require us to organize a finite set according to some constraints. Combinatorial designs allow us to satisfy given constraints by carefully arranging the elements of a set into subsets.

Consider the example of a quality control engineer working for an electronics manufacturer. Suppose he must ensure that his company produces reliable computers. The company builds the computers to various specifications, out of components from many different manufacturers. At the very least, he must detect whether there are problems with a given computer.

There may be a single dysfunctional component, or several. Worse still, there may be components which are not individually problematic, but interact in unexpected, erroneous ways with one or more other components. Ideally, our engineer should be able to efficiently detect all such problems, and locate the specific faulty components and troublesome interactions. Such a task can be accomplished with particular kinds of combinatorial designs called locating and detecting arrays [14] and error locating arrays [32].

The goal of this thesis is to efficiently create error locating arrays for a given testing problem The introductory example of manufacturing electronics is but one of many areas of applications. There are numerous others, including software testing, pharmaceutical development, agriculture, material engineering, and analysis of gene interactions. More details and references for specific applications can be found in a survey by Colbourn [12].

This chapter is organized as follows. In Section 1.1, we describe the testing problem to which we will apply error locating arrays. Our combinatorial design of choice, the error locating array, has several ancestors, some of which are described in Sections 1.2 and 1.3. Next, we motivate the need for stronger arrays in Section 1.4. We then review graph theory in Section 1.5, and we conclude the introductory chapter with an overview of the thesis in Section 1.6.

1.1 The Testing Problem

Companies with high quality standards must test their products thoroughly before releasing them to be sold. Such products are typically composed of many components, or factors, and each factor may have different options. In the introductory example of a computer, one factor is the CPU, and the options are the different specific types of CPUs available.

We typically define the options for a given factor with an alphabet whose entries are integers. For convenience, we shorten the notation of a set of consecutive integers by denoting $[a, a+b] = \{a, a+1, ..., a+b\}$, where a and b are integers, and $b \geq 0$. We use the following common convention for alphabets, and we also define strings over alphabets in anticipation of the key definition which follows this one.

Definition 1.1.1 *A g-**alphabet** is a finite set of g symbols. By convention, this set is assumed to be $[0, g-1]$. If $g = 2$, we have a **binary alphabet**.*

*A **string** over a g-alphabet is a finite sequence $S = a_1 a_2 ... a_n$ whose characters a_i are symbols from the g-alphabet. We say that the **length** of S is n. A sequence*

*$S' = a_{i_1}a_{i_2}...a_{i_m}$ is a **subsequence** of S if, for all $j < k \leq m \leq n$, we have $i_j < i_k$. We call S' a **substring** of S if $i_{j+1} = i_j + 1$ for all $j \in [1, m-1]$.*

*If we append string $T = b_1b_2...b_m$ to the end of string $S = a_1a_2...a_n$, we say that we **concatenate** S and T to make a new string $ST = a_1a_2...a_nb_1b_2...b_m$. Let $c > 0$ be an integer. We denote c copies of string S concatenated together by S^c.*

Now that we have a way to describe options for each component in a system, we formally define the problem of testing a system. For convenience, we use the definition from Maltais [31]. Throughout the combinatorial design and testing literature, tests are often defined in terms of tuples. However, we define them as strings for the sake of neater presentation.

Definition 1.1.2 *A **testing problem** is a system with k components, called factors, which are labeled by indices in $[1, k]$. The ith factor has g_i potential options, called **values** (and sometimes called **levels**). We use a g_i-alphabet to denote the possible values of factor i. For the sake of convenience, we denote such a testing problem as $TP(k, (g_1, g_2, ..., g_k))$. We shorten this notation to $TP(k, g)$ if we have a constant-size alphabet (i.e. $g_1 = g_2 = ... = g_k = g$).*

*A **test** associated with a $TP(k, (g_1, g_2, ..., g_k))$ is a string $T = T_1T_2...T_k$ of length k, where the ith factor has value $T_i \in [0, g_i - 1]$. An **array** associated with a $TP(k, (g_1, g_2, ..., g_k))$ is an $N \times k$ array A whose rows (indexed 1 to N) are tests associated with the same $TP(k, (g_1, g_2, ..., g_k))$.*

*A **subtest** of a test T with respect to the index set $A = \{a_1, a_2, ..., a_t\} \subseteq [1, k]$, denoted T_A, is the subsequence $T_{a_1}T_{a_2}...T_{a_t}$ of T. If we wish to specify that the length of T_A is t, we call T_A a **t-subtest**. Two tests T, T' are **disjoint** if, for each $i \in [1, k]$, we have $T_i \neq T'_i$. More generally, two subtests T_A, T'_A are **disjoint with respect to** A if, for each $i \in A$, we have $T_i \neq T'_i$.*

We show one example of a desktop computer testing problem in Table 1.1. In this case, a test is the choice of one CPU, one motherboard, one RAM chip, one hard

Factors	Values
1=CPU	0=AMD
	1=Intel
2=Motherboard	0=Asus
	1=Biostar
	2=EVGA
	3=Intel
3=RAM	0=Corsair
	1=Crucial
	2=Kingston
	3=OCZ
4=Hard Drive	0=Hitachi
	1=Seagate
	2=Toshiba
	3=Western Digital
5=Video Card	0=Diamond
	1=Nvidia
	2=MSI
	3=Sapphire
	4=Xfx
	5=Zotac

Table 1.1: A desktop computer testing problem.

drive, and one video card. Note that there are many ways to test such a system. We could run several common pieces of commercial software on each machine for an hour each. Alternatively, we could run one piece of software on several different operating systems. Clearly, tests can differ by the requirements of the product's users, and even by the context of the testing problem. We consider a simple testing model applied to a certain system where each test can have only one of two results: pass or fail.

Definition 1.1.3 *If a test T contains a subtest corresponding to a faulty component or a faulty combination of components, we say that T is a **failing test**. Otherwise, we say that T is a **passing test**.*

Ideally, we should be able to conduct enough tests to locate all faulty parts and all faulty combinations of parts in the system. However, for the sake of pragmatism, we are often required to find a compromise between testing many configurations of components and not conducting too many tests.

For example, consider a computer manufacturer building desktop computers with components given in Table 1.1. This is a $TP(5,(2,4,4,4,6))$. On one hand, we would only need $\max\{g_i\} = 6$ tests to know whether any individual components are faulty. However, a combination of certain components, such as a Biostar motherboard with a Zotac video card, may cause system instability, even if the individual components are not faulty. Such faults are called interaction faults (see [12] and [41]).

If we conduct enough tests to collectively cover every possible choice for each t-subset of factors, up to an integer $t \leq k$, we know whether there are any faults caused by so-called t-way interactions, defined below. We use the following definition from [31].

Definition 1.1.4 *[31] Consider a $TP(k,(g_1,g_2,...,g_k))$, and let $t \in [1,k]$ be a positive integer. A **t-way interaction** (also called a **strength-t interaction**) is a set of values assigned to t distinct factors. We denote such an interaction by $I = \{(f_1,a_{f_1}),(f_2,a_{f_2}),...,(f_t,a_{f_t})\}$ where, for every $1 \leq i \leq t$, each $f_i \in [1,k]$ is distinct, and each $a_{f_i} \in [0,g_{f_i}-1]$.*

*We say that a test $T = T_1T_2...T_k$ **covers** interaction I if $T_{f_i} = a_{f_i}$ for each $i \in [1,t]$, and we sometimes represent interaction I as a t-subtest $T_{\{f_1,f_2,...,f_t\}} = T_{f_1}T_{f_2}...T_{f_t}$. If T covers I and $f_1,f_2,...,f_t \in D \subset [1,k]$, then we say that the subtest T_D also **covers** I.*

*A test (or subtest) S **avoids** I if S does not cover I. We call 1-way and 2-way interactions **pointwise** and **pairwise** interactions, respectively. An interaction which causes a test to fail is called a **faulty interaction**, which we sometimes refer to as an **error**.*

Existing research indicates that pairwise testing is highly effective for most applications (see [7, 15, 26]). However, Kuhn et al. [27] show that higher-strength testing, with $t \in [4, 6]$, is needed for truly robust fault detection in certain situations: their empirical results indicate that 4-way testing is needed to detect at least 95% of faults in the applications they tested, and 6-way testing is needed to detect all faults.

We note here that there are two main testing methodologies: nonadaptive, and adaptive, which we define below. We focus on the former method throughout Chapters 1 and 2.

Definition 1.1.5 *If a testing method constructs each test without any knowledge of results from any other tests, it is a* ***nonadaptive*** *testing method. Otherwise, it is an* ***adaptive*** *testing method.*

In the next section, we introduce combinatorial designs which cover all t-way interactions. Such designs are arrays whose rows represent corresponding tests.

1.2 Covering Arrays

A covering array (CA) is a type of combinatorial design which, given a parameter t, covers each t-way interaction at least once. More formally, we define a covering array as follows.

Definition 1.2.1 *A* ***covering array*** *C is an $N \times k$ array with entries from a g-alphabet, such that each possible t-way interaction I of a testing problem $TP(k, g)$ occurs as a subtest of some row of C. The parameters N, t, k, and g are the* ***size, strength, number of factors****, and* ***order****, respectively. We denote such an array by $CA(N; t, k, g)$.*

Notice that a $CA(N; t, k, g)$ always exists for a testing problem $TP(k, g)$ since we can always construct a test suite composed of all k-tuples of a g-alphabet. However, we wish to minimize the number of tests, so we present some results on the minimum size N for which a $CA(N; t, k, g)$ can exist.

Definition 1.2.2 *The minimum integer N for which a $CA(N; t, k, g)$ exists is called the **covering array number**, which we denote by $CAN(t, k, g)$. A covering array of size $N = CAN(t, k, g)$ is called **optimal**.*

Regrettably, not much is known about the exact values of covering array numbers. However, some general bounds can be easily inferred. First, for any $A \subseteq [1, k]$ such that $|A| = t$, a $CA(N; t, k, g)$ must include each of the g^t possible t-subtests that are indexed by the set A at least once. Therefore,

$$g^t \leq CAN(t, k, g).$$

Likewise, any covering array of strength t also covers all $(t-1)$-subtests, and any covering array with alphabet g clearly covers all $(g-1)^t$ subtests created from a $(g-1)$-alphabet. Furthermore, we can create a covering array on $k-1$ factors by simply removing one column. Hence, we get the following inequalities.

$$CAN(t-1, k, g) \leq CAN(t, k, g)$$

$$CAN(t, k-1, g) \leq CAN(t, k, g)$$

$$CAN(t, k, g-1) \leq CAN(t, k, g)$$

Fortunately, the exact covering array number is known for pairwise interactions and binary ($g = 2$) alphabets. Katona [24] and Kleitman and Spencer [25] independently discovered and proved the following bound; see [33, Section 3.4] for a proof.

Theorem 1.2.3 *Let k be a positive integer. Then:*

$$CAN(2,k,2) = \min\left\{N : \binom{N-1}{\lceil N/2 \rceil} \geq k\right\}.$$

The matrix construction associated with the above bound is well-known and relatively simple. Let S be the set of all distinct binary N-tuples such that each tuple has a zero in the first position, and exactly $\lceil N/2 \rceil$ ones. It is easy to see that any k-subset of S forms a $CA(N;2,k,2)$.

Following Definition 1.2.1, an $N \times k$ array A is a $CA(N;2,k,2)$ if any $N \times 2$ subarray includes $00, 01, 10$, and 11 as rows. Let A be an $N \times k$ array whose columns correspond to distinct elements of S, and consider an $N \times 2$ subarray B. Every tuple in S begins with 0, so 00 is a row of B. More than half of the remaining $N-1$ positions in each column of B have 1 as an entry, so 11 is a row of B as well. The columns of B are distinct, and have the same number of ones, so 01 and 10 are also rows of B. Therefore B is a $CA(N;2,2,2)$. Clearly $k = |S| = \binom{N-1}{\lceil N/2 \rceil}$, so A contains a $CA(N;2,k',2)$ for any $k' \leq \binom{N-1}{\lceil N/2 \rceil}$.

Example 1.2.4 By Theorem 1.2.3, an optimal $CA(2,k,2)$ has four rows when $k \in [2,3]$, five rows when $k = 4$, and six rows when $k \in [5,10]$.

Consider the following arrays. It is easy to see that A is an optimal $CA(2,3,2)$, and any 4×2 submatrix of A' is an optimal $CA(2,2,2)$. Similarly, A' is an optimal $CA(2,4,2)$. Finally, A'' is an optimal $CA(2,10,2)$, and any $6 \times k'$ submatrix of A'' is an optimal $CA(2,k',2)$ for $k' \in [5,9]$.

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad A' = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad A'' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Furthermore, $CAN(2, k, 2)$ is asymptotically logarithmic in k, as shown below.

Theorem 1.2.5 *As $k \to \infty$, we have*

$$CAN(2, k, 2) \sim \log k.$$

Proof: Consider an optimal $CA(N; 2, k, 2)$. By Theorem 1.2.3, N is the smallest integer such that $\binom{N-1}{\lceil N/2 \rceil} \geq k$. Therefore,

$$\binom{N-2}{\lceil (N-1)/2 \rceil} < k \leq \binom{N-1}{\lceil N/2 \rceil}.$$

Asymptotically, $N \sim N \pm 1$, so we may assume that N is even, without loss of generality. Then the preceding inequality becomes $\binom{N-2}{N/2} < k \leq \binom{N-1}{N/2}$. A few calculations reveal that

$$\lim_{N\to\infty} \frac{\binom{N-1}{N/2}}{\binom{N-2}{N/2}} = \lim_{N\to\infty} \frac{2(N-1)}{N-2} = 2.$$

Therefore, as $N \to \infty$, we have $\frac{1}{2}\binom{N-1}{N/2} < k \leq \binom{N-1}{N/2}$. We conclude that, as $N \to \infty$, we have $k \sim c\binom{N-1}{N/2}$ for some real number $0.5 < c \leq 1$. A few further simplifications to the binomial coefficient $\binom{N-1}{N/2}$ give us:

$$k \sim c\binom{N-1}{N/2} = \frac{c}{2}\frac{N!}{(N/2)!^2}.$$

Recall the famous Stirling Formula (due to James Stirling (1692 - 1770) - see [18] for more information): $n! \sim n^n e^{-n}\sqrt{2\pi n}$ as $n \to \infty$. We apply this formula to each factorial in the previous equation. After several obvious simplifications, we have:

$$\frac{N!}{(N/2)!^2} \sim \frac{2^{N+1}}{\sqrt{2\pi N}}.$$

Therefore, $k \sim c\frac{2^N}{\sqrt{2\pi N}}$ as $N \to \infty$. We then take the logarithm (base 2) of each side, and further simplify using the well-known properties of logarithms:

$$\log_2 k \sim N - \log_2 \sqrt{N} + \log_2(c/\sqrt{2\pi}).$$

Finally, we notice that

$$N - \log_2 \sqrt{N} + \log_2(c/\sqrt{2\pi}) = N\left(1 - \frac{\log_2 \sqrt{N}}{N} + \frac{\log_2(c/\sqrt{2\pi})}{N}\right) = N\big(1 + o(1)\big).$$

Hence $\log_2 k \sim N$ as $N \to \infty$. Furthermore, $k \to \infty$ if and only if $N \to \infty$, since $\binom{N-2}{N/2} < k \le \binom{N-1}{N/2}$. We conclude that $N \sim \log_2 k$ when $k \to \infty$. ∎

We also know of a more general asymptotic bound for pairwise testing with an alphabet of fixed, constant size greater than two. A result of Gargano, Körner, and Vaccaro [20] has been applied in the context of covering arrays [12] to obtain the following result.

Theorem 1.2.6 *[20, 12] Let $g > 2$ be a positive integer. Then, as $k \to \infty$, we have*

$$CAN(2, k, g) \sim \frac{g}{2} \log k.$$

Two more general bounds on the covering array number are also known. First, we have a bound restricted only to binary alphabets.

Theorem 1.2.7 *[2, 12, 36, 37] Let t and k be positive integers. Then:*

$$CAN(t,k,2) \leq 2^t t^{\mathcal{O}(\log t)} \log k.$$

Next, we have a completely general bound, due to Godbole, Skipper, and Sunley. In [21], they study random selection, using a uniform distribution, of each entry in an $N \times k$ array from a g-alphabet. They conclude that, for N large enough with respect to t, k, and g, their random array has a nonzero probability of being a $CA(N; t, k, g)$. The general bound given below follows from their results.

Theorem 1.2.8 *[21] Let t, k, and $g \geq 2$ be positive integers, and let $w = \frac{g^t}{g^t - 1}$. Then:*

$$CAN(t,k,g) \leq \frac{(t-1)}{\log w}(1 + o(1)) \log k.$$

For a more thorough treatment of covering arrays and their associated bounds and constructions, see Colbourn's comprehensive survey [12]. For a more recent survey on binary covering arrays in particular, see Lawrence et al. [28]

1.3 Mixed Covering Arrays

In many real testing problems, some factors will have more or fewer values than others. For this reason, we briefly introduce the mixed covering array (MCA), which is a more general version of a covering array.

Definition 1.3.1 *A **mixed covering array** C is an $N \times k$ array where entries in the ith column are from a g_i-alphabet, such that each possible t-way interaction I of a testing problem $TP(k, (g_1, g_2, ..., g_k))$ occurs as a subtest of some row of C. The parameters N, t, and k are the **size, strength**, and **number of factors**, respectively. We denote such an array by $MCA(N; t, k, (g_1, g_2, ..., g_k))$.*

Example 1.3.2 The following is an example of a small mixed covering array, taken from [14]. It is an $MCA(11; 2, 6, (2, 3, 3, 3, 3, 3))$.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 2 & 2 & 1 \\ 0 & 1 & 0 & 1 & 2 & 2 \\ 0 & 2 & 1 & 0 & 1 & 2 \\ 0 & 2 & 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 2 & 1 & 0 \\ 1 & 0 & 2 & 1 & 1 & 2 \\ 1 & 2 & 0 & 2 & 1 & 1 \\ 1 & 1 & 2 & 0 & 2 & 1 \\ 1 & 1 & 1 & 2 & 0 & 2 \\ 1 & 2 & 1 & 1 & 2 & 0 \end{bmatrix}$$

We know some results about the minimum number of tests for which a covering array exists. Similarly, we wish to know some results regarding the corresponding minimum for mixed covering arrays.

Definition 1.3.3 *For a given set of parameters $t, k, g_1, g_2, ..., g_k$, the **mixed covering array number** is the minimum integer N for which an $MCA(N; t, k, (g_1, g_2, ..., g_k))$ exists. We denote it by $MCAN(t, k, (g_1, g_2, ..., g_k))$. An MCA of size $N = MCAN(t, k, (g_1, g_2, ..., g_k))$ is called **optimal**.*

We notice that, if we rearrange columns in an MCA, we get another MCA with the same parameters, possibly with a reordered alphabet tuple. Reorder the factors so that $g_1 \geq g_2 \geq ... \geq g_k$, and let $g_1, g_2, ..., g_t$ be the t largest alphabet sizes. Within the first t columns, there are $\prod_{i=1}^{t} g_i$ possible tuples, so we get one simple lower bound:

$$\prod_{i=1}^{t} g_i \leq MCAN(t, k, (g_1, g_2, ..., g_k))$$

As with CAs, any MCA of strength t covers all $(t-1)$-subtests, and any MCA with alphabets $g_1, g_2, ..., g_k$ covers all subtests created from alphabets $g'_1, g'_2, ..., g'_k$, where $g'_i \leq g_i$ for all $i \in [1, k]$. Furthermore, we can create an MCA on $k-1$ factors by simply removing one column. Hence, we get the following inequalities.

$$MCAN(t-1, k, (g_1, g_2, ..., g_k)) \leq MCAN(t, k, (g_1, g_2, ..., g_k))$$

$$MCAN(t, k-1, (g_1, g_2, ..., g_{k-1})) \leq MCAN(t, k, (g_1, g_2, ..., g_k))$$

$$MCAN(t, k, (g'_1, g'_2, ..., g'_k)) \leq MCAN(t, k, (g_1, g_2, ..., g_k))$$

Furthermore, MCAs are related to CAs in the following fundamental ways. A CA of order g_1 is an $MCA(N; t, k, (g_1, g_1, ..., g_1))$, and covers all tests that an MCA with alphabets $g_1, g_2, ..., g_k$ would cover since $g_1 = \max[g_k, g_1]$. Similarly, an MCA with alphabets $g_1, g_2, ..., g_k$ covers all tests that a CA of order g_k would cover, since such a CA is an $MCA(N; t, k, (g_k, g_k, ..., g_k))$ and $g_k = \min[g_k, g_1]$. Therefore,

$$CAN(t, k, g_k) \leq MCAN(t, k, (g_1, g_2, ..., g_k)) \leq CAN(t, k, g_1).$$

More bounds and constructions for MCAs can be found in [12, 13, 35].

1.4 Motivation for Stronger Arrays

Any company which wants to release a high-quality product on the market must be able to determine if any of the product's components or interactions are faulty. The company must then identify and repair the faulty components and interactions. For products with many components, and/or many options per component, one failing test in a covering array may not give us enough information about the faulty interactions. We notice that CAs and MCAs cover all interactions up to a given strength t, and can tell us if such errors exist, but they do not necessarily identify the errors, as we explain below.

Example 1.4.1 Consider A, the MCA given in the preceding example. Suppose that all tests pass, except for the ninth one. Every pointwise interaction is covered by a passing test, so we can conclude that there are no strength-1 errors. However, there are six strength-2 interactions which are covered only by the ninth test $T = 112021$. They are:

$$I_1 = \{(1,1),(4,0)\}$$

$$I_2 = \{(2,1),(4,0)\}$$

$$I_3 = \{(2,1),(6,1)\}$$

$$I_4 = \{(3,2),(4,0)\}$$

$$I_5 = \{(3,2),(5,2)\}$$

$$I_6 = \{(4,0),(6,1)\}$$

Test T could fail due to any combination of one or more of the above interactions, so there are anywhere between one and six errors of strength two. In this case, the results of the tests in A do not give us enough information to identify all of the errors, so we need more tests to determine exactly which interactions are faulty.

In Chapter 2, we introduce some arrays which help us identify certain errors, namely (d,t)-locating arrays, due to Colbourn and McClary [14]. We also introduce other arrays which, in addition to identifying up to a certain number of errors up to to a certain strength, also determine whether there are any as-yet unidentified errors. Such arrays are called (d,t)-detecting arrays (due to Colbourn and McClary [14]) and error locating arrays (due to Martínez et al. [32]). First, we need to review some graph theory since error locating arrays are defined in terms of graphs.

1.5 Graph Theory

We review here some graph terminology, and only those aspects of graph theory that we need in later chapters. We mostly adhere to definitions and notation from the most recent edition of Bondy and Murty [3]. For the sake of brevity, we introduce only the type of graph we need.

Definition 1.5.1 *A* ***finite simple graph*** *G* ***with loops allowed*** *is an ordered pair $\big(V(G), E(G)\big)$ of finite sets. The first set, $V(G) \neq \phi$, is called the* ***vertex set****, and its elements are called* ***vertices****. The second set, $E(G)$, is called the* ***edge set****, and its elements are called* ***edges****. Edges are subsets of $V(G)$ of cardinality 1 (****loops****) or 2 (****links****), and their elements are called the* ***end(s)*** *(or* ***endpoint(s)****) of the edge. If $e = \{u, v\}$ is an edge, then u and v are called* ***adjacent*** *vertices, and we say that e is* ***incident*** *with each of u, v, and vice-versa. We denote the number of vertices and edges of a graph G by $|V(G)|$ and $|E(G)|$, respectively, or simply by $|V|$ and $|E|$ when the context is clear.*

In a given graph G, the **degree** of a vertex v is the number of edges for which v is an end, where each loop counts as two edges. We denote this quantity by $d_G(v)$. A vertex whose degree is zero is called **isolated**. The minimum and maximum degrees of G are denoted by $\delta(G)$ and $\Delta(G)$, respectively. Two adjacent vertices are called **neighbours**. The set of all vertices adjacent to v is called the **neighbourhood** of v, and is denoted $N_G(v)$.

Graphs can be represented as matrices. There are two standard matrix representations of a graph: one that describes which of its edges are incident with particular vertices, and another which describes which vertices are adjacent to each other.

Definition 1.5.2 *Let $G = (V, E)$ be a graph such that $n = |V|$ and $m = |E|$. The* ***incidence matrix of*** *G is the $n \times m$ matrix whose entry in row i, column j is the number of times that vertex i is incident with edge j. The* ***adjacency matrix of*** *G is the $n \times n$ matrix whose entry in row i, column j is the number of edges which join vertex i with vertex j (note that a loop counts as two edges here).*

If two graphs G, H satisfy $V(G) = V(H)$ and $E(G) = E(H)$, we write $G = H$, and we call them **identical**. A graph H which satisfies $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ is called a **subgraph** of G, and we denote this relationship by $H \subseteq G$. We call H a **proper subgraph** of G, denoted $H \subset G$, if $H \subseteq G$, but $H \neq G$.

Now, let G be a graph. If $V' \subseteq V(G)$, we define the **subgraph of G induced by** V' (denoted $G[V']$) as follows: the vertex set of $G[V']$ is V', and the edge set of $G[V']$ is $E' \subseteq E(G)$ where E' is the set of all edges $e \in E$ such that both ends of e are in V'. If $E'' \subseteq E(G)$, we define the **subgraph of G induced by** E'' (denoted $G[E'']$) as follows: the edge set of $G[E'']$ is E'', and the vertex set of $G[E'']$ is the set of ends of edges in E''.

If $V'' \subseteq V(G)$, we denote the graph $G[V(G) - V'']$, obtained by deleting from G all vertices in V'' and all edges incident to at least one vertex in V'', simply as $G - V''$. If $V = \{v\}$, we write $G - v$. If $E''' \subseteq G$, we denote the graph $G[E(G) - E''']$, obtained by first deleting from G all edges in E''' and then deleting all isolated vertices, simply as $G - E'''$. If $E''' = \{e\}$, we write $G - e$.

A **proper k-colouring** of G is an assignment of colours $c \in [1, k]$ to the vertices of G such that no two adjacent vertices are assigned the same colour. Vertices that are assigned the same colour are said to be in the same **colour class**. A graph G is **k-partite** (**bipartite** if $k = 2$) if its vertex set can be partitioned into k subsets called **parts**, such that no edge has both ends in any one subset. A k-partite graph is called **equipartite** if all parts are of equal size, and any graph that has a proper k-colouring is necessarily k-partite, and its colour classes correspond to the k parts.

If a k-**partite** graph contains every possible edge from each part to every other, we call it a **complete** k-**partite** graph. If such a graph has parts of sizes $g_1, g_2, ..., g_k$, we denote it by $K_{(g_1,g_2,...,g_k)}$. If $g_1 = g_2 = ... = g_k = g$, then the graph is equipartite, and we simply write $K_{k,g}$. We make use of k-partite graphs in Chapter 3.

Hypergraphs are more general versions of graphs. We define them as follows.

Definition 1.5.3 *A* ***finite simple hypergraph*** *H is an ordered pair $\big(V(H), E(H)\big)$ of finite sets. The first set, $V(H) \neq \phi$, is called the* ***vertex set****, and its elements are called* ***vertices****. The second set, $E(H)$, is called the* ***edge set*** *(or* ***hyperedge set****), and its elements are called* ***edges*** *(or* ***hyperedges****). Edges are nonempty subsets of $V(H)$, and for the sake of consistency, we refer to the elements of an edge as the* ***end(s)*** *of the edge. We denote the number of vertices and edges of a hypergraph H by $|V(H)|$ and $|E(H)|$, respectively, or simply by $|V|$ and $|E|$ when the context is clear.*

We now give an overview of the thesis.

1.6 Overview

The rest of this thesis is structured as follows.

In Chapter 2, we introduce arrays which determine more detailed information about errors than CAs and MCAs, namely (d,t)- locating and detecting arrays [14] and error locating arrays (ELAs) [32]. We also summarize existing results for error locating arrays with binary alphabets. We give assumptions and upper bounds on the number of tests required in both the adaptive and nonadaptive cases.

In Chapter 3, we give new adaptive algorithms for error location in testing problems with a binary alphabet. First, we give an algorithm that generates a set of tests that, for each system with at most two errors of strengths up to two, contains a passing test if one exists, and otherwise determines that a passing test does not exist.

We then give a second algorithm which, if the preceding algorithm returns a passing test, generates further tests which allow it to identify and return the set of all errors in either the given system or in an equivalent one.

In Chapter 4, we introduce combinatorial group testing (CGT), and we summarize the current results for CGT algorithms. We show the relation between CGT and error locating arrays with so-called safe values, as found in [32].

In Chapter 5, we give the first adaptive CGT algorithm that can identify, among a given set of k items, all faulty interactions involving up to three items. We analyze its performance, and we compare it to current-best nonadaptive method that can identify faulty interactions involving up to three items. We also give the first adaptive ELA-building algorithm that can identify all faulty interactions involving up to three items when safe values are known.

In Chapter 6, we conclude by summarizing our main results, their importance, and avenues for future research.

Chapter 2

Arrays for Error Determination

So far, we have concerned ourselves with determining whether faulty interactions occur. However, we desire more information about faulty interactions. In this chapter, we introduce a few kinds of stronger arrays, following some necessary terminology.

2.1 Locating and Detecting Arrays

Consider a testing problem concerned with building products out of components. We must determine if there are any faulty interactions and, if so, identify all such interactions before the products can be sold on the market. We use locating arrays for this task. Furthermore, even if we locate some faults, there may be more faulty interactions we are unaware of. We need to detect whether there are more faulty interactions which we haven't located, and for this, we use detecting arrays.

The entire contents of this section regarding locating and detecting arrays are due to Colbourn and McClary [14]. We introduce the arrays here, beginning with some notation and prerequisite definitions.

Let I be a strength-t interaction in a $TP(k, (g_1, g_2, ..., g_k))$, as defined in Definition 1.1.4, and let A be an $N \times k$ array. Define $\rho(A, I)$ as the set of all rows of A

which cover interaction I. More generally, we define the set of all rows covering the interactions in a set $\mathcal{I}$ as $\rho(A,\mathcal{I}) = \cup_{I\in\mathcal{I}}\rho(A,I)$.

Now, suppose T is a t-way interaction, and that $S \subset T$ is an interaction of lower strength such that $\rho(A,S) \subseteq \rho(A,T)$. In this case, if T is faulty, then we cannot determine whether or not S is also faulty. We can, in practice, locate only independent faults (defined below) unless we know beforehand the strengths of the faults. This leads us to the following useful definition.

Definition 2.1.1 *Within a given testing problem, a set $\mathcal{I}$ of interactions is **independent** if, for all $I \in \mathcal{I}$, there is no $J \in \mathcal{I}$ such that $J \subset I$.*

In the following definition, let $\mathcal{I}_t$ be the set of all t-way interactions in the given $TP(k,(g_1, g_2, ..., g_k))$, and let $\overline{\mathcal{I}_t}$ be the set of all interactions of strengths t or less.

Definition 2.1.2 *Let A be an array associated with a $TP(k,(g_1, g_2, ..., g_k))$. Then A is a (d,t)-**locating array** if it satisfies*

$$\rho(A,\mathcal{T}_1) = \rho(A,\mathcal{T}_2) \iff \mathcal{T}_1 = \mathcal{T}_2$$

*whenever $\mathcal{T}_1,\mathcal{T}_2 \subseteq \mathcal{I}_t$ and $|\mathcal{T}_1| = d = |\mathcal{T}_2|$. We also say that A **locates** d errors, each of strength t.*

*Similarly, A is a $(d,\hat{t})$-**locating array** if it satisfies*

$$\rho(A,\mathcal{T}_1) = \rho(A,\mathcal{T}_2) \iff \mathcal{T}_1 = \mathcal{T}_2$$

*whenever $\mathcal{T}_1,\mathcal{T}_2 \subseteq \overline{\mathcal{I}_t}$ and $|\mathcal{T}_1| = d = |\mathcal{T}_2|$. We also say that A **locates** d errors, each of strength up to t.*

*Furthermore, A is a $(d,\overline{t})$-**locating array** if it satisfies*

$$\rho(A,\mathcal{T}_1) = \rho(A,\mathcal{T}_2) \iff \mathcal{T}_1 = \mathcal{T}_2$$

whenever $\mathcal{T}_1, \mathcal{T}_2 \subseteq \overline{\mathcal{I}_t}$ are independent and $|\mathcal{T}_1| = d = |\mathcal{T}_2|$. We also say that A ***locates*** *d independent errors, each of strength up to t.*

*If we relax the above definitions so that $|\mathcal{T}_1| \leq d$ and $|\mathcal{T}_2| \leq d$ rather than requiring $|\mathcal{T}_1| = d = |\mathcal{T}_2|$, then A is a $(\overline{d}, t)$**-locating array** which locates d or fewer errors of strength t, a $(\overline{d}, \hat{t})$**-locating array** which locates d or fewer errors of strengths up to t, or a $(\overline{d}, \overline{t})$**-locating array** which locates d or fewer independent errors of strengths up to t.*

From a (d, t)-locating array (or one of its variations) we may be able to infer the existence of some faulty interactions of strength greater than t, given the results of some tests. However, we cannot make guarantees about interactions of strength $t' > t$. However, if an array for a testing problem is (d, t)-detecting (as defined below), then we can know whether there are more than d faulty t-way interactions.

Definition 2.1.3 *Let A be an array associated with a $TP(k, (g_1, g_2, ..., g_k))$, and let $\mathcal{T}$ be a set of d interactions of strength t in the given testing problem. Then A is a (d, t)**-detecting array** if it satisfies*

$$\rho(A, T) \subseteq \rho(A, \mathcal{T}) \iff T \in \mathcal{T}$$

whenever $\mathcal{T} \subseteq \mathcal{I}_t$, $|\mathcal{T}| = d$, and $T \in \mathcal{I}_t$. If this is the case, we also say that A ***detects*** *whether there are more than d errors, each of strength t.*

*Now, let $\mathcal{T}$ is a set of d interactions of strengths up to t. Then A is a $(d, \hat{t})$**-detecting array** if it satisfies*

$$\rho(A, T) \subseteq \rho(A, \mathcal{T}) \iff T \in \mathcal{T}$$

whenever $\mathcal{T} \subseteq \overline{\mathcal{I}_t}$, $|\mathcal{T}| = d$, and $T \in \overline{\mathcal{I}_t}$. If this is the case, we also say that A ***detects*** *whether there are more than d errors, each of strengths at most t.*

*Finally, let $\mathcal{T}$ be a set of d independent interactions of strengths up to t. Then A is a $(d, \overline{t})$-**detecting array** if it satisfies*

$$\rho(A, T) \subseteq \rho(A, \mathcal{T}) \iff T \in \mathcal{T}$$

*whenever $\mathcal{T} \subseteq \overline{\mathcal{I}_t}$, $|\mathcal{T}| = d$, $T \in \overline{\mathcal{I}_t}$, and $\mathcal{T} \cup T$ is independent. If this is the case, we also say that A **detects** whether there are more than d independent errors, each of strength up to t.*

*If we relax the above definitions so that $\mathcal{T}$ is a set of at most d interactions, then A is a $(\overline{d}, t)$-**detecting array**, a $(\overline{d}, \hat{t})$-**detecting array**, or a $(\overline{d}, \overline{t})$-**detecting array**, respectively.*

We now give some examples to further clarify the above definitions.

Example 2.1.4 We apply the following two arrays from [14] to a $TP(6, (2, 3, 3, 3, 3, 3))$.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 2 & 1 \\ 0 & 1 & 0 & 1 & 2 & 2 \\ 1 & 2 & 1 & 0 & 1 & 2 \\ 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 2 & 1 & 0 \end{bmatrix} \qquad A' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 2 & 1 \\ 0 & 1 & 0 & 1 & 2 & 2 \\ 1 & 2 & 1 & 0 & 1 & 2 \\ 1 & 2 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 2 & 1 & 0 \\ 0 & 2 & 2 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 2 & 2 \end{bmatrix}$$

We begin by associating the $(1, 1)$-locating array A ($d = 1$ and $t = 1$) with our given testing problem. Given any 1-way interaction (f, v), no other 1-way interaction occurs in exactly the same set of rows. For instance, let $T_1 = \{(1, 0)\}$, $T_2 = \{(2, 0)\}$, and $\mathcal{T}_i = \{T_i\}$ for $i = 1, 2$. Now notice that:

$$\rho(A, \mathcal{T}_1) = \{1, 2, 3\} \neq \{1, 2\} = \rho(A, \mathcal{T}_2)$$

However, A is not $(1,1)$-detecting. We see that $\rho(A, T_2) \subseteq \rho(A, \mathcal{T}_1)$ but $T_2 \notin \mathcal{T}_1$.

Fortunately, we can associate a $(1,1)$-detecting array A' with our given testing problem by appending two additional rows to A. For instance, we see that $\rho(A', \mathcal{T}_1) = \{1,2,3,7\} = \rho(A', T_1)$ and $\rho(A', \mathcal{T}_2) = \{1,2,8\} = \rho(A', T_2)$, so we have:

$$\rho(A', T_1) \not\subseteq \rho(A', \mathcal{T}_2) \text{ and } \rho(A', T_2) \not\subseteq \rho(A', \mathcal{T}_1).$$

To further show that A' is indeed $(1,1)$-detecting, we would need to verify that for any two strength-1 interactions T and T', we have $\rho(A', T) \subseteq \rho(A', T') \iff T = T'$. However, A' is not $(2,1)$-locating. Let $\mathcal{T}_3 = \big\{\{(1,1)\}, \{(2,2)\}\big\}$ and $\mathcal{T}_4 = \big\{\{(3,2)\}\big\}$. Then notice that:

$$\rho(A', \mathcal{T}_4) = \{5,6,7\} \subseteq \{4,5,6,7,8\} = \rho(A', \mathcal{T}_3), \text{ but } T_4 \notin \mathcal{T}_3.$$

If the tests corresponding to rows $4, 5, 6, 7,$ and 8 all fail when applied to a system with two strength-1 errors, then we cannot determine whether the error set is $\big\{\{(1,1)\}, \{(2,2)\}\big\}$, or $\big\{\{(1,1)\}, \{(3,2)\}\big\}$.

For a much richer set of examples of locating and detecting arrays, refer to Colbourn and McClary [14]. We now summarize their known results regarding locating and detecting arrays. First, we count the number of possible interactions of particular sizes.

The number of possible t-way interactions is given by

$$\tau_t = \sum_{I \subseteq [1,k], |I| = t} \left(\prod_{i \in I} g_i \right).$$

The total number of possible s-way interactions, for all $s \in [1,t]$, is given by

$$\gamma_t = \sum_{s=1}^{t} \tau_s.$$

The known relationships between detecting and locating properties of an array are given in Figure 2.1. The symbol $+$ denotes the assumption that $d < \tau_t$, and $*$ denotes the assumption that $d < \gamma_t$.

$$\begin{tikzcd}
(\overline{d},\hat{t})\text{-detecting} \arrow[r, Rightarrow] \arrow[d, Leftrightarrow, "*"] & (\overline{d},\overline{t})\text{-detecting} \arrow[r, Rightarrow] \arrow[d, Leftrightarrow, "+"] & (\overline{d},t)\text{-detecting} \arrow[d, Leftrightarrow, "+"] & \\
(d,\hat{t})\text{-detecting} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & (d,\overline{t})\text{-detecting} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & (d,t)\text{-detecting} \arrow[d, Rightarrow] & \\
(\overline{d},\hat{t})\text{-locating} \arrow[r, Rightarrow] \arrow[d, Rightarrow, "*"] & (\overline{d},\overline{t})\text{-locating} \arrow[r, Rightarrow] \arrow[d, Rightarrow, "+"] & (\overline{d},t)\text{-locating} \arrow[r, Rightarrow, "+"] \arrow[d, Rightarrow, "+"] & (d-1,t)\text{-detecting} \\
(d,\hat{t})\text{-locating} \arrow[r, Rightarrow] & (d,\overline{t})\text{-locating} \arrow[r, Rightarrow] & (d,t)\text{-locating} \arrow[r, Rightarrow, "+"] & (d-1,t)\text{-locating}
\end{tikzcd}$$

Figure 2.1: Relationships between detecting and locating arrays [14].

For example, consider a (d, t)-detecting array A for $d < \tau_t$. Then A is also a $(\overline{d}, t)$-detecting array, and vice-versa, as given in Figure 2.1.

If we wish to construct a detecting array, we can sometimes do so by constructing a covering array of higher strength.

Theorem 2.1.5 *[14] Let $d < \tau_t$. Then every $CA(N; t+d, k, g)$ with $d < g$ is a $(\overline{d}, t)$-detecting array. More generally, for $d < g_k \leq g_{k-1} \leq ... \leq g_1$, every $MCA(N; t+d, k, (g_1, g_2, ..., g_k))$ is a $(\overline{d}, t)$-detecting array.*

Similarly, every $(\overline{d}, t)$-detecting array (with all factors from the same alphabet) is also a covering array.

Theorem 2.1.6 *[14] Every $(\overline{d}, t)$-detecting array is a $CA(N; t+1, k, d+1)$.*

Colbourn and McClary also give a table of bounds for parameters of detecting and locating arrays, and associated necessary and sufficient conditions governing their existence. We reproduce that table here, as Table 2.1, with factors rearranged to match our order of alphabet sizes: $g_1 \geq g_2 \geq ... \geq g_k$.

Type of Array		**Constraint**
(d,t)-	locating	$d < \min(g_k+1, g_{k-1})$ or $d = \tau_t$ if $k > t$
		$d \le \tau_t$ if $k = t$
	detecting	$d < g_k$ or $d = \tau_t$ if $k > t$
		$d \le \tau_t$ if $k = t$
$(\overline{d},t)$-	locating	$d < \min(g_k+1, g_{k-1})$ if $k > t$
		any d if $k = t$
	detecting	$d < g_k$ if $k > t$
		any d if $k = t$
$(d,\overline{t})$-	locating	$d < \min(g_k+1, g_{k-1})$ or $d = \tau_t$
	detecting	$d < g_k$ or $d = \tau_t$
$(\overline{d},\overline{t})$-	locating	$d < g_k$
	detecting	$d < g_k$
$(d,\hat{t})$-	locating	$d \in \{0, 1, \gamma_t\}$
	detecting	$d \in \{0, \gamma_t\}$
$(\overline{d},\hat{t})$-	locating	$d \in \{0, 1\}$
	detecting	$d = 0$

Table 2.1: Existence constraints for detecting and locating arrays [14].

We notice the severe restrictions on the number of errors we can locate or detect. For instance, consider $(\overline{d},\overline{t})$- detecting and locating arrays. In the context of a testing problem whose smallest alphabet is binary, such arrays only exist if $d = 1$!

For more results concerning detecting and locating arrays, see [14].

2.2 Error Locating Arrays

We recall that detecting and locating arrays may not exist if d is too large relative g_k, the size of the smallest alphabet. Clearly, we need combinatorial designs which exist for larger values of d relative to g_k. In this section, we give definitions, notation, and properties of a new combinatorial design, called the error locating array, from Martínez et al. [32]. This type of array determines whether all errors in a system can be identified, and, if so, it identifies all of them, up to a certain strength. This does depend, however, on some assumptions regarding the structure of the errors, which we define in terms of graphs.

Suppose we want to model a system with a graph. First, a simple graph with loops allowed (as in Definition 1.5.1) can be used to model a system with faulty individual components (modeled by loops) and with faulty pairwise interactions (modeled by edges). In the context of a testing problem, components of the same type cannot interact, because they represent different values of the same factor. For example, an Asus motherboard cannot interact with an Intel motherboard, because a desktop computer contains only one or the other. For this reason, we model a testing problem using a multipartite graph, where each part represents a type of component, and contains vertices representing options (values) for that component. We also need to consider faulty interactions of strength higher than 2, so instead of modeling a testing problem with a graph, we use a hypergraph.

Definition 2.2.1 *[32] Let $H_{t,(g_1,g_2,\ldots,g_k)}$ denote a k-partite hypergraph with k parts of sizes $g_1, g_2, \ldots, g_k$, respectively, and hyperedges of cardinality t. Its vertex set is $\{v_{i,a_i} : i \in [1,k]$ and $a_i \in [0, g_i - 1]\}$. Replace t by $\bar{t}$ to allow edges of cardinalities up to t, and simplify the notation to $H_{t,k,g}$ or $H_{\bar{t},k,g}$ when all factors (parts) have the (alphabet) size $g = g_1 = g_2 = \ldots = g_k$. We associate with each $TP(k, (g_1, g_2, \ldots, g_k))$ an **error hypergraph** H of the form $H_{t,(g_1,g_2,\ldots,g_k)}$ such that the ith part of H corresponds to the ith factor of the TP $\big($with vertices **labeled** by the (factor, value) pairs (i, a_i), $a_i \in [0, g_i - 1]\big)$, and each edge $e_I = \{v_{i,a_i} : (i, a_i) \in I\}$ of H corresponds to a faulty interaction I in the testing problem.*

A test $T = T_1T_2...T_k$ associated with our given TP ***avoids*** *H if, for all $D \subseteq [1, k]$, we have $\{v_{i,T_i} : i \in D\} \notin E(H)$. An interaction I is called* ***relevant*** *to H if I contains no proper subset $J \subset I$ such that $e_J \in E(H)$. If $t \leq 2$, then we have an* ***error graph*** *G of the form $G_{(g_1,g_2,...,g_k)}$, or simply $G_{k,g} = H_{\overline{2},k,g}$ if $g = g_1 = g_2 = ... = g_k$.*

Our goal is to establish the existence of arrays that can determine whether each relevant interaction is faulty or not.

Definition 2.2.2 *Let H be a hypergraph of the form $H_{t,(g_1,g_2,...,g_k)}$ associated with a $TP(k, (g_1, g_2, ..., g_k))$. A t-way interaction $I = \{(f_1, a_1), (f_2, a_2), ..., (f_t, a_t)\}$ and its corresponding edge e_I are* ***locatable with respect to*** *H if there exists a test $T = T_1T_2...T_k$ with $T_{f_i} = a_i$ for all $(f_i, a_i) \in I$ that avoids* $\begin{cases} H - e_I & \text{if } e_I \in E(H) \\ H & \text{otherwise.} \end{cases}$

We say that such a test T ***locates*** *interaction I (and edge e_I)* ***with respect to*** *H. A hypergraph H is called t-**locatable** if every t-way interaction is locatable with respect to H. More generally, a hypergraph H, with $E(H)$ independent (in the sense of independent interactions, as in Definition 2.1.1), is called $\overline{t}$-**locatable** if, for all $s \in [1, t]$, every relevant s-way interaction is locatable with respect to H. For simplicity, we call an interaction (and its corresponding edge)* ***locatable*** *when the context is clear.*

Now that locatability is defined for interactions and hypergraphs, we define error locating arrays.

Definition 2.2.3 *Let H be a hypergraph of the form $H_{t,(g_1,g_2,...,g_k)}$ associated with a $TP(k, (g_1, g_2, ..., g_k))$. An* ***error locating array of fixed strength*** *t for H is an $N \times k$ array A where each column i has symbols from a g_i-alphabet, such that every t-way interaction I (where I is locatable with respect to H) is located with respect to H by at least one test T that is a row of A. We denote this by $ELA(N; t, H)$.*

Note that one array A may be an $ELA(N; t, H)$ as well as an $ELA(N; t, H')$ for distinct hypergraphs H and H'. We also notice that, following the two preceding definitions, the next two statements are equivalent for a given hypergraph H.

1. An ELA with strength t exists for H.
2. H is t-locatable.

Definition 2.2.4 *Let $\mathcal{H}$ be a class of hypergraphs of the form $H_{t,(g_1,g_2,...,g_k)}$, each of which is associated with the same $TP(k, (g_1, g_2, ..., g_k))$. Then an ELA $(N; t, \mathcal{H})$ is an array that is an $ELA(N; t, H)$ for all $H \in \mathcal{H}$.*

These definitions can be further generalized for hypergraphs whose edges have varied cardinalities (representing errors of varying strengths in the associated testing problem).

Definition 2.2.5 *Let H be a hypergraph of the form $H_{\bar{t},(g_1,g_2,...,g_k)}$ associated with a $TP(k, (g_1, g_2, ..., g_k))$. An* ***error locating array of full strength up to*** *t for H is an $N \times k$ array A where each column i has symbols from a g_i-alphabet, such that every relevant s-way interaction I with $s \in [1, t]$ (where I is locatable with respect to H) is located with respect to H by at least one test T that is a row of A. We denote this by $ELA(N; \bar{t}, H)$. Given a class $\mathcal{H}$ of hypergraphs of the form $H_{\bar{t},(g_1,g_2,...,g_k)}$, an $ELA(N; \bar{t}, \mathcal{H})$ is an array that is an $ELA(N; \bar{t}, H)$ for all $H \in \mathcal{H}$. When our hypergraph is simply a graph G, we simplify the notation as follows. Let $G_{(g_1,g_2,...,g_k)}$ and $ELA(N; G)$ denote $H_{\overline{2},(g_1,g_2,...,g_k)}$ and $ELA(N; \overline{2}, G)$, respectively. We refer to a $\overline{2}$-locatable graph as* ***locatable****.*

For a given hypergraph H, the following two statements are equivalent.

1. An ELA with full strength up to t exists for H.
2. H is $\bar{t}$-locatable.

As with CAs and MCAs, we wish to minimize the number of tests, so we define the minimum size N for which an $ELA(N; t, H)$ can exist.

Definition 2.2.6 *Suppose that an $ELA(N; t, H)$ exists for a given $TP(k, (g_1, g_2, ..., g_k))$ whose associated hypergraph is H. If there is no $ELA(N'; t, H)$ such that $N' < N$, then we call N the **error locating array number**, which we denote by $ELAN(t, H)$. An error locating array of size $N = ELAN(t, H)$ is called **optimal**. We define the ELAN for hypergraphs with edges of cardinality up to t by replacing t by $\bar{t}$ in the above definition. We denote the ELAN for a graph G by $ELAN(G)$ (with the assumption that $t \leq 2$, since loops are allowed, by 1.5.1).*

For results concerning error locating array numbers, see Danziger et al. [16]. We now give an example of an ELA in a case where no (d, t)-detecting array exists.

Example 2.2.7 Consider the following home theatre testing problem, given in Table 2.2. Some errors could occur with such a setup. For example, suppose that images from our Xbox do not display properly on our Sylvania TV. In this case, interaction $I_1 = \{(1, 3), (2, 1)\}$ is faulty. If this was our only error, we could identify it using a $(1, 2)$-detecting array.

Factors	**Values**
1=Game System	0=Sega Dreamcast
	1=Sony PlayStation 2
	2=Nintendo GameCube
	3=Microsoft Xbox
2=TV	0=Samsung
	1=Sylvania
3=Sound System	0=LG
	1=Philips
4=Universal Remote	0=Philips
	1=Zenith

Table 2.2: A home theatre testing problem.

Suppose that we have two additional errors: the audio from our Dreamcast sounds odd when played through our Philips sound system, and not all buttons on our Zenith universal remote work with our Philips sound system. Then interactions $I_2 = \{(1,0),(3,1)\}$ and $I_3 = \{(3,1),(4,1)\}$ are also errors. For the home theatre testing problem described here, the faulty (pairwise) interactions are represented by the edges in a $G_{(4,2,2,2)}$ which is given in Figure 2.2.

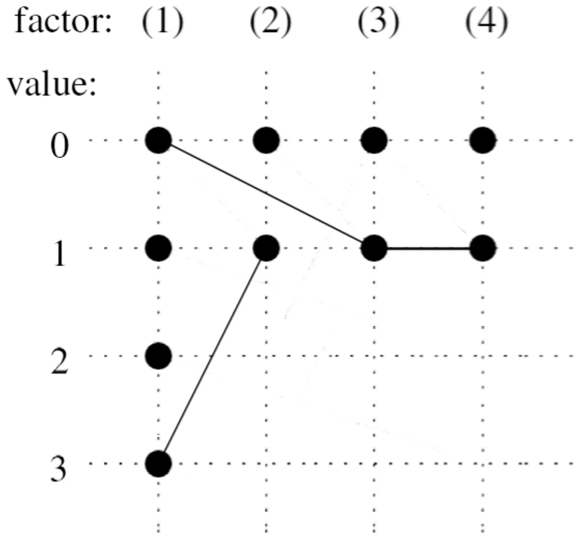


Figure 2.2: A $G_{(4,2,2,2)}$ for the home theatre testing problem from Table 2.2.

No $(3,2)$-detecting array exists since $d = 3 \geq g_k = 2$, but we can construct an ELA. In particular, the array on Page 31 is an $ELA(11; G)$. The eleven rows represent tests of the system, and results of each test are given. Errors and their corresponding failing test results are marked in bold in the array.

Notice that the graph in Figure 2.2 has no loops and not many edges because there are no faulty single components and not many faulty 2-way interactions. Also, notice that each nonfaulty pairwise interaction (and hence, each pointwise interaction) occurs in some passing test, and that the only 2-way interactions not covered by any passing test are I_1, I_2, and I_3 as described above, and each one occurs in a distinct failing test. We conclude that the array is an $ELA(G)$.

$$\begin{array}{r} test\,1 \\ test\,2 \\ test\,3 \\ test\,4 \\ test\,5 \\ test\,6 \\ test\,7 \\ test\,8 \\ test\,9 \\ test\,10 \\ test\,11 \end{array} \left[\begin{array}{cccc} \mathbf{0} & 0 & \mathbf{1} & 0 \\ 1 & 1 & \mathbf{1} & \mathbf{1} \\ \mathbf{3} & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 1 & 1 & 0 \\ 3 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 \end{array}\right] \begin{array}{l} \textbf{fail} \\ \textbf{fail} \\ \textbf{fail} \\ pass \\ pass \\ pass \\ pass \\ pass \\ pass \\ pass \\ pass \end{array}$$

We remark here that tests 4 to 9 collectively cover all nonfaulty pairwise interactions between factors 1 and 2 except for the interactions containing $(1,3)$, and tests 10 and 11 collectively cover all nonfaulty pairwise interactions containing $(1,3)$, so the above ELA is optimal, and $ELAN(G) = 11$.

Martínez et al. [32] have proven the following relationship between ELAs and detecting arrays.

Theorem 2.2.8 *[32] Fix $d, k,$ and $t \leq k$, and $g_i \geq 2$ for $i \in [1,k]$. Let $\mathcal{H}^d_{t,(g_1,g_2,\ldots,g_k)}$ be the class of hypergraphs H of the form $H_{t,(g_1,g_2,\ldots,g_k)}$ with $|E(H)| \leq d$. Then, A is an $ELA(N; t, \mathcal{H}^d_{t,(g_1,g_2,\ldots,g_k)})$ if and only if A is a $(\overline{d}, t)$-detecting array.*

More generally, they prove that the same relationship holds if we replace t by $\overline{t}$, and restrict ourselves to the hypergraphs $H \in \mathcal{H}^d_{\overline{t},(g_1,g_2,\ldots,g_k)}$ with $E(H)$ independent (in the sense of Definition 2.1.1). They also give necessary conditions for an $ELA(N; G)$ to exist, i.e. for a graph G to be locatable. In particular, they show that, for a (multipartite) graph G to be locatable, it cannot have two vertices u, v in distinct

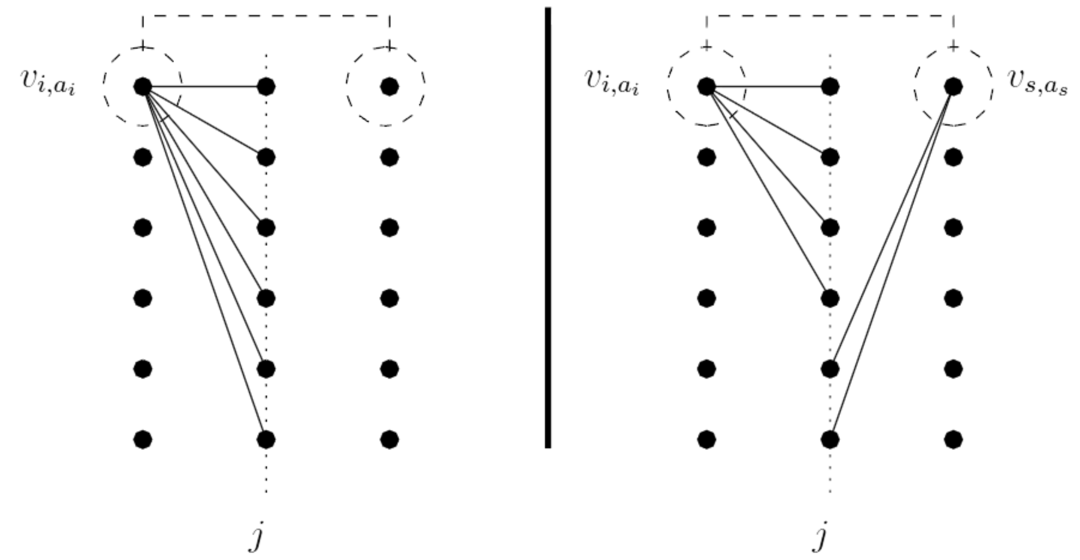


Figure 2.3: Structures which prevent the location of errors [32].

parts such that all vertices in a third part are contained in $N(u) \cup N(v)$. We see this in Figure 2.3, where dashes indicate a nonlocatable pairwise interaction in each case.

Theorem 2.2.9 *[32] Let G be a $G_{(g_1,g_2,\dots,g_k)}$ with $k \geq 3$.*

1. If there exist a vertex $v_{i,a_i} \in V(G)$ and a part (factor) $j \in [1,k] \setminus \{i\}$ such that $\{v_{i,a_i}, v_{j,x}\} \in E(G)$ for all $x \in [0, g_j - 1]$, then G is not locatable.

2. If there exist vertices $v_{i,a_i}, v_{s,a_s} \in V(G)$ with $i \neq s$ and a factor $j \in [1,k] \setminus \{i,s\}$ such that for all $x \in [0, g_j - 1]$ we have $\big\{\{v_{i,a_i}, v_{j,x}\}, \{v_{s,a_s}, v_{j,x}\}\big\} \cap E(G) \neq \emptyset$, then G is not locatable.

The conditions given in the preceding theorem can be prevented by the existence of at least one vertex in each part (factor) which is not an end of an edge. In the context of testing, the values of such vertices are called safe values, which we formally define here.

Definition 2.2.10 *Let H be an $H_{t,(g_1,g_2,...,g_k)}$ or an $H_{\bar{t},(g_1,g_2,...,g_k)}$ (associated with a $TP(k,(g_1,g_2,...,g_k))$) whose parts are $V_1, V_2, ..., V_k$ of cardinalities $g_1, g_2, ..., g_k$, respectively. Then H has* ***safe values*** *if for each $i \in [1,k]$ there exists a vertex $v_{i,s_i} \in V_i$ such that v_{i,s_i} is not contained in any hyperedge. We call the values $s_1, s_2, ..., s_k$* ***safe values*** *(for factors $1, 2, ..., k$, respectively) for H.*

Martínez et al. [32] have shown that a hypergraph is locatable if it has safe values
.

Theorem 2.2.11 *Let H be an $H_{\bar{t},(g_1,g_2,...,g_k)}$ such that $E(H)$ is independent (in the sense of Definition 2.1.1). If H has safe values, then H is $\bar{t}$-locatable (refer to Definition 2.2.2).*

Furthermore, the following theorems allow many errors to be located, based on the hypergraph structure. If a given hypergraph has safe values, then we can construct an ELA which locates up to d errors of strengths up to t via an MCA of strength $t+d$.

Theorem 2.2.12 *[32] Fix $d, k, t, g_1, g_2, ..., g_k$ such that $t+d \leq k$. Let $\mathcal{SH}$ denote the class of hypergraphs of the form $H_{t,(g_1,g_2,...,g_k)}$ which have safe values and at most d hyperedges. Let $\overline{\mathcal{SH}}$ denote the class of hypergraphs H of the form $H_{\bar{t},(g_1,g_2,...,g_k)}$ such that H has safe values, $E(H)$ is independent (in the sense of Definition 2.1.1), and $|E(H)| \leq d$. Then, every $MCA(N; t+d, k, (g_1, g_2, ..., g_k))$ is also an $ELA(N; t, \mathcal{SH})$ and an $ELA(N; \bar{t}, \overline{\mathcal{SH}})$.*

We notice that, if t is small relative to k, an $MCA(N; t+d, k, (g_1, g_2, ..., g_k))$ may be an $ELA(N; \bar{t}, \overline{\mathcal{SH}})$ even for hypergraphs $H \in \mathcal{SH}$ that have many edges (which represent equally many errors in the testing problem associated with H). More specifically, since $d \leq k-t$, we can have d grow linearly with k. For reasons that will be made apparent later, we prefer to avoid this sort of growth of d relative to k.

Pragmatically speaking, we often cannot assume that a given hypergraph has safe values. However, we can construct an ELA which locates up to d errors of strengths up to t, for a given hypergraph H, as long as H is locatable, by using an MCA of strength $t(d+1)$.

Theorem 2.2.13 *[32] Fix $d, k, t, g_1, g_2, ..., g_k$ such that $t(d+1) \leq k$. Let $\mathcal{H}$ denote the class of hypergraphs of the form $H_{t,(g_1,g_2,\dots,g_k)}$ with at most d hyperedges. Let $\overline{\mathcal{H}}$ denote the class of hypergraphs H of the form $H_{\bar{t},(g_1,g_2,\dots,g_k)}$ which have $E(H)$ independent (in the sense of Definition 2.1.1) and $|E(H)| \leq d$. Let $\mathcal{LH}$ and $\overline{\mathcal{LH}}$ be the sets of t-locatable and $\bar{t}$-locatable hypergraphs in $\mathcal{H}$ and $\overline{\mathcal{H}}$, respectively. Let A be an $MCA(N; t(d+1), k, (g_1, g_2, ..., g_k))$. Then A is also an $ELA(N; t, \mathcal{LH})$ and an $ELA(N; \bar{t}, \overline{\mathcal{LH}})$. Moreover, if $H \in \mathcal{H} \cup \overline{\mathcal{H}}$, then H is $\bar{t}$-locatable if every relevant s-way interaction (for $s \in [1,t]$) is locatable by a row (test) of A.*

We notice that here, d may also grow linearly with respect to k, for a fixed t, since $d \leq \frac{k}{t} - 1$. So, given a large enough k, an $MCA(N; t(d+1), k, (g_1, g_2, ..., g_k))$ can still locate plenty of errors. For instance, if we (reasonably) assume that all faulty interactions are of strength at most $t = 6$, then there exists an ELA for a system with 600 factors that locates up to 99 errors! The only problem is that the number of tests (i.e. rows) may be too large. Fortunately, as we observe next, if we fix d, t, and g, then the size N of an ELA grows in proportion to $\log k$.

Theorem 2.2.14 *[32] Fix g and t, and let $\mathcal{H}(t, k, d)$ be any set of hypergraphs of the form $H_{\bar{t},(g_1,g_2,\dots,g_k)}$, with at most d hyperedges, where $g_i \leq g$ for all $i \in [1, k]$, and satisfying the extra conditions given below. Then,*

1. if all hypergraphs in $\mathcal{H}(k, d)$ are $\bar{t}$-locatable and $t(d+1) \leq k$, then there exists an $ELA(N; \bar{t}, \mathcal{H}(t, k, d))$ for $N \in \mathcal{O}(dg^{td} \log k)$; and

2. if all hypergraphs in $\mathcal{H}(k, d)$ have safe values and $t + d \leq k$, then there exists an $ELA(N; \bar{t}, \mathcal{H}(t, k, d))$ for $N \in \mathcal{O}(dg^{d} \log k)$.

We notice potential problems if d grows too quickly relative to k. If we fix g and t, and we have $d = ck$ for some constant $c < 1$, then our ELA has $N \in \mathcal{O}(dg^{ctk} \log k)$ if H is locatable, and $N \in \mathcal{O}(dg^{ck} \log k)$ if H has safe values. In either case, the number of tests N is exponential in k. Since we wish to apply our ELAs to the problem of locating errors in large systems with many factors, we clearly prefer to avoid this.

Martínez et al. [32] note that such bounds apply to nonadaptive testing, where we create an ELA all at once, without allowing results of some tests to affect the choice of subsequent tests.

In the next section, we give upper bounds on the size of an ELA for fixed $g = t = 2$ for the purpose of later comparison with the size of an ELA which is adaptively generated by our algorithm in Chapter 3.

2.3 Nonadaptive Location of Errors for Binary Alphabets

In Chapter 3, we give a new algorithm which adaptively locates up to 2 errors of strengths at most 2, given a binary alphabet. We give here some upper bounds on the size of an ELA, given $g = t = 2$.

Recall that, by Theorem 2.2.12, every CA is also an ELA of lower strength, for graphs with safe values. By applying Theorem 1.2.7, we get the following upper bound.

Corollary 2.3.1 *Fix d and k so that $2 + d \leq k$. Let $\overline{\mathcal{SG}}$ be any set of graphs G of the form $H_{\overline{2},k,2}$ which have safe values, $E(G)$ independent (in the sense of Definition 2.1.1), and $|E(G)| \leq d$. Then:*

$$ELAN(\overline{\mathcal{SG}}) \leq 2^{2+d}(2+d)^{\mathcal{O}(\log d)} \log k.$$

Proof: Let A be an optimal $CA(N;2+d,k,2)$ and notice that A is also an $ELA(N;\overline{2},\overline{\mathcal{SG}})$, by Theorem 2.2.12. By Theorem 2.2.14, we have

$$N = CAN(2+d,k,2) \leq 2^{2+d}(2+d)^{\mathcal{O}(\log d)} \log k.$$

▮

Similarly, by Theorem 2.2.13, every CA is also an ELA of lower strength for locatable graphs with independent edge sets. We apply Theorem 1.2.7 again to get the following upper bound.

Corollary 2.3.2 *Fix d and k so that $2(d+1) \leq k$. Let $\mathcal{G}$ be any set of graphs G of the form $H_{\overline{2},k,2}$ such that $E(G)$ is independent (in the sense of Definition 2.1.1), and $|E(G)| \leq d$. Let $\mathcal{LG}$ be any set of locatable graphs in $\mathcal{G}$. Then:*

$$ELAN(\overline{\mathcal{LG}}) \leq 2^{2(d+1)}\big(2(d+1)\big)^{\mathcal{O}(\log d)} \log k.$$

Proof: Let A be an optimal $CA(N;2(d+1),k,2)$ and notice that A is also an $ELA(N;\overline{2},\overline{\mathcal{LG}})$, by Theorem 2.2.12. By Theorem 1.2.7, we have

$$N = CAN(2(d+1),k,2) \leq 2^{2(d+1)}\big(2(d+1)\big)^{\mathcal{O}(\log d)} \log k.$$

▮

Following Theorem 2.2.14, we get the following asymptotics.

Corollary 2.3.3 *Fix d and k. Let $\mathcal{H}(2,k,d)$ be any set of graphs of the form $H_{\overline{2},k,2}$ which satisfies the extra conditions given below, such that $E(G)$ is independent (in the sense of Definition 2.1.1) and $|E(G)| \leq d$. Then,*

1. if all graphs in $\mathcal{H}(2,k,d)$ are locatable, and $2(d+1) \leq k$, then there exists an $ELA(N;\mathcal{H}(2,k,d))$ for $N \in \mathcal{O}(d\,2^{2d}\log k)$; and

2. if all graphs in $\mathcal{H}(2,k,d)$ have safe values and $2+d \leq k$, then there exists an $ELA(N;\mathcal{H}(2,k,d))$ for $N \in \mathcal{O}(d\,2^{d}\log k)$.

Proof: This follows directly from Theorem 2.2.14, after substituting $g = t = 2$. ∎

Chapter 3

Robust Error Location for Binary Alphabets

In this chapter, we describe algorithms which can be applied only to testing problems whose errors have strengths at most two, so we refer to $\overline{2}$-locatable interactions, graphs, etc. as simply locatable, following the convention given in Definition 2.2.2.

Let G be a $G_{k,2}$ associated with a $TP(k, 2)$ whose relevant interactions are all assumed to be locatable with respect to G. Martínez et al. [32] give an algorithm called DiscoverEdges which constructs a strength-$\overline{2}$ ELA for G without knowledge of safe values. Their algorithm also identifies and returns the set of all errors in the given testing problem. DiscoverEdges begins by finding a passing test via a random selection process. When applied to a $TP(k, 2)$ with at most 2 errors, DiscoverEdges has an expected running time of at most $2\big(1 + o(1)\big)(\log k)^2 + \mathcal{O}(\log k)$ tests.

In this chapter, we give a new, completely deterministic algorithm which adaptively constructs an $ELA(N; \overline{2}, G)$, and does not require the assumption that all relevant interactions be locatable. When the sequence of tests (constructed by the algorithm) is applied to a testing problem, they collectively either identify all relevant faulty interactions, or determine that we have a certain structure of nonlocatable errors. The algorithm has a worst-case running time of at most $2\big(1+o(1)\big)(\log k)^2 + \mathcal{O}(\log k)$ tests - this is the same as the average running time of the algorithm in [32], whose worst-case running time would require more than the expected number of tests.

We begin by giving a characterization of locatable graphs with at most two edges, which represent testing problems with binary alphabets and at most two errors of strengths up to two. This characterization is a simplification of the more general characterization of locatable graphs with binary alphabets found in [32].

3.1 A Characterization of Locatable $G_{k,2}$ Graphs with $d \leq 2$ Edges

Consider graphs of the form $G_{k,2} = H_{\overline{2},k,2}$. Martínez et al. [32] characterize these graphs as either locatable or nonlocatable. They first notice that any graph of the form $G_{k,2}$ with fewer than two edges is locatable. Next, they notice that some graphs with two edges are nonlocatable, and that any nonlocatable graph with more than two edges must contain a nonlocatable subgraph with exactly two edges.

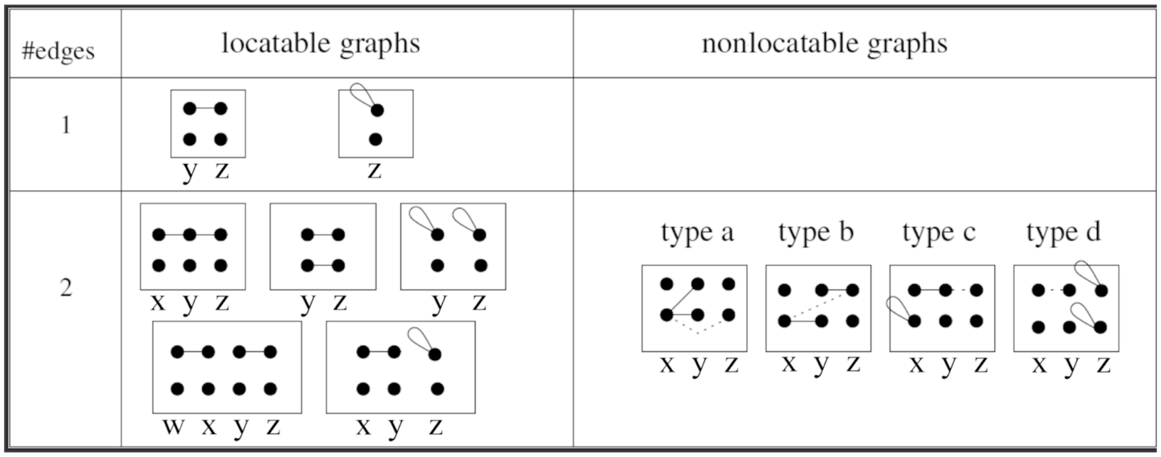


Figure 3.1: Locatable versus nonlocatable $G_{k,2}$ graphs with parts (vertical pairs of vertices) corresponding to factors w, x, y, and/or z [32].

Figure 3.1, reproduced from [32], shows four types of graph structures (called **type-a**, **type-b**, **type-c**, and **type-d**, respectively), each of which causes certain edges to be nonlocatable. Examples of nonlocatable edges are indicated by dotted lines in the above figure. Martínez et al. [32] give the following theorem.

Theorem 3.1.1 *[32] Let G be a graph of the form $G_{k,2}$ such that $E(G)$ is independent (in the sense of Definition 2.1.1). Then G is not $\overline{2}$-locatable if and only if it contains a subgraph of type-a, type-b, type-c, or type-d as given in Figure 3.1.*

We are concerned only with locating up to two errors, so we give a simpler characterization as a corollary. When we are unable to locate the errors in some nonlocatable graph G, the next-best thing would be to identify at least one vertex per pair of nonlocatable edges of G. We begin with an example which illustrates the concept of equivalence of a strength-1 error with a certain pair of strength-2 errors.

Consider a $TP(k, 2)$ whose associated error graph G (see definition 2.2.1) contains a type-a nonlocatable subgraph G', as depicted in Figure 3.1, and let $\big\{\{(x, l), (y, 0)\}, \{(x, l), (y, 1)\}\big\}$ be the set of faulty interactions corresponding to the edges of G'. Then every test T such that $T_x = l$ will fail. Now consider a second $TP(k, 2)$ whose error graph H contains a loop corresponding to the 1-way interaction $\{(x, l)\}$, and notice that every test T such that $T_x = l$ will also fail for this system. These facts lead us to the following definition.

Definition 3.1.2 *Let G_1 and G_2 be two graphs of the form $G_{k,2}$ such that $V(G_1) = V(G_2)$. Let I_1 and I_2 be the sets of failing interactions corresponding to the edge sets $E_1 = E(G_1)$ and $E_2 = E(G_2)$, respectively. Let $\mathcal{S}$ be the exhaustive test suite composed of all 2^k tests on k factors. We say that G_1 and G_2 are **location-equivalent graphs** if each test in $\mathcal{S}$ yields the same pass/fail result for G_2 as for G_1. If G_1 and G_2 are location-equivalent, we also say that E_1 and E_2 are **location-equivalent edge sets,** and that I_1 and I_2 are **location-equivalent interaction sets**.*

We also notice that certain graphs are "more nonlocatable than others" because they cause every test to fail. Furthermore, some graphs are "less nonlocatable than others" because they contain edges which can be located. We describe these graphs in the following definition.

Definition 3.1.3 *Consider a $TP(k,2)$ applied to a system whose error graph G is of the form $G_{k,2}$. We call G* ***strongly nonlocatable*** *if it is location-equivalent to a graph H such that $V(G) = V(H)$ and $K_{2,2} \subseteq H$. An interaction I $\big($and its corresponding edge or nonedge $e_I \in E(G) \cup E(\overline{G})\big)$ is* ***strongly locatable with respect to*** *G if there exists a test that locates I and covers only interactions which are locatable with respect to G.*

We get the following corollary to Theorem 3.1.1, as a consequence of the preceding two definitions.

Corollary 3.1.4 *Consider a $TP(k,2)$ applied to a system whose error graph G is of the form $G_{k,2}$, has two edges, and has a subgraph of type-a, type-b, type-c, or type-d as given in Figure 3.2.*

1. If G contains a type-a (induced) subgraph G' with edges corresponding to interactions $I_1 = \{(x,1),(y,0)\}$ and $I_2 = \{(x,1),(y,1)\}$, then the location-equivalent subgraph G'', whose only edge is a loop corresponding to interaction $I_{1,2} = \{(x,1)\}$, is locatable.

2. If G contains a type-b (induced) subgraph, then every edge of G is strongly locatable.

3. If G contains a type-c (induced) subgraph whose edges correspond to interactions $I_1 = \{(x,0),(y,0)\}$ and $I_2 = \{(x,1)\}$, then the location-equivalent subgraph (whose edges are loops corresponding to interactions $I_{1'} = \{(x,1)\}$ and $I_{2'} = \{(y,0)\}$) is locatable.

4. If G contains a type-d subgraph, then G is strongly nonlocatable.

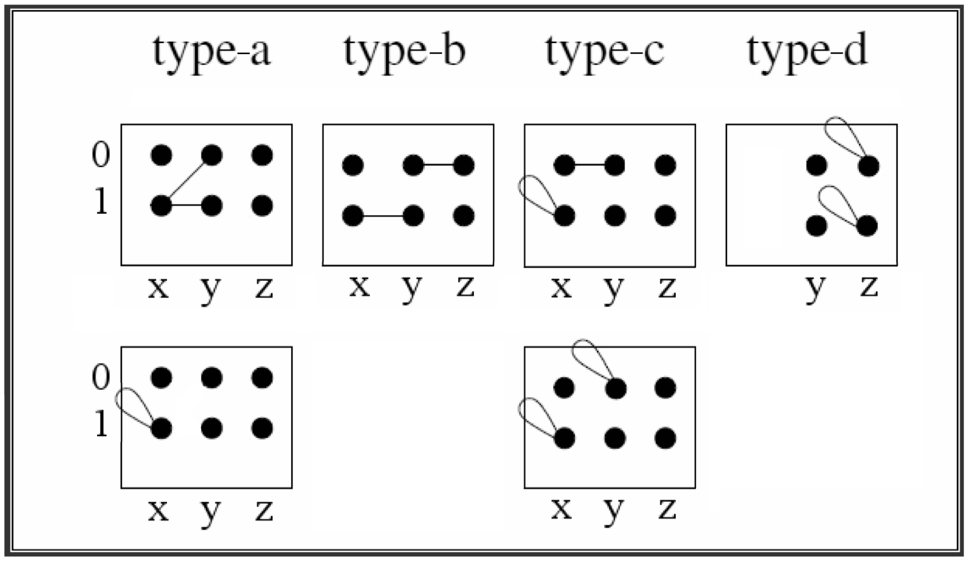


Figure 3.2: Nonlocatable $G_{k,2}$ graphs with two edges (top row), and their location-equivalents (bottom row).

Proof: Let C be a $CA(4; 2, 3, 2)$ whose rows are tests $000, 011, 101$, and 110.

1. Suppose G contains a type-a induced subgraph G'. Let G'' be as defined in the statement of this corollary. G'' has only a single edge, so it must be $\overline{2}$-locatable by [32]. By the paragraph preceding Definition 3.1.2, G' and G'' are location-equivalent. Observe that G'' has safe values, by Definition 2.2.10. Hence by Theorem 2.2.12, any $CA(N; 1 + 1, k, 2)$, including C, is an $ELA(N; 1, G'')$. Therefore G'' is locatable.

2. Suppose G contains a type-b induced subgraph G' with edges corresponding to the interactions $I_1 = \{(x, 1), (y, 1)\}$ and $I_2 = \{(y, 0), (z, 0)\}$. Then the only nonlocatable interaction with respect to G is $I_* = \{(x, 1), (z, 0)\}$. The tests 111 and 000 both avoid I_*, and they locate I_1 and I_2, respectively. Hence every edge of G is strongly locatable.

3. Suppose G contains a type-c induced subgraph whose edges represent interactions $I_1 = \{(x, 1)\}$ and $I_2 = \{(x, 0), (y, 0)\}$. First replace the loop corresponding to I_1 by a location-equivalent (nonlocatable) pair of edges whose corresponding interactions are $I_{1'} = \{(x, 1), (y, 0)\}$ and $I_{1''} = \{(x, 1), (y, 1)\}$. Then notice that there are

now two nonlocatable pairs of interactions, $\{I_{1'}, I_{1''}\}$ and $\{I_{1'}, I_2\}$, and replace each corresponding nonlocatable pair of edges by a location-equivalent loop. Denote by G' the resulting graph whose edges are loops which represent interactions $\{(x,1)\}$ and $\{(y,0)\}$, and notice that G' is location-equivalent to G. Let C' be an array containing each row in C, plus rows 100 and 010. Then C' is an $ELA(6;1,G')$.

4. Suppose G contains a type-d subgraph. Then every test will fail, so the graph is location-equivalent to $K_{2,2}$. Therefore, it is strongly nonlocatable. ∎

In summary, the preceding corollary says that if G is a graph with at most two edges and no type-d induced subgraph, then we can locate the edges of either G or G', where G' is location-equivalent to G. In the next section we give an algorithm which, when applied to a $TP(k,2)$ whose associated error graph is G, generates a set of tests for the given system, one of which is guaranteed to pass.

3.2 Finding a Passing Test, Binary Alphabet

In this section, we give an algorithm which finds a passing test for any $TP(k,2)$ whose associated graph G has at most two edges, and contains no type-d subgraph. The purpose is to simplify this chapter's main algorithm, which is given in Section 3.3. We begin with a definition.

Definition 3.2.1 *Consider a $TP(k,2)$ with $k \geq 2$, let t be an integer such that $t \leq k$, and let $\boldsymbol{s} = s_1 s_2 ... s_t$ be a string. An **$\boldsymbol{s}$-error** is a strength-t faulty interaction $I = \{(i_1, s_1), (i_2, s_2), ..., (i_t, s_t)\}$ for some factors $i_1, i_2, ..., i_t$. We say that the **type** of error I is $\boldsymbol{s}$, and that an element (i_j, s_j) of I is an **end** of the $\boldsymbol{s}$-error I, or simply an s_j-**end** of I.*

Notice that for a $TP(k,2)$ whose errors are of strength at most two we can have 0-errors, 00-errors, 1-errors, 11-errors, and 01-errors (which can also be called 10-errors). If we have a passing test T, we can relabel the (factor, value) pairs in our testing problem so that $T = 0^k$. As a result of this relabeling, the only possible error types are 1, 11, and 01. This chapter's main algorithm is a case-by-case analysis of a $TP(k,2)$ with at most two errors of strengths up to two. The algorithm depends on error types present in the given system, and it is greatly simplified by the fact that there are only three types of errors possible (once a passing test is found).

We now present the following theorem which guarantees the existence of a passing test for a $TP(k,2)$ if and only if its associated error graph has at most two edges of strengths up to two and contains no type-d subgraph.

Theorem 3.2.2 *Consider a $TP(k,2)$ whose associated error graph G has at most 2 edges, and $E(G)$ is independent (in the sense of Definition 2.1.1). There exists a passing test for the $TP(k,2)$ if and only if G contains no type-d subgraph.*

Proof: First, suppose that G contains a type-d subgraph. Then G is strongly nonlocatable, by Corollary 3.1.4. Therefore, no passing test exists for the system associated with G, by Theorem 3.1.1.

Conversely, suppose that G contains no type-d subgraph. Then, by Theorem 3.1.1, it is either locatable or it contains a (nonlocatable) subgraph of type-a, type-b, or type-c. If G is locatable, then a passing test exists for the system associated with G, as a consequence of being locatable (see Definition 2.2.2). Otherwise, we may assume that G contains a subgraph of type-a, type-b, or type-c, such that the vertices of G have been relabeled to match Figure 3.2. In this case, we may also assume that $k \geq 3$.

Let T be a test for our given testing problem such that $S = S_x S_y S_z = 010$ is a subtest of T. Notice that S does not cover any edge depicted in a graph of type-a, type-b, or type-c in Figure 3.2. Therefore T is a passing test. ∎

Now, consider a pair $[T, T'] = [0^k, 1^k]$ of tests. If either test passes, then we have a passing test. Otherwise, we have a disjoint pair of failing tests (see Definition 1.1.1). We notice below that, in general, no interaction can be simultaneously covered by a pair of disjoint tests (or even a pair of disjoint subtests).

Lemma 3.2.3 *Consider a testing problem for which $[T, T']$ is a pair of disjoint tests. No interaction can be covered by both T and T'. Furthermore, if T and T' are both failing tests, then each covers a distinct faulty interaction.*

Proof: This follows directly from Definition 1.1.1. ∎

Since we restrict ourselves to systems with binary alphabets, tests which are disjoint are also complementary in the following way.

Definition 3.2.4 *The* ***complement of the binary character*** *$x \in [0, 1]$ is $\overline{x} = (x + 1)(mod 2)$.*

Let $T = T_1T_2...T_k$ be a test applied to a $TP(k, 2)$ (hence $T_i \in [0, 1]$ for all $i \in [1, k]$). The ***complement of the test*** *T is $\overline{T} = \overline{T_1}\,\overline{T_2}...\overline{T_k}$. We call a pair of tests $p = [T, \overline{T}]$* ***complementary****, and we say that p is a* ***failing pair*** *of tests if T and $\overline{T}$ both fail.*

We exploit Lemma 3.2.3 and Definition 3.2.4 in the following way. The complementary pair of tests $p = [0^k, 1^k]$ determines the error types for a $TP(k, 2)$ with $d \leq 2$ errors.

Lemma 3.2.5 *Consider a $TP(k, 2)$ with at most two errors of strengths up to 2, and let $p = [0^k, 1^k]$ be a failing pair of tests. Then there is a unique error which causes 0^k to fail, and it is either a 0-error or a 00-error. Similarly, there is a unique error which causes 1^k to fail, and it is either a 1-error or a 11-error.*

Proof: Since both 0^k and 1^k fail, and we have at most two errors of strengths at most 2, it follows that only a 0-error or a 00-error could cause the test 0^k to fail. Similarly, only a 1-error or a 11-error could cause 1^k to fail. ∎

We introduce here some terminology regarding a set of factors associated with either an error or part of an error. When we focus on finding an error that has at least some of its factors in the set $D \subseteq [1, k]$, we call D an **inspection set**. We also define here what it means for an error to be either within or across an inspection set.

Definition 3.2.6 *Consider a $TP(k, 2)$ whose associated error graph G has an independent edge set (in the sense of Definition 2.1.1), and let $e = \{(f_1, v_1), (f_2, v_2), ..., (f_t, v_t)\}$ be an error (see Definition 1.1.4) whose associated factor set is $E = \{f_1, f_2, ..., f_t\}$. We say that e (and its associated edge in G) is **within** a set $D \subseteq [1, k]$ if $E \subseteq D$. Let $B_1, B_2, ..., B_t$ be disjoint sets such that $B_1 \cup B_2 \cup ... \cup B_t = D$. We say that $[B_1, B_2, ..., B_t]$ is a **partition** of D, and we call $B_1, B_2, ..., B_t$ the **parts** of D. An error e (and its associated edge in G) is **across** the partition if $E \cap B_i \neq \emptyset$ for at least two values of $i \in [1, t]$.*

Notice that we can easily transform a system with k factors into a system with $k' > k$ factors by adding $k' - k$ "dummy" factors which do not affect the results of tests. For the rest of this chapter, we assume without loss of generality that $k = 2^z$ for some integer $z \geq 1$ since this assumption greatly simplifies our analysis.

Now, suppose that 0^k and 1^k are failing tests. Then we have two errors, and Lemma 3.2.5 gives some information about them. As we shall see below, a sequence of pairs of complementary failing tests gives us precise information about the inspection sets containing the factors associated with the errors.

Theorem 3.2.7 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ with at most two errors of strengths up to 2. Let j be an integer such that $1 \leq j \leq z + 1$, and for every integer $i \in [1, j]$ let $m_i = 2^{i-2}$. Define a sequence of pairs $p_1, p_2, ..., p_j$ of tests as follows: $p_i = [T(i), \overline{T(i)}]$ such that* $T(i) = \begin{cases} 0^k & \text{if } i = 1 \\ (0^{k/2m_i}1^{k/2m_i})^{m_i} & \text{if } i > 1 \end{cases}$

If all pairs of tests $p_1, p_2, ..., p_j$ fail, then there are two errors in our $TP(k, 2)$, each within the same inspection set D of cardinality $|D| = k/2m_j$, such that the subtest $T(j)_D = 0^{k/2m_j}$ or $T(j)_D = 1^{k/2m_j}$ (see Definition 1.1.1).

Proof: We use induction on j, with the restriction that $j \in [1, z + 1]$ since no inspection set can have cardinality less than 1. By Lemma 3.2.5, there are two errors: an error e of type 0 or 00, and another error e' of type 1 or 11.

First, consider the base case with $j = 1$. The errors are clearly within the same inspection set of size $k/2m_1 = k$.

Suppose that, for some integer $j \in [1, z]$, if all pairs of tests $p_1, p_2, ..., p_j$ fail, then there are two errors in our $TP(k, 2)$, each within the same inspection set D of cardinality $|D| = k/2m_j$, such that the subtest $T(j)_D = 0^{k/2m_j}$ or $T(j)_D = 1^{k/2m_j}$.

Consider the failing pair of tests $p_{j+1} = [T(j+1), \overline{T(j+1)}]$ such that $T(j+1) = (0^{k/4m_j}1^{k/4m_j})^{2m_j}$. Suppose without loss of generality that e causes $T(j)$ to fail, and e' causes $\overline{T(j)}$ to fail. Then:

$$T(j)_D = 0^{k/2m_j}.$$

$$\overline{T(j)}_D = 1^{k/2m_j}.$$

By Lemma 3.2.3, e must also cause one of $T(j + 1), \overline{T(j + 1)}$ to fail. Suppose without loss of generality that e causes $T(j + 1)$ to fail, and notice that:

$$T(j + 1)_D = 0^{k/4m_j}1^{k/4m_j}.$$

$$\overline{T(j + 1)}_D = 1^{k/4m_j}0^{k/4m_j}.$$

Let $[B, C]$ be a partition of D such that $T(j+1)_B = 0^{k/4m_j}$ and $T(j+1)_C = 1^{k/4m_j}$. Clearly e must be within B. Furthermore, $\overline{T(j+1)}$ is a failing test, so it must cover e', by Lemma 3.2.3. Test $\overline{T(j+1)}$ has value 1 for all factors in B and value 0 for all factors in C, therefore e' must be within B. Hence, both errors are within the inspection set B of cardinality $k/4m_j = k/2m_{(j+1)-2}$, and $T(j+1)_B = 0^{k/2m_{(j+1)-2}}$. ▮

Next we notice that if the sequence of failing pairs of tests is long enough, then both errors must be within a very small inspection set. For the following corollary, if both errors are of strength 2, then we know the factors corresponding to the errors once we find the inspection set.

Corollary 3.2.8 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ with two errors of strengths at most 2. Suppose that pairs of tests $p_1, p_2, ..., p_z$ (as defined in Theorem 3.2.7) all fail. Then both errors are within the same inspection set $D = \{i, i+1\}$ of cardinality 2, where $i \in [1, k-1]$ is an odd integer.*

Proof: This follows directly from Theorem 3.2.7 since $k/2m_z = k/2^{\log_2 k - 1} = 2$. ▮

Next we notice that if we maximize the number of pairs of failing tests then we get either a pair of strength-1 errors corresponding to a type-d (strongly nonlocatable) graph or a passing test.

Corollary 3.2.9 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose error graph G has two edges representing errors of strengths at most 2. Suppose pairs of tests $p_1, p_2, ..., p_z$ (as defined in Theorem 3.2.7) all fail. If the pair of tests $p_{z+1} = [(01)^{k/2}, (10)^{k/2}]$ fails, then the errors correspond to loops in the same part of G, and G contains a type-d strongly nonlocatable subgraph. Otherwise, one of $(01)^{k/2}, (10)^{k/2}$ is a passing test.*

Proof: Following Theorem 3.2.7, if p_{z+1} fails, then we have two errors within the same inspection set D of cardinality $k/2m = k/2^z = 1$, so let $D = \{d\}$. Following Lemma 3.2.5, D contains a 0-error and a 1-error, both corresponding to the same factor d, and they are represented in the graph G by a pair of loops corresponding to a type-d subgraph. ∎

Algorithm 3.1 on Page 51 contains two versions of a method for finding a passing test, and we refer to these versions of FindPassingTest as **Version 1** and **Version 2** on Page 51. We first show that Version 1 correctly returns *null* if it is given a testing problem whose associated error graph contains a (strongly nonlocatable) type-d subgraph, and returns a passing test otherwise.

Theorem 3.2.10 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose associated error graph G has $d \leq 2$ edges. Then Version 1 of* FindPassingTest *(see Algorithm 3.1) returns null if G contains a type-d (strongly nonlocatable) subgraph, and it returns a passing test otherwise.*

Proof: This follows directly from Theorem 3.2.7 and its corollaries. ∎

We also give a second, more general version of FindPassingTest which detects the presence of errors corresponding to a type-a nonlocatable subgraph, under certain assumptions relevant to the algorithm presented in the next section. We show here the correctness of FindPassingTest, Version 2.

Theorem 3.2.11 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose associated error graph G has $d \leq 2$ edges. Suppose that 0^k is a passing test, and at most one interaction I^* of type 1 or 10 (see Definition 3.2.1) is faulty. Let $I^* = \{(a, 1)\}$ if it is a 1-error, otherwise let $I^* = \{(a, 1), (b, 0)\}$, where $a, b \in [1, k]$.*

Construct a $TP(k+1,2)$ from our $TP(k,2)$ by introducing a $(k+1)$th dummy factor whose corresponding alphabet is $\{0\}$, such that $\{(k+1,0)\} \not\subseteq I$ for any faulty interaction I in the $TP(k+1,2)$. Let $X = [k+1] \setminus \{a\}$. If we fix $T'_a = 1$ for all tests T' conducted by Version 2 of FindPassingTest, *then it returns null if $\{(a,1)\}$ corresponds to a loop or a degree-2 vertex in a type-a nonlocatable subgraph of G, or a passing test otherwise.*

Proof: A passing test (0^k) exists, therefore G does not contain a type-d strongly nonlocatable subgraph. Notice that each test generated by the algorithm covers $\{(a,1)\}$. By Theorem 3.2.7 applied to the set X, if the algorithm does not find a new passing test $\big($covering $\{(a,1)\}\big)$, then we have two errors in X, each within the same inspection set $\{x\}$ (such that $x \in X$) of cardinality 1. These errors either correspond to a pair of loops in a type-d subgraph of G, or else the seemingly faulty interactions $\{(x,0)\}$ and $\{(x,1)\}$ are each an end of a distinct strength-2 error such that every test generated by the algorithm covers at least one of those strength-2 errors. The former possibility contradicts the fact that a passing test exists.

The latter is only satisfied by the pair of errors$\big\{\{(a,1),(x,0)\},\ \{(a,1),(x,1)\}\big\}$, therefore we have either a type-a nonlocatable subgraph G' whose edges correspond to the set of errors $\big\{\{(a,1),(x,0)\},\ \{(a,1),(x,1)\}\big\}$, or a location-equivalent subgraph containing a loop corresponding to interaction $\{(a,1)\}$ (see Definition 3.1.2). ∎

Algorithm 3.1 Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose associated error graph G has at most two edges (and $E(G)$ is independent, in the sense of Definition 2.1.1). Applied to a $TP(k, 2)$, this algorithm checks complementary pairs of tests according to Theorem 3.2.7.

```
▷ Version 1: By Theorem 3.2.10, this returns $null$ if $G$ contains a strongly
▷ nonlocatable type-d subgraph and it returns a passing test otherwise.
procedure FindPassingTest($k$)
    $j \leftarrow 1$
    $m \leftarrow 1/2$
    ▷ Check pairs of tests from Theorem 3.2.7.
    while $j \leq \log_2 k + 1$ do
        $T \leftarrow \begin{cases} 0^k & \text{if } j = 1 \\ (0^{k/2m}1^{k/2m})^m & \text{otherwise} \end{cases}$
        for $S \in \{T, \overline{T}\}$ do
            if Test($S$) = pass then return $S$
        $j \leftarrow j + 1$
        $m \leftarrow 2m$
    return $null$

▷ Version 2: Suppose our system has a set $X$ of $k = 2^z$ factors, one of which is a
▷ dummy factor, plus a factor $a \notin X$ such that all tests run by this algorithm fix
▷ $T'_a = 1$. Under the assumptions of Theorem 3.2.11, this algorithm returns $null$ if
▷ $\{(a, 1)\}$ corresponds to either a loop or a degree-2 vertex in a type-a nonlocatable
▷ subgraph, and it returns a passing test otherwise.
procedure FindPassingTest($X, k$)
    Let $X = \{x_1, x_2, ..., x_k\}$ such that $x_1 \leq x_2 \leq ... \leq x_k$. Rearrange the $k + 1$
    factors so that $x_i = i$ for all $i \in [1, k]$, making $a$ the $(k + 1)$st factor. This
    allows compact notation for each test $T'$ conducted on our $TP(k + 1, 2)$.
    $j \leftarrow 1$
    $m \leftarrow 1/2$
    while $j \leq \log_2 k + 1$ do
        $T \leftarrow \begin{cases} 0^k & \text{if } j = 1 \\ (0^{k/2m}1^{k/2m})^m & \text{otherwise} \end{cases}$
        for $T' \in \{T1, \overline{T}1\}$ do
            if Test($T'$) = pass then
                Reverse the earlier rearrangement of the $k + 1$ factors.
                return $T'$
        $j \leftarrow j + 1$
        $m \leftarrow 2m$
    Reverse the earlier rearrangement of the $k + 1$ factors.
    return $null$
```

Next, we give the worst-case running time of both versions of Algorithm 3.1.

Lemma 3.2.12 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose associated error graph G has at most two edges. Then each version of* FindPassingTest *given in Algorithm 3.1 executes at most $2(\log_2 k + 1)$ tests.*

Proof: Each version of FindPassingTest clearly conducts only the tests within its while loop: there are two complementary tests in each iteration, and at most $\log_2 k + 1$ iterations. ▮

In the next section, we give an algorithm which applies FindPassingTest to determine whether errors corresponding to a nonlocatable subgraph are present in a given $TP(k, 2)$.

3.3 Strength-$\overline{2}$ Error Location for $G_{k,2}$ Graphs with At Most Two Edges

In this section, we give an algorithm called LocateAllErrors which does the following when applied to a $TP(k, 2)$ whose associated error graph G has up to 2 edges. If G contains a type-d (strongly nonlocatable) subgraph, then LocateAllErrors (given as Algorithm 3.3 on Page 64) exits after printing a warning message regarding the presence of a strongly nonlocatable type-d subgraph (which would cause every test to fail). Otherwise, the algorithm adaptively builds a strength-$\overline{2}$ ELA for either G or a location-equivalent graph G', and it returns a set E of all errors corresponding to the edges in either G or G' (see Definition 3.1.2).

Recall that Martínez et al. also give an algorithm that constructs a strength-$\overline{2}$ *ELA* for G without knowledge of safe values. Their algorithm, DiscoverEdges (see [32]), begins by finding a passing test via a random selection process, and subsequently identifies and returns the set of all errors in the given testing problem, under the assumption that all relevant interactions are locatable. Within their algorithm, Martínez et al. give an auxiliary binary search procedure called SEARCHENDPOINT, which they use to efficiently identify the edges of G, one endpoint at a time. We reproduce their procedure, adapted to our notation, as Algorithm 3.2.

Algorithm 3.2 [32] Under the assumptions of Lemma 3.3.1, this auxiliary procedure returns the set of factors in D corresponding to the errors either covered by the failing test T or covered by neither T nor $\overline{T}$.

```
procedure SEARCHENDPOINT($T, D$)
    if $|D| = 1$ then return $D$
    else
        ▷ Complement opposing halves of $D$ as follows:
        Partition $D$ into $[B, C]$ such that $|C| \leq |B| \leq |C| + 1$.
        $V'' \leftarrow V' \leftarrow \emptyset$
        Define $T'$ by: $T'_f \leftarrow \begin{cases} \overline{T_f} & \text{if } f \in C \\ T_f & \text{otherwise} \end{cases}$
        Define $T''$ by: $T''_f \leftarrow \begin{cases} \overline{T_f} & \text{if } f \in B \\ T_f & \text{otherwise} \end{cases}$
        if TEST($T'$) = fail then $V' \leftarrow$ SEARCHENDPOINT($T', B$)
        if TEST($T''$) = fail then $V'' \leftarrow$ SEARCHENDPOINT($T'', C$)
    return $V' \cup V''$
```

Let T be a failing test, and let $[D, D']$ be a partition of $[1, k]$. In the following lemma, we give conditions on the use of SEARCHENDPOINT(T, D), and we show that it identifies and returns the set of factors in D which correspond to strength-1 errors covered by T_D, strength-2 errors across $[D, D']$ and covered by T, or strength-2 errors within D and covered by neither T nor $\overline{T}$ (i.e. errors of the form $\{(d_1, v_1), (d_2, v_2)\}$, where $T_{d_1} = v_1$, $\overline{T_{d_2}} = v_2$, and $d_1, d_2 \in D$.

Lemma 3.3.1 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k,2)$ whose associated error graph G has up to 2 edges representing relevant errors of strengths at most 2. Let $T = T_1 T_2 \ldots T_k$ be a failing test for G, and let $[D, D']$ be a partition of $[1,k]$ such that the following three conditions are satisfied:*

1. No errors are covered by $T_{D'}$, i.e. $\{v_{f',T_{f'}}, v_{g',T_{g'}}\} \notin E(G)$ for all $f', g' \in D'$.

2. No errors are covered by $\overline{T_D}$, i.e. $\{v_{f,\overline{T_f}}, v_{g,\overline{T_g}}\} \notin E(G)$ for all $f, g \in D$.

3. No errors across $[D, D']$ are covered by the test $S = \overline{T_D} T_{D'}$, i.e. $\{v_{f,\overline{T_f}}, v_{f',T_{f'}}\} \notin E(G)$ for all $f \in D$, $f' \in D'$.

Let R be the set of factors $f \in [1,k]$ that are returned by SearchEndPoint(T, D) *(see Algorithm 3.2). Then:*

(a) If there exist $f \in D$, $f' \in D'$ such that $\{v_{f,T_f}, v_{f',T_{f'}}\} \in E(G)$, then $f \in R$.

(b) If there exists $l \in D$ such that $\{v_{l,T_l}, v_{l,T_l}\} \in E(G)$, then $l \in R$.

(c) If $f \in R$, then there exists $g \in [1,k]$ such that $\{v_{f,T_f}, v_{g,T_g^*}\} \in E(G)$, where

$$T_g^* = \begin{cases} T_g & \text{if } g \in D' \text{ or } g = f \in D \\ \overline{T_g} & \text{if } g \in D \text{ and } g \neq f \end{cases}$$

Proof: We begin by proving the following claim:

$\underline{\mathcal{C}:}$ *Suppose that T and $[D, D']$ satisfy conditions 1 - 3. Let $[B, C]$ be a partition of D, and let T' be the corresponding test defined in* SearchEndPoint(T, D)*. If $[B, B']$ is a partition of B, then T' and $[B, B']$ satisfy conditions 1 - 3.*

We first show how T' and $[B, B']$ satisfy condition 1: no errors are covered by $T'_{B'}$. Let $g', h' \in B'$, and note that $[C, D']$ is a partition of B'. If $g', h' \in C \subset D$, then by condition 2 of this lemma with respect to T and $[D, D']$, we have

$$\{v_{g',T'_{g'}}, v_{h',T'_{h'}}\} = \{v_{g',\overline{T_{g'}}}, v_{h',\overline{T_{h'}}}\} \notin E(G).$$

If $g', h' \in D'$, then by condition 1 of this lemma with respect to T and $[D, D']$, we have

$$\{v_{g',T'_{g'}}, v_{h',T'_{h'}}\} = \{v_{g',T_{g'}}, v_{h',T_{h'}}\} \notin E(G).$$

If $g' \in C \subset D$ and $h' \in D'$, then by condition 3 of this lemma with respect to T and $[D, D']$, we have

$$\{v_{g',T'_{g'}}, v_{h',T'_{h'}}\} = \{v_{g',\overline{T_{g'}}}, v_{h',T_{h'}}\} \notin E(G).$$

Hence T' and $[B, B']$ satisfy condition 1.

Next, we show that T' and $[B, B']$ satisfy condition 2: no errors are covered by $\overline{T'_B}$. Notice that $T'_B = T_B$, $B \subset D$, and T_B is a subtest of T_D. Therefore $\overline{T'_B}$ is a subtest of $\overline{T_D}$, which covers no errors, by condition 2 of this lemma with respect to T and $[D, D']$.

Finally, we show that T' and $[B, B']$ satisfy condition 3: no errors across $[D, D']$ are covered by the test $S = \overline{T_D}T_{D'}$, i.e. $\{v_{b,\overline{T'_b}}, v_{b',T'_{b'}}\} \notin E(G)$ for all $b \in B$, $b' \in B'$. Take any $b \in B \subset D$ and $b' \in B'$. Note that either $b' \in C$ or $b' \in D'$, since $[C, D']$ is a partition of B'. If $b' \in C \subset D$, then by condition 2 of this lemma with respect to T and $[D, D']$, we have

$$\{v_{b,\overline{T'_b}}, v_{b',T'_{b'}}\} = \{v_{b,\overline{T_b}}, v_{b',\overline{T_{b'}}}\} \notin E(G).$$

If $b' \in D'$, then by condition 3 (with respect to T and $[D, D']$), we have

$$\{v_{b,\overline{T'_b}}, v_{b',T'_{b'}}\} = \{v_{b,\overline{T_b}}, v_{b',T_{b'}}\} \notin E(G).$$

Hence T' and $[B, B']$ satisfy condition 3 as well.

Now that we have proven our claim $\mathcal{C}$, we prove that each of (a), (b), and (c) holds for T and $[D, D']$, when conditions 1 - 3 are satisfied by T and $[D, D']$. In each case, we proceed by induction on $|D|$.

To prove (a): Suppose that $|D| = 1$, and T and $[D, D']$ satisfy conditions 1 - 3. Take any $f \in D, f' \in D'$ such that $\{v_{f,T_f}, v_{f',T_{f'}}\} \in E(G)$. We see that the algorithm returns $R = D = \{f\}$ since $|D| = 1$. Hence (a) holds when $|D| = 1$.

Assume that for some integer $w \geq 2$, if $|D| < w$ and T and $[D, D']$ satisfy conditions 1 - 3, then (a) holds.

Suppose that $|D| = w$, and T and $[D, D']$ satisfy conditions 1 - 3. Take any $f \in D, f' \in D'$ such that $\{v_{f,T_f}, v_{f',T_{f'}}\} \in E(G)$. Let B and C be as defined in SEARCHENDPOINT(T, D). Notice that $f \in B$ or $f \in C$ since $[B, C]$ is a partition of D. Without loss of generality, assume that $f \in B$ (since the case where $f \in C$ is symmetrical), and let $B' = [1, k] \setminus B$. Since $f \in B \subset D$ and $f' \in D'$, the test T' defined in SEARCHENDPOINT(T, D) is a failing test. Let R' be the set returned by SEARCHENDPOINT(T', B).

By our claim $\mathcal{C}$ above, T' and $[B, B']$ satisfy conditions 1 - 3. Then (a) holds for T' and $[B, B']$, by the induction hypothesis, since $|B| < w$. Therefore $f \in R'$. It is easy to see (from the algorithm) that $R' \subseteq R$. We conclude that $f \in R$; that is, statement (a) holds for T and $[D, D']$.

To prove (b): Suppose that $|D| = 1$, and T and $[D, D']$ satisfy conditions 1 - 3. Take any $l \in D$ such that $\{v_{l,T_l}, v_{l,T_l}\} \in E(G)$. Since $|D| = 1$, the algorithm returns $R = D = \{l\}$.

Assume that for some integer $w \geq 2$, if $|D| < w$ and T and $[D, D']$ satisfy conditions 1 - 3, then (b) holds.

Suppose that $|D| = w$, and T and $[D, D']$ satisfy conditions 1 - 3. Take any $l \in D$ such that $\{v_{l,T_l}, v_{l,T_l}\} \in E(G)$. Then either $l \in B$ or $l \in C$; again, assume that $l \in B$, without loss of generality. Let $B' = [1, k] \setminus B$. Since $l \in B \subset D$, the test T' defined in SEARCHENDPOINT is a failing test. Let R' be the set returned by SEARCHENDPOINT(T', B).

By our claim $\mathcal{C}$ above, T' and $[B, B']$ satisfy conditions 1 - 3. Then (b) holds for T' and $[B, B']$, by the induction hypothesis, since $|B| < w$. Therefore $l \in R' \subseteq R$;

that is, statement (b) holds for T and $[D, D']$.

To prove (c): Suppose that $|D| = 1$, and T and $[D, D']$ satisfy conditions 1 - 3. Take any $f \in R$. We see that the algorithm returns $R = D = \{f\}$ since $|D| = 1$. Since T is a failing test and condition 1 holds, any error covered by T must be either a strength-1 error within D or a strength-2 error across $[D, D']$. That is, there exists $g \in [1, k]$ such that $\{v_{f,T_f}, v_{g,T_g}\} \in E(G)$ for either $g = f \in D$ or $g \in D'$. Therefore (c) holds for $|D| = 1$.

Now, assume that for some integer $w \geq 2$, if $|D| < w$ and T and $[D, D']$ satisfy conditions 1 - 3, then (c) holds.

Suppose that $|D| = w$, and T and $[D, D']$ satisfy conditions 1 - 3. Take any $f \in R$ (as returned by Algorithm 3.2). Consider the sets B, C, V', and V'', and the tests T' and T'' as they are computed in SearchEndPoint(T, D). It is not difficult to see (e.g. by induction) that $V' \subset B$ and $V'' \subset C$, and of course, $f \in V'$ or $f \in V''$. Without loss of generality, assume that $f \in V'$ (since the case where $f \in V''$ is symmetrical). Then T' is a failing test, and SearchEndPoint(T', B) returns f.

By our claim $\mathcal{C}$ above, T' and $[B, B']$ satisfy conditions 1 - 3. Then (c) holds for T' and $[B, B']$, by the induction hypothesis, since $|B| < w$. Therefore there exists $g \in [1, k]$ such that $\{v_{f,T'_f}, v_{g,T'^*_g}\} \in E(G)$, where

$$T'^*_g = \begin{cases} T'_g & \text{if } g \in B' \text{ or } g = f \in B \\ \overline{T'_g} & \text{if } g \in B \text{ and } g \neq f \end{cases}$$

If $g \in B'$, then $T'^*_g = T'_g$, and either $g \in C \subset D$ or $g \in D'$, since $[C, D']$ is a partition of B'. In the former case, we have

$$\{v_{f,T_f}, v_{g,\overline{T_g}}\} = \{v_{f,T'_f}, v_{g,T'_g}\} = \{v_{f,T'_f}, v_{g,T'^*_g}\} \in E(G), \text{ where } f, g \in D \text{ and } g \neq f.$$

In the latter case, we have

$$\{v_{f,T_f}, v_{g,T_g}\} = \{v_{f,T'_f}, v_{g,T'_g}\} = \{v_{f,T'_f}, v_{g,T'^*_g}\} \in E(G), \text{ where } g \in D'.$$

If $g \notin B'$, suppose first that $g = f \in B \subset D$, so $T'^*_g = T'_g$ and

$$\{v_{f,T_f}, v_{f,T_f}\} = \{v_{f,T'_f}, v_{f,T'_f}\} = \{v_{f,T'_f}, v_{f,T'^*_f}\} \in E(G).$$

Now, suppose that $g \in B$ and $g \neq f$. Then $T^*_g = \overline{T'_g}$ and

$$\{v_{f,T_f}, v_{g,\overline{T_g}}\} = \{v_{f,T'_f}, v_{g,\overline{T'_g}}\} = \{v_{f,T'_f}, v_{g,T'^*_g}\} \in E(G), \text{ where } g \in D \text{ and } g \neq f.$$

Hence there exists $g \in [1,k]$ such that $\{v_{f,T_f}, v_{g,T_g}\} \in E(G)$ if $g \in D'$ or $g = f \in D$, or $\{v_{f,T_f}, v_{g,\overline{T_g}}\} \in E(G)$ if $g \in D, g \neq f$; that is, statement (c) holds for T and $[D, D']$. ▮

We now summarize the three main steps of this chapter's primary algorithm, LocateAllErrors (given as Algorithm 3.3), which we apply to a $TP(k,2)$ whose error graph is G.

Step 0: This step improves upon Step 0 of DiscoverEdges given in [32], which assumes that all errors are locatable, and finds a passing test by a random selection process. We run Version 1 of the procedure FindPassingTest, which either finds a passing test, or determines that G contains a type-d nonlocatable subgraph (see Theorem 3.2.10). If FindPassingTest successfully finds a passing test P, we relabel P as 0^k so that any error in our given $TP(k,2)$ is either a 1-error (corresponding to a loop in G), a 10-error, or a 11-error. If this procedure does not find a passing test, then it returns a warning message about the presence of a strongly nonlocatable type-d subgraph.

Two of the three error types mentioned above may be easily mixed-up, so we introduce the following convention to prevent confusion between 1-errors and 11-errors.

Definition 3.3.2 *Suppose we have applied Algorithm 3.3 to a $TP(k,2)$, and a passing test P was successfully found in Step 0 (and the factor values were relabeled so that $P = 0^k$). Let $f \in [1,k]$. In this context, we say that a 1-error $\{(f,1)\}$ (if it exists) is a **loop at factor** f.*

Step 1: This is the same as Step 1 of DiscoverEdges given in [32]. We assume that 0^k is a passing test, and we find the set A of all loops and 1-ends of 10-errors. More precisely, we determine the set

$$A = \left\{ f \in [1,k] : \{v_{f,1}, v_{f,1}\} \in E(G) \text{ or } \exists j \in [1,k] \text{ such that } \{v_{f,1}, v_{j,0}\} \in E(G) \right\}$$

.

Step 2: We have three possible error-types: loops, 10-errors, and 11-errors. This step determines whether loops or 11-errors are present in our $TP(k,2)$ by checking the result of the test 1^k. Any loop or 11-error would cause this test to fail, therefore if 1^k passes, then our testing problem has only 10-errors. We refer to this situation as **Case 0** of Algorithm 3.3. Otherwise, there is at least one error which is either a loop or a 11-error. We refer to this as **Case 1** of Algorithm 3.3.

We sometimes need to conduct a test which does both of the following:

1. Avoids all interactions containing $\{(a,1)\}$ for some known factor $a \in [1,k]$.
2. For all factors $f \in [1,k] \setminus \{a\}$, covers the interaction $\{(f,1)\}$.

For this reason, we introduce the following notation.

Definition 3.3.3 *In the context of a $TP(k,2)$, the test $T^{(a)}$ is defined by*

$$T_f^{(a)} = \begin{cases} 0 & \textit{if } f = a \\ 1 & \textit{otherwise} \end{cases}$$

Occasionally, we also need to use its complement, $\overline{T^{(a)}}$, which covers $\{(a,1)\}$ while avoiding $\{(f,1)\}$ for all factors $f \in [1,k] \setminus \{a\}$.

Algorithm 3.3 is a case-by-case algorithm. We use the following lemma to define its first level of subcases.

Lemma 3.3.4 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k,2)$ whose associated error graph G has $d \leq 2$ edges corresponding to relevant interactions. Let $A = \big\{f \in [1,k] : \{v_{f,1}, v_{f,1}\} \in E(G)$ or $\exists j \in [1,k]$* such that *$\{v_{f,1}, v_{j,0}\} \in E(G)\big\}$, and assume that 0^k is a passing test. Then $|A| \in [0,2]$.*

Proof: Elements of A either correspond to loops or 1-ends of distinct 10-errors, so $0 \leq |A| \leq d \leq 2$. ∎

As a result of the preceding lemma, we can index our subcases of Case 0 and Case 1 by the values of $|A|$. This fact, combined with whether 1^k is a passing test (Case 0) or a failing test (Case 1), yields **Subcases** 0.0, 0.1, 0.2, 1.0, 1.1, and 1.2, which are outlined in the table on Page 62. Note that the aforementioned table contains five pairs of equivalent subcases. Equivalences (1), (2), and (3) can be seen by swapping a with b, since $a, b \in A$ in subcases 0.2 and 1.2. Equivalences (4) and (5) can be seen by swapping b with c, since $b, c \in A' = [1, k] \setminus A$.

(1) Subcases 0.2.2 and 0.2.3. The former occurs when $T^{(a)}$ is a failing test and $T^{(b)}$ is a passing test within subcase 0.2 (see Algorithm 3.4 on Page 67). The latter occurs when $T^{(a)}$ is a passing test and $T^{(b)}$ is a passing test within subcase 0.2.

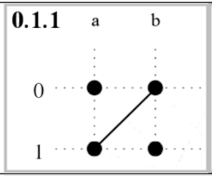
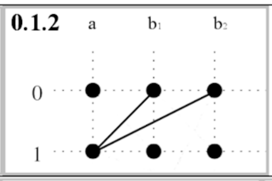
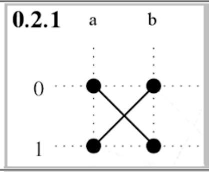
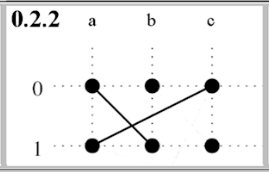
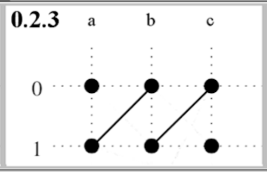
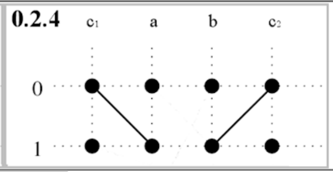
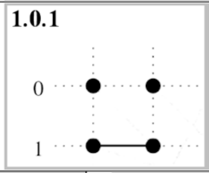
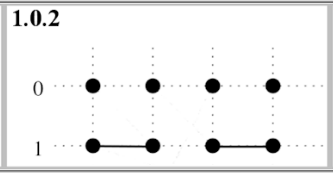
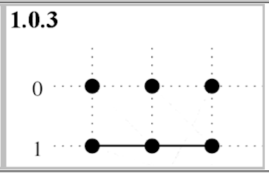
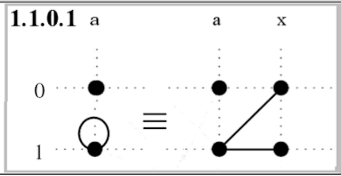
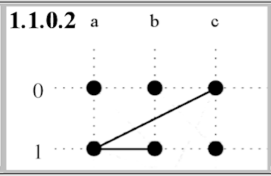
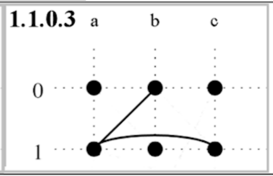
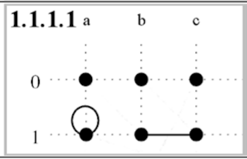
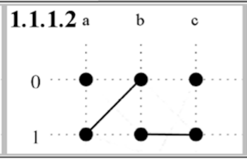
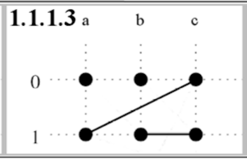
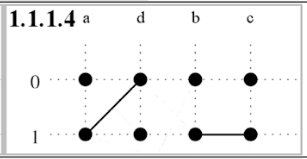
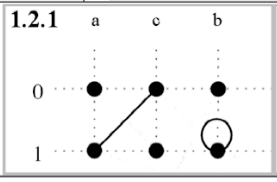
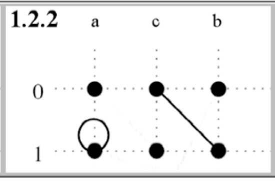
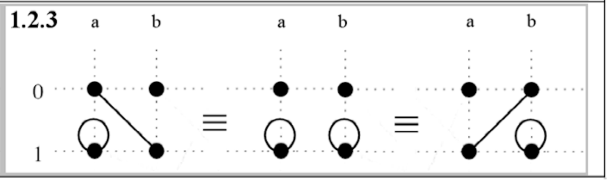
(2) Subcases 1.2.1 and 1.2.2. The former occurs when $T^{(b)}$ is a passing test within subcase 1.2 (see Algorithm 3.5 on Page 70). The latter occurs when $T^{(a)}$ is a passing test within subcase 1.2.

(3) The first and third graphs in subcase 1.2.3. There is no set of tests which can distinguish between any of the three (location-equivalent) graphs in subcase 1.2.3. However, we note that exactly three distinct error graphs are possible.

(4) Subcases 1.1.0.2 and 1.1.0.3. Each can occur within subcase 1.1.0, if the call to FindPassingTest within A1No11inAprime returns a passing test P (see Algorithm 3.7 on Page 74). The former occurs when $\overline{T^{(b)}}$ is a failing test, and the latter occurs when $\overline{T^{(b)}}$ is a passing test.

(5) Subcases 1.1.1.2 and 1.1.1.3. The former occurs when, within subcase 1.1.1, $T^{(b)}$ is a failing test and $T^{(c)}$ is a passing test (again, see Algorithm 3.7). The latter occurs when $T^{(b)}$ is a passing test and $T^{(c)}$ is a failing test.

The reader may notice that two subcases seem to be missing from within subcase 1.1: those with error sets $\big\{\{(a,l)\},\{(a,l),(b,0)\}\big\}$ and $\big\{\{(a,l)\},\{(a,l),(b,1)\}\big\}$, respectively. However, we concern ourselves only with error graphs whose edge sets are independent (in the sense of Definition 2.1.1).

Case	Condition	Subcase	|A|	A	Errors
0	Test(0^k) passes and Test(1^k) passes	0.0	0	ϕ	(none)
		0.1	1	{a}	 
		0.2	2	{a,b}	   
1	Test(0^k) passes and Test(1^k) fails	1.0	0	ϕ	  
		1.1	1	{a}	1.1.0   
					1.1.1    
		1.2	2	{a,b}	  

Next we prove that the procedure LocateAllErrors (given as Algorithm 3.3 on Page 64) correctly determines whether G contains a type-d (strongly nonlocatable) subgraph, and if G contains no such subgraph, then LocateAllErrors identifies and returns the set E of errors which correspond to the edges of either G or a graph G' which is location-equivalent to G.

Theorem 3.3.5 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose associated error graph G has $d \leq 2$ edges (and $E(G)$ is independent, in the sense of Definition 2.1.1).*

Assume the correctness of the two main subprocedures of LocateAllErrors, *namely* No11sNorLoops *and* Has11sOrLoops *(to be proven in Lemmas 3.3.6 and 3.3.8, respectively). The former procedure corresponds to Case 0, and is given as Algorithm 3.4 on Page 67. The latter procedure corresponds to Case 1, and is given as Algorithm 3.5 on Page 70.*

Then LocateAllErrors *either determines that G contains a type-d strongly nonlocatable subgraph (and prints a warning message), or it returns the set E of all errors corresponding to the edges of either G or a graph G' which is location-equivalent to G.*

Proof: Step 0 is correct as a result of Theorem 3.2.10.

If $P \neq null$ when Step 0 concludes, then the following conditions are satisfied: 0^k is a passing test (due to the relabeling of some (factor, value) pairs in Step 0), and G contains no strongly nonlocatable type-d subgraph. In particular, after Step 0 concludes, every error present in our $TP(k, 2)$ is either a 1-error (loop), a 10-error, or a 11-error.

We now show that Step 1 correctly computes the set

$$A = \{f \in [1, k] : \{v_{f,1}, v_{f,1}\} \in E(G) \text{ or } \exists j \in [1, k] \text{ such that } \{v_{f,1}, v_{j,0}\} \in E(G)\}.$$

Algorithm 3.3 Under the assumptions of Theorem 3.3.5, LocateAllErrors does the following when applied to a $TP(k,2)$ whose associated (unknown) error graph is G:
If G contains a type-d (strongly nonlocatable) subgraph, then this algorithm prints a warning message and exits. Otherwise, it returns the set E of all errors corresponding to the edges in either G or location-equivalent graph G'.

```
procedure LocateAllErrors($k$)
    ▷ Step 0: Find a passing test $P$, if possible, and relabel the $v \in V(G)$ so that $P = 0^k$:
    $P \leftarrow$ FindPassingTest($k$)
    if $P = null$ then print "Warning! Strongly nonlocatable subgraph" and exit
    else
        $S \leftarrow \emptyset$
        for each factor $f$ such that $P_f = 1$ do
            $S \leftarrow S \cup \{f\}$, and swap labels $(f,0)$ and $(f,1)$.

    ▷ Step 1: Discover the factors corresponding to loops and 1-ends of 10-errors.
    $\mathcal{C} \leftarrow$ BinaryCA($k$)
    Let $T[i]$ denote the $i$th test (row) of $\mathcal{C}$, where $i \in [1,N]$ is an integer.
    $A \leftarrow \emptyset$, and $i \leftarrow 1$
    while $|A| < 2$ and $i < N$ do
        if Test($T[i]$) = fail then
            $A \leftarrow A \cup$ SearchEndPoint($T[i], B$) where $B = \{f : T[i]_f = 1\}$
        $i \leftarrow i + 1$

    ▷ Step 2: Determine if there is at least one loop or 11-error.
    if Test($1^k$) = pass then $E \leftarrow$ No11sNorLoops($A, k$) ▷ ——————[Case 0]
    else $E \leftarrow$ Has11sOrLoops($A, k$) ▷ ————————————————[Case 1]
    for each factor $f \in S$ do swap labels $(f,0)$ and $(f,1)$.
    return $E$

▷ Construct an optimal $CA(N;2,k,2)$ (see Theorems 1.2.3 and 1.2.5).
procedure BinaryCA($k$)
    $N \leftarrow \min\left\{N : \binom{N-1}{\lceil N/2 \rceil} \geq k\right\}$
    Let $S$ be the set of all distinct binary $N$-tuples such that each tuple has a zero
    in its first position, and exactly $\lceil N/2 \rceil$ ones.
    Let $\mathcal{C}$ be a matrix whose columns are the elements of $S$.
    return $\mathcal{C}$.
```

Step 1 begins by building a strength-2 binary covering array $\mathcal{C}$ via the subprocedure BinaryCA. Each failing test corresponding to a row of $\mathcal{C}$ covers at least one error in our $TP(k, 2)$, and by Definition 1.2.1, every interaction of strength up to 2 (and hence, every edge of G) is covered by at least one row of $\mathcal{C}$. In particular, every loop and 10-error is covered by at least one failing test which is a row of $\mathcal{C}$.

Let T be a failing test corresponding to a row of $\mathcal{C}$, and let $[B, B']$ be a partition of $[1, k]$ such that $B = \{f : T_f = 1\}$. Then all three conditions of Lemma 3.3.1 are satisfied by T and $[B, B']$ since $\overline{T_B}T_{B'} = 0^k$, which is a passing test (and hence, covers no errors - see Step 0).

By part (a) of Lemma 3.3.1, if there exist $f \in B$, $f' \in B'$ such that $\{v_{f,1}, v_{f',0}\} \in E(G)$, then f is returned by SearchEndPoint(T, B). Similarly, by part (b) of Lemma 3.3.1, if there exists $l \in B$ such that $\{v_{l,1}, v_{l,1}\} \in E(G)$, then l is returned by SearchEndPoint(T, B). Therefore SearchEndPoint(T, B) returns the set of factors in B which are also in A.

Note that multiple calls to SearchEndPoint may find the same factor multiple times, and SearchEndPoint(T, B) may return the empty set if T covers only 11-errors. However, LocateAllErrors does not need to check for more elements of A if at any point $|A| = 2$, by Lemma 3.3.4. As long as A contains fewer than two elements, and there are still failing tests T which are rows of $\mathcal{C}$, we run SearchEndPoint(T, B), where $B = \{f : T_f = 1\}$. Therefore, after Step 1, the following conditions are satisfied: 0^k is a passing test and $A = \{f \in [1, k] : \{v_{f,1}, v_{f,1}\} \in E(G)$ or $\exists j \in [1, k]$ such that $\{v_{f,1}, v_{j,0}\} \in E(G)\}$.

If 1^k is also a passing test, then the preconditions of No11sNorLoops are satisfied (see Lemma 3.3.6). By the correctness of No11sNorLoops under these assumptions (to be proven in Lemma 3.3.6), it returns the set E of all errors.

Otherwise, 1^k is a failing test, in which case the preconditions of Has11sOrLoops are satisfied (see Lemma 3.3.8). By the correctness of Has11sOrLoops under these assumptions (to be proven in Lemma 3.3.8), it returns the set E of all errors, or a location-equivalent set of errors.

Recall that some (factor, value) pairs were relabeled in Step 0. This algorithm concludes Step 2 by restoring the original labels for each (factor, value) pair that was changed in Step 0. Therefore the set E returned by LocateAllErrors(k) corresponds to the set of edges of either G or a graph G' which is location-equivalent to G. ∎

We now proceed to prove that the procedure No11sNorLoops correctly returns the set E of all errors (corresponding to Case 0) if its preconditions are satisfied.

Lemma 3.3.6 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k,2)$ whose associated error graph G has $d \leq 2$ edges. Let $A = \big\{f \in [1,k] : \{v_{f,1}, v_{f,1}\} \in E(G)$ or $\exists j \in [1,k]$ such that $\{v_{f,1}, v_{j,0}\} \in E(G)\big\}$. Assume that 0^k and 1^k are passing tests (i.e. we are in Case 0).*

Then the procedure No11sNorLoops(A, k) *(given as Algorithm 3.4) correctly identifies and returns the set of all errors in the given $TP(k,2)$.*

Proof: Since 1^k is a passing test, our $TP(k,2)$ has only 10-errors, and the 1-ends of these errors must be elements of A. Recall that $|A| \in [0,2]$, by Lemma 3.3.4. We now prove that for each subcase (see the table on Page 62), the procedure No11sNorLoops correctly identifies and returns the set E of all errors in the given $TP(k,2)$.

Subcase 0.0: We have $A = \emptyset$, therefore there are no errors, and No11sNorLoops clearly returns $\emptyset$.

Subcase 0.1: We have $|A| = 1$, so let $A = \{a\}$ and let $A' = [1,k] \setminus A$. There are up to two 10-errors across the partition $[A, A']$ of $[1,k]$, each sharing $(a,1)$ as the common 1-end. The test $\overline{T^{(a)}}$ has value 1 for factor a, and value 0 for all other factors, therefore it covers (all of) the 10-errors. Let R be the set returned by SearchEndPoint($\overline{T^{(a)}}, A'$).

Algorithm 3.4 Under the assumptions of Lemma 3.3.6, NO11SNORLOOPS (when applied to a $TP(k,2)$ whose associated error graph is G) returns the set E of all errors (which are 10-errors) corresponding to the edges in G.

```
procedure NO11sNORLOOPS($A, k$) ▷ ——————————————————[Case 0]
    $E \leftarrow \emptyset$ ▷ ————————————————————————-[Subcase 0.0: $\mathbf{A = \emptyset}$]
    $A' \leftarrow [1,k] \setminus A$
    if $A \neq \emptyset$ then
        Let $a \in A$.
        if $|A| = 1$ then ▷ ———————————————————[Subcase 0.1]
            ▷ Up to two errors sharing a common 1-end.
            $B \leftarrow$ SEARCHENDPOINT($\overline{T^{(a)}}, A'$)
            for $b \in B$ do
                $E \leftarrow E \cup \{(a,1),(b,0)\}$
        else ▷ ————————————————————————————[Subcase 0.2]
            ▷ Two errors, possibly in an 'X' formation.
            Let $A = \{a, b\}$.
            $E \leftarrow$ LOCATE01($a, b$) $\cup$ LOCATE01($b, a$)
    return $E$

▷ Under the assumptions of Lemma 3.3.6 and Subcase 0.2, let $A = \{x, y\}$,
▷ where $(y, 1)$ is the 1-end of a 10-error whose 0-end is either $(x, 0)$ or within $A'$.
procedure LOCATE01($x, y$)
    $A' \leftarrow [1,k] \setminus \{x, y\}$
    if TEST($T^{(x)}$) = fail then
        $e \leftarrow \{(x,0),(y,1)\}$
    else
        $\{c\} \leftarrow$ SEARCHENDPOINT($\overline{T^{(y)}}, A'$)
        $e \leftarrow \{(y,1),(c,0)\}$
    return $\{e\}$
```

In subcase 0.1.1, the only error is $e = \{(a,1),(b,0)\}$ for some $b \in A'$. We apply Lemma 3.3.1 with respect to $\overline{T^{(a)}}$ and the partition $[A', A]$ of $[1, k]$. Since $b \in A'$, we have $b \in R$. Furthermore, since e is the only error, we have $R = \{b\}$.

In subcase 0.1.2, the only errors are $e_1 = \{(a,1),(b_1,0)\}$ and $e_2 = \{(a,1),(b_2,0)\}$ for some $b_1, b_2 \in A'$. Again, we apply Lemma 3.3.1 with respect to $\overline{T^{(a)}}$ and the partition $[A', A]$ of $[1, k]$. Since $b_1, b_2 \in A'$, we have $b_1, b_2 \in R$. Furthermore, since e_1 and e_2 are the only errors, we have $R = \{b_1, b_2\}$.

In either subcase, the for-loop matches each 0-end to the common 1-end, and stores each error in E, which No11sNorLoops returns.

Subcase 0.2: We have $|A| = 2$, so let $A = \{a, b\}$. Then for each of the two 10-errors, the 0-end is either within $A = \{a, b\}$ or not, therefore there are four subcases (0.2.1, 0.2.2, 0.2.3, and 0.2.4) of Subcase 0.2, and they are depicted in the table on Page 62.

We now look at the two calls to the auxiliary subprocedure Locate01. Suppose that $T^{(x)}$ fails in both calls to Locate01. Then $T^{(a)}$ and $T^{(b)}$ are both failing tests which cover the 10-errors $\{(a,0),(b,1)\}$ and $\{(a,1),(b,0)\}$, respectively, and we are in Subcase 0.2.1. In this case, No11sNorLoops returns the set $E = \big\{\{(a,0),(b,1)\},\{(b,0),(a,1)\}\big\}$.

Next, notice the symmetry between Subcases 0.2.2 and 0.2.3 (refer to the table on Page 62): if we relabel the factors a, b in Subcase 0.2.2 as b, a, then we get Subcase 0.2.3. The two cases are equivalent by symmetry; the former occurs when $T^{(x)}$ fails in only the first call to Locate01, and the latter occurs when $T^{(x)}$ fails in only the second call to Locate01.

Without loss of generality, suppose that $T^{(x)}$ fails in only the first call to Locate01. Then $T^{(a)}$ is a failing test, and the first call to Locate01 returns a set containing one 10-error, $e = \{(a,0),(b,1)\}$. Additionally, $T^{(b)}$ is a passing test, therefore the other 10-error is $e' = \{(a,1),(c,0)\}$ for some $c \in A'$, and we are in Subcase 0.2.2. We apply Lemma 3.3.1 with respect to $\overline{T^{(b)}}$ and $[A', A]$ Let R be the set returned by

SearchEndPoint($\overline{T^{(b)}}, A'$). Since $c \in A'$, we have $c \in R$. Furthermore, since e and e' are the only errors, and $a, b \notin A'$, the set R contains neither a nor b (i.e. $R = \{c\}$). In this case, No11sNorLoops returns the set $E = \big\{\{(a,0),(b,1)\},\{(a,1),(c,0)\}\big\}$ for the appropriate factor $c \in A'$.

Now, suppose that $T^{(x)}$ passes in both calls to Locate01 (i.e. $T^{(a)}$ and $T^{(b)}$ are both passing tests). Therefore there is no 0-end at either of a, b, so the 10-errors are $e = \{(a,1),(c_1,0)\}$ and $e' = \{(b,1),(c_2,0)\}$ for some factors $c_1, c_2 \in A'$, and we are in Subcase 0.2.4. Note that the (failing) tests $\overline{T^{(a)}}$ and $\overline{T^{(b)}}$ cover e and e', respectively.

We first apply Lemma 3.3.1 with respect to $\overline{T^{(a)}}$ and $[A', A]$. Let R be the set returned by SearchEndPoint($\overline{T^{(a)}}, A'$). Since $c_1 \in A'$, we also have $c_1 \in R$. Furthermore, since e is the only error satisfying statement (c) of Lemma 3.3.1, we in fact have $R = \{c_1\}$.

Next, we apply Lemma 3.3.1 with respect to $\overline{T^{(b)}}$ and $[A', A]$. This is equivalent to the preceding application of this lemma, by symmetry. Hence, $R = \{c_2\}$ is returned by SearchEndPoint($\overline{T^{(b)}}, A'$).

Therefore, in Subcase 0.2, the procedure No11sNorLoops returns the set $E = \big\{\{(a,1),(c_1,0)\},\{(b,1),(c_2,0)\}\big\}$. ∎

Our main procedure for Case 1, Has11sOrLoops, has subprocedures related to its subcases. When we are in Subcase 1.0 or Subcase 1.1.1, we have a set D such that every error within D is a 11-error, and we use the subprocedure called Locate11s (see Algorithm 3.6). It is a simplified version of LocateErrorsInTest from [32]. We briefly describe how it works here, however we omit its proof of correctness since Chapter 5 contains a proof of correctness for a more generalized, higher-strength version of LocateErrorsInTest, called LocateTriple (see Lemma 5.1.6 and Algorithm 5.3).

Locate11s is a recursive procedure. In the base case where $|D| = 2$, it returns

Algorithm 3.5 Under the assumptions of Lemma 3.3.8, this procedure (when applied to a $TP(k, 2)$ whose associated error graph is G) returns the set E of all errors corresponding to the edges in either G or a location-equivalent graph G'.

```
procedure Has11sOrLoops($A, k$) ▷ ——————————————[Case 1]
    $E \leftarrow \emptyset$
    $A' \leftarrow [1, k] \setminus A$
    if $A = \emptyset$ then ▷ ——————————————[Subcase 1.0]
        ▷ Only 11-errors are possible.
        $E \leftarrow$ Locate11s($A', k$)
    else
        if $|A| = 1$ then ▷ ——————————————[Subcase 1.1]
            Let $A = \{a\}$.
            ▷ There may be a 11-error.
            if Test($T^{(a)}$) = pass then
                $E \leftarrow$ A1No11inAprime($a, k$) ▷ ——————————-[Subcase 1.1.0]
            else
                $E \leftarrow$ A1Locate11inAprime($a, k$) ▷ ——————[Subcase 1.1.1]
        else ▷ ——————————————-[Subcase 1.2]
            Let $A = \{a, b\}$.
            ▷ At least one of $\{(a, 1)\}, \{(b, 1)\}$ is a loop. If there are not two loops,
            ▷ then there is a 1-end of a 10-error at the factor which does not
            ▷ correspond to a loop. Tests $T^{(a)}, T^{(b)}$ cannot both pass, since $1^k$ failed.
            if Test($T^{(b)}$) = pass then
                $E \leftarrow$ A2No11inAprime($a, b, k$) ▷ ——————-[Subcase 1.2.1]
            else
                if Test($T^{(a)}$) = pass then
                    $E \leftarrow$ A2No11inAprime($b, a, k$) ▷ ——————[Subcase 1.2.2]
                else
                    ▷ A loop at each of $a, b$, or a location-equivalent subgraph.
                    $E \leftarrow \big\{\{(a, 1)\}, \{(b, 1)\}\big\}$ ▷ ——————[Subcase 1.2.3]
    return $E$
```

a 11-error whose factors constitute D. Otherwise, it partitions D into approximate halves B and C, and defines tests T and T', each of which has 1s in one (approximate) half of D, and 0s elsewhere. If one of T, T' fails, then there is a 11-error within either B or C, respectively, and the procedure is called recursively.

At any given step, Locate11s determines whether there may be errors across the partition $[B, C]$ of D. If neither B nor C contain an error, then there are either one or two 11-errors across $[B, C]$, in which case Locate11s calls the subprocedure LocateOneOrTwo11sAcross. This subprocedure first calls SearchEndPoint to identify the ends in B of the 11-errors across $[B, C]$. For each end identified in B, another call to SearchEndPoint identifies its matching end in C.

If B contains one error, and C does not, then there may be one error across $[B, C]$. In this case, Locate11s calls the subprocedure LocateOne11Across. The case where C contains one error (and B does not) is symmetrical. All tests conducted by LocateOne11Across avoid the error $e = \{(b_1, 1), (b_2, 1)\}$ within B. The first test, T, fails if $\{(b_1, 1), (c, 1)\}$ is an error, for some $c \in C$. In this case, this subprocedure identifies c via one call to SearchEndPoint. The second test, T', fails if $\{(b, 1), (c, 1)\}$ is an error, for some $b \in B \setminus \{b_1\}, c \in C$. In this case, this subprocedure identifies b and c via two calls to SearchEndPoint. If both T and T' pass, then there is no error across $[B, C]$, in which case LocateOne11Across returns the empty set.

The following lemma summarizes how Locate11s functions for the two subcases in which it is called by our main algorithm.

Lemma 3.3.7 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose associated graph G has $d \leq 2$ edges representing errors of strength at most 2. Assume that 0^k is a passing test and 1^k is a failing test. Let $A = \{f \in [1, k] : \{v_{f,1}, v_{f,1}\} \in E(G)$ or $\exists j \in [1, k]$ such that $\{v_{f,1}, v_{j,0}\} \in E(G)\}$ and let $D = [1, k] \setminus A$.*

1. If G has neither loops nor 10-errors (Subcase 1.0), then Locate11s(D, k)

identifies and returns the set of all errors in the given $TP(k, 2)$.

2. If there is exactly one 11-error within D and exactly one error e of the form $\{(a, 1)\}$ or $\{(a, 1), (g, 0)\}$ for $a \in A, g \in D$ (Subcase 1.1.1), then Locate11s(D, k) *identifies and returns the set containing the only 11-error e' which is within D.*

Proof: 1. In Subcase 1.0, every error is a 11-error within $D = [1, k]$.

2. In Subcase 1.1.1, every test T conducted within Locate11s(D, k) avoids e by setting $T_a = 0$, therefore T fails if and only if it covers e'. ∎

Now that we can identify 11-errors whenever they are the only errors in a set D, we prove the correctness of the main procedure for Case 1, Has11sOrLoops (see Algorithm 3.5 on Page 70).

Lemma 3.3.8 *Let $k = 2^z$ for some integer $z \geq 1$, and consider a $TP(k, 2)$ whose associated error graph G has $d \leq 2$ edges (and $E(G)$ is independent, in the sense of Definition 2.1.1). Assume that 0^k is a passing test, 1^k is a failing test, and let $A = \{f \in [1, k] : \{v_{f,1}, v_{f,1}\} \in E(G)$ or $\exists j \in [1, k]$ such that $\{v_{f,1}, v_{j,0}\} \in E(G)\}$.*

Then the procedure Has11sOrLoops(A, k) *returns the set E of all errors corresponding to the edges of either G or a graph G' which is location-equivalent to G (see Corollary 3.1.4).*

Proof: Since 1^k is a failing test, our $TP(k, 2)$ has at least one error that is either a loop or a 11-error. Therefore we are in Case 1. We now prove that for each subcase (see the table on Page 62), the procedure Has11sOrLoops correctly identifies and returns the set E of all errors corresponding to the edges of either G or a graph G' which is location-equivalent to G

Subcase 1.0: We have $|A| = 0$, so there are neither loops nor 1-ends of 10-errors. Therefore, only 11-errors are present in this $TP(k, 2)$, and by Lemma 3.3.7, they are returned by one call to Locate11s.

Algorithm 3.6 Under the assumptions of Lemma 3.3.7, this procedure does the following: if we are in Subcase 1.0, then LOCATE11S returns the set of all errors (one or two 11-errors), and if we are in Subcase 1.1.1, then LOCATE11S returns the set containing the only 11-error, which is within D.

```
procedure LOCATE11S($D, k$) ▷ ————————-[Subcase 1.0] or [Subcase 1.1.1]
    $E \leftarrow E' \leftarrow E'' \leftarrow \emptyset$
    if $|D| = 2$ then Let $D = \{b, c\}$, and $E \leftarrow \{(b,1),(c,1)\}$.
    else
        Partition $D$ as $[B, C]$ such that $|C| \leq |B| \leq |C| + 1$.
        Define $T$ by: $T_f \leftarrow \begin{cases} 1 & \text{if } f \in B \\ 0 & \text{otherwise} \end{cases}$
        Define $T'$ by: $T'_f \leftarrow \begin{cases} 1 & \text{if } f \in C \\ 0 & \text{otherwise} \end{cases}$
        if TEST($T$) = fail then $E \leftarrow$ LOCATE11S($B, k$)
        if TEST($T'$) = fail then $E' \leftarrow$ LOCATE11S($C, k$)
        if $E = \emptyset = E'$ then $E'' \leftarrow$ LOCATEONEORTWO11SACROSS($B, C, k$)
        if $|E| = 1$ and $E' = \emptyset$ then $E'' \leftarrow$ LOCATEONE11ACROSS($E, B, C, k$)
        if $E = \emptyset$ and $|E'| = 1$ then $E'' \leftarrow$ LOCATEONE11ACROSS($E', C, B, k$)
    return $E \cup E' \cup E''$

▷ Precondition: only 11-errors across $[B, C]$ are present in the given $TP(k, 2)$.
▷ This subprocedure of LOCATE11S returns the set containing those 11-errors.
procedure LOCATEONEORTWO11SACROSS($B, C, k$)
    $V \leftarrow$ SEARCHENDPOINT($1^k, B$), and $E'' \leftarrow \emptyset$
    for $b \in V$ do
        Define $T$ by: $T_f \leftarrow \begin{cases} 1 & \text{if } f \in \{b\} \cup C \\ 0 & \text{otherwise} \end{cases}$
        $V' \leftarrow$ SEARCHENDPOINT($T, C$)
        for $c \in V'$ do $E'' \leftarrow E'' \cup \{(b,1),(c,1)\}$
    return $E''$
▷ Precondition: $E$ contains a 11-error which is within $B$. If there is a 11-error
▷ across $[B, C]$, then this subprocedure returns the set containing that 11-error.
procedure LOCATEONE11ACROSS($E, B, C, k$)
    Let $E = \big\{\{(b_1,1),(b_2,1)\}\big\}$. ▷ Avoid $E$, check for an error across $[B, C]$.
    Define $T$ by: $T_f \leftarrow \begin{cases} 1 & \text{if } f \in \{b_1\} \cup C \\ 0 & \text{otherwise} \end{cases}$
    if TEST($T$) = fail then
        $\{c\} \leftarrow$ SEARCHENDPOINT($T, C$), then return $\{(b_1,1),(c,1)\}$.
    else
        Define $T'$ by: $T'_f \leftarrow \begin{cases} 1 & \text{if } f \in B \cup C \setminus \{b_1\} \\ 0 & \text{otherwise} \end{cases}$
        if TEST($T'$) = fail then
            $\{c\} \leftarrow$ SEARCHENDPOINT($T', C$)
            $\{b\} \leftarrow$ SEARCHENDPOINT($T', B$), then return $\{(b,1),(c,1)\}$.
        else return $\emptyset$
```

Algorithm 3.7 Under the assumptions of Lemma 3.3.8, the following subprocedures for Case 1 have the following preconditions: A1No11inAprime requires that $A = \{a\}$ and $T^{(a)}$ is a passing test, A1Locate11inAprime requires that $A = \{a\}$ and $T^{(a)}$ is a failing test, and A2No11inAprime requires that $A = \{a, b\}$ and $T^{(b)}$ is a passing test. Each subprocedure returns the set E of all errors corresponding to edges of either G or a location-equivalent graph G'.

```
procedure A1No11inAprime($a, k$) ▷ ——————————[Subcase 1.1.0]
    For each test $T$ within the following call to FindPassingTest, we fix $T_a = 1$
    and we index all subsequent tests by the set $X$ of factors, where factor $k + 1$
    is a dummy factor whose values do not influence results.
    $X \leftarrow [1, k + 1] \setminus \{a\}$.
    $P \leftarrow$ FindPassingTest($X, k$)
    if $P = null$ then $E \leftarrow \{(a, 1)\}$ ▷ ——————————[Subcase 1.1.0.1]
    else
        ▷ Two errors: a 10-error and a 11-error.
        Define $T'$ by: $T'_f \leftarrow \begin{cases} 0 & \text{if } f = a \\ \overline{P_f} & \text{otherwise} \end{cases}$
        $\{b, c\} \leftarrow$ SearchEndPoint($T', A'$), where $A' = X \setminus \{k + 1\}$.
        if Test($\overline{T^{(b)}}$) = fail then
            $e_{11} \leftarrow \{(a, 1), (b, 1)\}$ ▷ ——————————[Subcase 1.1.0.2]
            $e_{10} \leftarrow \{(a, 1), (c, 0)\}$
        else
            $e_{11} \leftarrow \{(a, 1), (c, 1)\}$ ▷ ——————————[Subcase 1.1.0.3]
            $e_{10} \leftarrow \{(a, 1), (b, 0)\}$
        $E \leftarrow \{e_{11}, e_{10}\}$
    return $E$

procedure A1Locate11inAprime($a, k$) ▷ ——————————-[Subcase 1.1.1]
    $A' \leftarrow [1, k] \setminus \{a\}$
    ▷ Every test $T$ conducted within Locate11s($A', k$), has $T_a = 0$ since $a \notin A'$.
    $\{e_{11}\} \leftarrow$ Locate11s($A', k$), and let $e_{11} = \{(b, 1), (c, 1)\}$.
    if Test($T^{(b)}$) = fail then
        if Test($T^{(c)}$) = fail then $e_1 \leftarrow \{(a, 1)\}$ ▷ ——————-[Subcase 1.1.1.1]
        else $e_1 \leftarrow \{(a, 1), (b, 0)\}$ ▷ ——————————-[Subcase 1.1.1.2]
    else
        if Test($T^{(c)}$) = fail then $e_1 \leftarrow \{(a, 1), (c, 0)\}$ ▷ ————[Subcase 1.1.1.3]
        else
            $\{d\} \leftarrow$ SearchEndPoint($\overline{T^{(a)}}, A' \setminus \{b, c\}$)
            $e_1 \leftarrow \{(a, 1), (d, 0)\}$ ▷ ——————————[Subcase 1.1.1.4]
    return $\{e_{11}, e_1\}$

procedure A2No11inAprime($a, b, k$) ▷ ——[Subcase 1.2.1] or [Subcase 1.2.2]
    $A' \leftarrow [1, k] \setminus \{a, b\}$, and let $e_1 = \{(b, 1)\}$.
    $\{c\} \leftarrow$ SearchEndPoint($\overline{T^{(a)}}, A'$), and $e_{10} \leftarrow \{(a, 1), (c, 0)\}$
    return$\{e_{10}, e_1\}$
```

Subcase 1.1: We have $|A| = 1$, so let $A = \{a\}$. There is at least one error which is either a loop $\{(a, 1)\}$ or a 10-error $\{(a, 1), (g, 0)\}$ for some factor $g \in A' = [1, k] \setminus A$. If the test $T^{(a)}$ passes, then there are no 11-errors within A', since $T^{(a)}_f = 1$ for all factors $f \in A'$. This is case 1.1.0., whose correctness is given below. Otherwise, $T^{(a)}$ necessarily fails due to a 11-error within A' (by the definition of $T^{(a)}$). This is Subcase 1.1.1., whose correctness is also given below.

We remark here that there are two subcases of Subcase 1.1 that we do not consider, since we assume that the edge set of G is independent (in the sense of the corresponding interactions - see Definition 2.1.1: $E = \big\{\{(a, 1)\}, \{(a, 1), (b, 0)\}\big\}$ and $E = \big\{\{(a, 1)\}, \{(a, 1), (b, 1)\}\big\}$.

Subcase 1.1.0: The procedure A1No11inAprime first determines whether we have a loop $\{(a, 1)\}$ or a location-equivalent type-a subgraph whose edges correspond to a pair of nonlocatable errors of the form $\big\{\{(a, 1), (x, 0)\}, \{(a, 1), (x, 1)\}\big\}$.

For each test T within this procedure's call to FindPassingTest, we fix $T_a = 1$ and we index all subsequent tests by the set X of factors, where factor $k + 1$ is a dummy factor whose values do not influence results. By Theorem 3.2.11, the procedure FindPassingTest (Version 2) returns either *null*, or it returns a passing test P.

In the former case, $\{(a, 1)\}$ is either a loop or a 1-end common to a pair of nonlocatable errors corresponding to a type-a nonlocatable subgraph, so we are in Subcase 1.1.0.1. Therefore, $E = \{(a, 1)\}$ returned by A1No11inAprime is correct.

In the latter case, we know that $\{(a, 1)\}$ is not a loop and $|A| = 1$, so $\{(a, 1)\}$ is the 1-end of the only 10-error present, $e = \{(a, 1), (b, 0)\}$ for some factor $b \in A'$. We also know that the test 1^k failed, but there is no 11-error in $A' = [1, k] \setminus A$, therefore $\{(a, 1)\}$ is an end of a 11-error, $e' = \{(a, 1), (c, 1)\}$ for some factor $c \in A', c \neq b$. In particular, both $\{(b, 0)\}$ and $\{(c, 1)\}$ must be covered by the same test $\overline{T}$ since neither end was covered by T, because T is a passing test.

Next, notice the symmetry between Subcases 1.1.0.2 and 1.1.0.3 (refer to the table on Page 62): if we relabel the factors a, b in Subcase 1.1.0.2 as b, a, then we get Subcase 1.1.0.3. The two subcases are equivalent by symmetry; the former occurs when $\overline{T^{(b)}}$ fails, and the latter occurs otherwise.

We apply Lemma 3.3.1 with respect to T' and the partition $[A', A]$ of $[1, k]$. Let R be the set returned by $\textsc{SearchEndPoint}(T', A')$, where $A' = X \setminus \{k+1\}$. Since $b, c \in A'$, we also have $b, c \in R$. Furthermore, since e and e' are the only errors, and the 1-end $(a, 1)$ is within A (i.e. $a \notin A'$), the set R does not contain a (i.e. $R = \{b, c\}$).

If $\overline{T^{(b)}}$ is a failing test, then $\{(b, 1)\}$ is a 1-end of our 11-error, so $\{(c, 0)\}$ is the 0-end of our 10-error, and we are in Subcase 1.1.0.2. Otherwise, we are in Subcase 1.1.0.3.

<u>Subcase 1.1.1</u>: Recall that in Subcase 1.1, there is at least one error e of the form $\{(a, 1)\}$ or $\{(a, 1), g, 0)\}$ for some $g \in A'$. We call the procedure $\textsc{A1Locate11inAprime}$ if $T^{(a)}$ is a failing test, which only occurs if there is a 11-error e_{11} within $A' = [1, k] \setminus A$. By Lemma 3.3.7, $\textsc{Locate11s}(A', k)$ returns a set containing $e_{11} = \{(b, 1), (c, 1)\}$ for some factors $b, c \in A'$.

Recall that $\{(a, 1)\}$ could be a loop or a 1-end of a 10-error, and consider the tests $T^{(b)}$ and $T^{(c)}$. If both fail, then each test covers an error. However, both tests avoid e_{11}, and there is no factor f such that $T^{(b)}_f = 0 = T^{(c)}_f$, so the tests cannot both cover a common 10-error. Hence, $\{(a, 1)\}$ is a loop, and we are in Subcase 1.1.1.1. Therefore, $\textsc{A1Locate11inAprime}(a, k)$ correctly returns the set $\big\{\{(a, 1)\}, \{(b, 1), (c, 1)\}\big\}$

It is easy to see that Subcases 1.1.1.2 and 1.1.1.3 are equivalent, by symmetry (refer to the table on Page 62): if we relabel the factors b, c in Subcase 1.1.1.2 as c, b, then we get Subcase 1.1.1.3. If $T^{(b)}$ fails and $T^{(c)}$ passes, then $\{(a, 1)\}$ is not a loop, and the only factor set to 0 in $T^{(b)}$ is b, so we have a 10-error $\{(a, 1), (b, 0)\}$, and we are in Subcase 1.1.1.2. Therefore, $\textsc{A1Locate11inAprime}(a, k)$ correctly returns the set $\big\{\{(a, 1), (b, 0)\}, \{(b, 1), (c, 1)\}\big\}$. However, if $T^{(c)}$ fails and $T^{(b)}$ passes, then we are in Subcase 1.1.1.3, and $\textsc{A1Locate11inAprime}(a, k)$ correctly returns the set

$\{\{(a,1),(c,0)\},\{(b,1),(c,1)\}\}$

If both $T^{(b)}$ and $T^{(c)}$ pass, then $\{(a,1)\}$ is not a loop, and we have a 10-error $e_1 = \{(a,1),(d,0)\}$ for some factor $d \in A' \setminus \{b,c\}$. This is Subcase 1.1.1.4, and this error is covered by the (failing) test $\overline{T^{(a)}}$.

We apply Lemma 3.3.1 with respect to $\overline{T^{(a)}}$ and the partition $[A' \setminus \{b,c\}, A \cup \{b,c\}]$ of $[1,k]$. Let R be the set returned by SearchEndPoint$(\overline{T^{(a)}}, A' \setminus \{b,c\})$. Since $d \in A'$, we also have $d \in R$. Furthermore, since e_{11} and e_1 are the only errors, and e_{11} and the 1-end of e_1 are both within $A \cup \{b,c\}$ (i.e. $a,b,c \notin A' \setminus \{b,c\}$), the set R does not contain a,b, or c (i.e. $R = \{d\}$).

Subcase 1.2: We have $|A| = 2$, so let $A = \{a,b\}$, and $A' = [1,k] \setminus A$. There are no 11-errors, since each of a,b must be either a loop or a 1-end of a 10-error, and the total number of errors in the given $TP(k,2)$ is at most 2. Since 1^k is a failing test, at least one of a,b must be a loop. Notice the symmetry between Subcases 1.2.1 and 1.2.2 (refer to the table on Page 62): if we relabel the factors a,b in Subcase 1.2.1 as b,a, then we get Subcase 1.2.2.

If $T^{(b)}$ passes, then $\{(a,1)\}$ is not a loop, therefore $e_1 = \{(b,1)\}$ is a loop. Furthermore, $\{(a,1)\}$ is the 1-end of a 10-error whose 0-end cannot be $\{(b,0)\}$, otherwise $T^{(b)}$ would have failed. Therefore we are in Subcase 1.2.1, and a call to A2No11inAprime is performed. The 10-error $e_{10} = \{(a,1),(c,0)\}$, for some $c \in A'$, is covered by $\overline{T^{(a)}}$.

We apply Lemma 3.3.1 with respect to $\overline{T^{(a)}}$ and the partition $[A', A]$ of $[1,k]$. Since $c \in A'$, we also have $c \in R$. Furthermore, since e_1 and e_{10} are the only errors, and $a,b \notin A'$, then set R contains neither a nor b (i.e. $R = \{c\}$).

Subcase 1.2.2 is, similarly, handled by a call to A2No11inAprime.

However, if both $T^{(a)}$ and $T^{(b)}$ fail, then we are in Subcase 1.2.3, and we have two loops $\{(a,1)\}$ and $\{(b,1)\}$, or a set of errors from a nonlocatable subgraph which is location-equivalent to G. Note that the first and third graphs depicted in the table on Page 62 are distinct, but no set of tests can distinguish them from each other or from a pair of loops at a and b. In this case, Has11sOrLoops correctly returns the set containing both loops, $\{\{(a, 1)\}, \{(b, 1)\}\}$ ∎

3.4 Algorithm Analysis

In this section, we prove that Algorithm 3.3 conducts at most $4(\log_2 k)^2 + \mathcal{O}(\log_2 k)$ tests. After several auxiliary lemmas, this result is given as Theorem 3.4.9.

Before we give the maximum number of tests conducted by SearchEndPoint, we need the following auxiliary lemma. Some details in the proof of the following lemma come from a special case of Theorem 1 on Page 428 of [38].

Lemma 3.4.1 *Let S_n be a sequence of integers satisfying*

$$S_n = \begin{cases} 0 & \text{if } n = 1 \\ 2 + S_{\lceil n/2 \rceil} & \text{if } n > 1 \end{cases}$$

Then $S_n \leq 2\lceil \log_2 n \rceil$ for all $n \geq 1$.

Proof: Let $S(n) = S_n$, and let $P(z) = S(2^z)$ for any integer $z \geq 0$. We first prove the following claim $\mathcal{C}$.

$\underline{\mathcal{C}}$: For every integer $z \geq 0$, $P(z) \leq 2z$.

We proceed by induction on z. If $z = 0$, then the inequality clearly holds, since $P(0) = S(1) = 0$.

Now, suppose that for some integer $z \geq 0$, we have $P(z) \leq 2z$. Consider $P(z+1)$. By definition, we have:

$$P(z+1) = S(2^{z+1}) = 2 + S(2^z) = 2 + P(z).$$

By the induction hypothesis, $P(z) \leq 2z$. Hence

$$P(z+1) = 2 + P(z) \leq 2 + 2z = 2(z+1).$$

Therefore claim $\mathcal{C}$ holds.

Next, we prove that $S(n) \leq 2\lceil \log_2 n \rceil$ for all $n \in \mathbb{Z}^+$.

If there exists an integer $z \geq 0$ such that $n = 2^z$, then we are done, by our claim $\mathcal{C}$ above.

Suppose there is no such integer z. Then $2^{z'} < n < 2^{z'+1}$ for some integer $z' \geq 0$. It is easy to see that S_n is a nondecreasing sequence, therefore

$$S(n) \leq S(2^{z'+1}) = P(z'+1).$$

We then apply our claim $\mathcal{C}$ to see that $P(z'+1) \leq 2(z'+1)$. Finally, we notice that $\log_2 n < z'+1$, therefore $\lceil \log_2 n \rceil = z'+1$. Hence

$$S(n) \leq S(2^{z'+1}) \leq 2(z'+1) = 2\lceil \log_2 n \rceil.$$

∎

Now, consider a call to SearchEndPoint(T, D) in which T covers exactly one error e (which is within D), and every failing test within recursive subcalls covers no errors other than e. The maximum number of tests conducted by this call to SearchEndPoint(T, D) is given by the following lemma.

Lemma 3.4.2 *Consider one call to* SEARCHEND POINT(T, D)*, under the assumptions of Lemma 3.3.1. Let* B, C, T'*, and* T'' *be as defined in* SEARCHENDPOINT(T, D) *(see Algorithm 3.2 on Page 53), and let* e *be the only error covered by* T*. If every failing test defined within this call to* SEARCHEND POINT *(including within its recursive subcalls) covers no errors except for* e*, then* SEARCHENDPOINT(T, D) *conducts at most* $2\lceil \log_2 n \rceil$ *tests, where* $n = |D| \leq k$*.*

Proof: Suppose that the conditions of Lemma 3.3.1 are satisfied, and let e be the only error covered by T. Suppose that every failing test defined within this call to SEARCHEND POINT (including within its recursive subcalls) covers no errors except for e.

Let $n = |D|$, and let $S(n)$ be the maximum number of tests conducted by this call to SEARCHENDPOINT(T, D). It is easy to see that at most one of T', T'' is a failing test, since T'_D and T''_D are disjoint, therefore SEARCHENDPOINT(T, D) makes at most one recursive subcall. Without loss of generality, suppose that T' is a failing test, and SEARCHENDPOINT(T', B) is called. Then $S(n)$ satisfies the recurrence relation given in Lemma 3.4.1, with $S_n = S(n)$. Therefore $S(n) \leq 2\lceil \log_2 n \rceil$. ∎

However, not every call to SEARCHENDPOINT(T, D) will follow the restrictions given in the preceding lemma. Our given $TP(k, 2)$ has up to two errors, so T may cover both of those errors. Furthermore, it is possible for the tests T' and T'' defined within SEARCHENDPOINT to cover distinct errors e' and e'', respectively, even when T to covers only one of e', e''.

Now, consider the set R returned by a call to SEARCHENDPOINT(T, D), and let $f \in R$. Note that f corresponds to an end (f, v) of some faulty interaction, and $\{(f, v)\}$ is covered by T_D (i.e. $T_f = v$ and $f \in D$). Consequently, $|R| \leq 2$, by Lemma 3.3.1. This bound on the cardinality of R enables the following lemma regarding the maximum number of tests conducted by SEARCHENDPOINT.

Lemma 3.4.3 *Consider one call to* $\textsc{SearchEndPoint}(T, D)$*, under the assumptions of Lemma 3.3.1. Let* B, C, T'*, and* T'' *be as defined in* $\textsc{SearchEndPoint}(T, D)$ *(se Algorithm 3.2 on Page 53). The procedure* $\textsc{SearchEndPoint}(T, D)$ *conducts at most* $4\log_2 n + 2$ *tests, where* $n = |D| \leq k$.

Proof: We begin by showing that if $\textsc{SearchEndPoint}(T, D)$ makes two recursive subcalls $\textsc{SearchEndPoint}(T', B)$, $\textsc{SearchEndPoint}(T'', C)$, then neither of those subcalls, nor any subcalls within them make two recursive calls at any given level of recursion.

Suppose that the conditions of Lemma 3.3.1 are satisfied, and suppose that T' and T'' are both failing tests. Notice that the two subtests T'_D, T''_D are disjoint, therefore any error covered by both T' and T'' is within D', and also covered by T. This contradicts assumption 1 of Lemma 3.3.1, so if T' and T'' are both failing tests, then each of T', T'' covers a distinct error. Let e' and e'' be the distinct errors covered by T' and T'', respectively. Then e' is either within B or across the partition $[B, B']$ of $[1, k]$. In either case, at least one end $\{(b, v)\}$ of e' is within B and covered by T', and thus not covered by T'', since $T''_B = \overline{T_B} = \overline{T'_B}$. Next, notice that every test T^* defined within either $\textsc{SearchEndPoint}(T'', C)$ or one of its recursive subcalls also avoids $\{(b, v)\}$, since $T^*_f = \overline{T_f}$ for all $f \in B$.

Since at most two errors are present in our given $TP(k, 2)$, every failing test defined within $\textsc{SearchEndPoint}(T'', C)$ covers only e''. Therefore, by Lemma 3.4.2, $\textsc{SearchEndPoint}(T'', C)$ conducts at most $2\lceil\log_2 n_C\rceil$ tests, where $n_C = |C|$. Similarly, T' does not cover e'', nor does any test defined within $\textsc{SearchEndPoint}(T', B)$ (including within its recursive subcalls), therefore $\textsc{SearchEndPoint}(T', B)$ conducts at most $2\lceil\log_2 n_B\rceil$ tests, where $n_B = |B|$.

Next, we show that if $\textsc{SearchEndPoint}(T, D)$ does make two recursive calls at the same level of recursion, then it conducts fewer than $4\log_2 n + 2$ tests. Let $S(n)$ be the maximum number of tests conducted by a call to $\textsc{SearchEndPoint}$

that satisfies the conditions of Lemma 3.4.2, and let $V(i)$ be the maximum number of tests conducted by SEARCHENDPOINT(T, D) when it makes two recursive calls at depth i of its search tree.

It is easy to see that $V(i) = 2i + 2S\big(\lceil \frac{n}{2^i} \rceil\big)$, where $1 \le i \le \lceil \log_2 n \rceil$. We apply Lemma 3.4.2 to get:

$$V(i) \le 2i + 4\left\lceil \log_2 \left\lceil \frac{n}{2^i} \right\rceil \right\rceil.$$

If $n = 2^z$ for some integer $z \ge 1$, a few manipulations yield $\big\lceil \log_2 \lceil \frac{n}{2^i} \rceil \big\rceil = \log_2 n - i$. Substituting this into the previous inequality, taking into account that $i \ge 1$, we get:

$$V(i) \le 4 \log_2 n - 2i \le 4 \log_2 n - 2.$$

If n is not a power of two, we have $2^z < n < 2^{z+1}$. In this case, we have $\big\lceil \log_2 \lceil \frac{n}{2^i} \rceil \big\rceil < \log_2 \big(\frac{2^{z+1}}{2^i}\big) = z + 1 - i$. Substituting this into the first inequality, taking into account that $z < \log_2 n$ and $i \ge 1$, we get:

$$V(i) < 4 \log_2 n + 4 - 2i \le 4 \log_2 n + 2.$$

∎

We now analyze LOCATE11S(D, k), a recursive subprocedure of HAS11SORLOOPS. We begin with the case where there is exactly one 11-error within D.

Lemma 3.4.4 *Under the assumptions of Lemma 3.3.7,* LOCATE11S(D, k) *conducts at most* $4 \log_2 k$ *tests when there is only one* 11*-error within D.*

Proof: Assume that there is exactly one error within D. Then at most one of T, T' fails at any given level of recursion within LOCATE11S(D, k).

Suppose that T and T' pass at the top level of the search tree. Then $E = \emptyset = E'$, and LOCATE11S calls LOCATEONEORTWO11SACROSS without making any more recursive calls. This procedure first identifies one 1-end within B, and then one 1-end within C, by making a total of two calls to SEARCHENDPOINT, each of which conducts at most $2\log_2 k$ tests, by Lemma 3.4.2.

Suppose instead that one of T, T' fails. Then LOCATE11S recursively calls itself one or more times prior to calling one of its subprocedures. Let $S(n)$ be the maximum number of tests conducted by LOCATE11S(D, k), where $n = |D|$, not including tests conducted by either of its subprocedures.

Since there is exactly one error within D, at most one of T, T' can fail at any given level of recursion, therefore LOCATE11S calls itself at most once per level of recursion. Then $S(n)$ satisfies the recurrence relation $S(n) = 2 + S\big(\lceil \frac{n}{2} \rceil\big)$, and by Lemma 3.4.1, we have $S(n) \leq 2\lceil \log_2 n \rceil \leq 2\log_2 k$. Therefore LOCATE11S$(D, k)$ conducts at most $2\log_2 k$ tests before it enters a level of recursion where it must do one of the following:

(a) call LOCATEONEORTWO11SACROSS, or

(b) recursively call itself again.

We analyze the options with the fewest tests first. If it selects option (a) directly, then it conducts at most $2\log_2 k$ more tests, as we have already shown. Furthermore, since it called LOCATEONEORTWO11SACROSS, we have $E = \emptyset = E'$, therefore it does not call LOCATEONE11ACROSS, and it returns the set E'' after conducting a total of $4\log_2 k$ tests.

If it selects option (b), it can continue until some depth $i \leq \log_2 k$. At this point, the base case call at depth i returns a set E or E' (say E, without loss of generality) to the previous level $i - 1$. Then $|E| = 1$ and $E' = \emptyset$, and one call to LOCATEONE11ACROSS(E, B^*, C^*, k) is made, for some disjoint sets B^*, C^*. However, there is exactly one error e within D, and it was already found within B^*. Both tests defined within LOCATEONE11ACROSS avoid e, therefore each call to this procedure conducts exactly 2 tests when there is no error across $[B^*, C^*]$. At the $(i-1)$th level of recursion, $E'' = \emptyset$, so $E \cup E' \cup E'' = E$ is returned to the previous depth $i - 2$. At this level, we again have $|E| = 1$ and $E' = \emptyset$, so the process of calling LocateOne11Across can repeat itself once at each level of recursion. The depth of the search tree is at most $\log_2 k$, therefore at most $2\log_2 k$ tests are conducted, in total, by the calls to LocateOne11Across. Therefore the total number of tests conducted by Locate11s(D, k) is still $4\log_2 k$ when there is only one 11-error within D. ▮

We use the preceding result to see the maximum number of tests conducted by Locate11s(D, k), even when there are two errors within D.

Lemma 3.4.5 *Under the assumptions of Lemma 3.3.7,* Locate11s(D, k) *conducts at most* $2 + 10\log k$ *tests.*

Proof: Suppose that T and T' both fail. Then Locate11s(D, k) calls both Locate11s(B, k) and Locate11s(C, k). Since B and C are disjoint, we apply Lemma 3.4.4 to each instance to see that the total number of tests in this instance is at most $2 + 2(4\log k) = 2 + 8\log k$.

Suppose instead that exactly one of T, T' fails at one or more levels of recursion prior to both failing. Then at most $2\log_2 k$ tests are conducted prior to Locate11s being 'split' into two different instances In this case, the total number of tests is at most $2 + 10\log_2 k$.

Alternatively, suppose that exactly one of T and T' fails (say, T), and one error is found within B, in at most $4\log_2 k$ tests, and then stored in E. As in proof of the previous lemma, the depth of the search tree is at most $\log_2 k$, therefore LocateOne11Across is called at most $\log_2 k$ times. Suppose that a call to this procedure at depth i in the search tree of Locate11s(D, k) yields a failing test.

Then LocateOne11Across calls SearchEndPoint up to twice, and each call to SearchEndPoint identifies an end of the error not already stored in E in at most $2\log_2 k$ tests, by Lemma 3.4.2. Then the error-set $E \cup E' \cup E''$ returned from depth i to the previous depth has cardinality two, and no further calls to LocateOne11Across are made. However, up to $\log_2 k - 1$ calls to LocateOne11Across may have been made prior to the final one, and in each of those calls, two tests were conducted. In this case, the total number of tests conducted is at most $4\log_2 k + 4\log_2 k + 2\log_2 k = 10\log_2 k$.

Therefore, LocateOne11Across conducts at most $2 + 10\log_2 k$ tests. ∎

Next, we analyze the remaining subprocedures for Case 1.

Lemma 3.4.6 *Under the assumptions of Lemma 3.3.8, the maximum number of tests conducted by each of the following is:*

A1No11inAprime*:* $5 + 6\log_2 k$

A1Locate11inAprime*:* $2 + 6\log_2 k$

A2No11inAprime*:* $2\log_2 k$*.*

Proof: A1No11inAprime calls FindPassingTest, which conducts at most $2\log_2 k + 2$ tests, by Lemma 3.2.12. If there is no passing test, then we proceed no further. Otherwise, it calls SearchEndPoint once in order to find two endpoints, which requires at most $4\log_2 k + 2$ tests. One further test, $T^{(b)}$, is conducted, for a total of $6\log_2 k + 5$ tests.

Next, we see that A1Locate11inAprime calls Locate11s once, to identify the sole 11-error within D, which requires at most $4\log_2 k$ tests, by Lemma 3.4.4. It then conducts two further tests $T^{(b)}, T^{(c)}$ and potentially calls SearchEndPoint once, which conducts at most $2\log_2 k$ tests, by Lemma 3.4.2. Therefore at most $2 + 6\log_2 k$ tests are conducted by A1Locate11inAprime.

Finally, A2No11inAprime calls SearchEndPoint once to identify one end-point, thus requiring at most $2\log_2 k$ tests. ∎

We now analyze the number of tests required in Case 1.

Lemma 3.4.7 *Under the assumptions of Lemma 3.3.8, at most* $2 + 10\log_2 k$ *tests are conducted by the procedure* Has11sOrLoops.

Proof: Our procedure for Case 1, Has11sOrLoops, executes only one of its subprocedures (see Algorithm 3.5).

If $A = \emptyset$, then Has11sOrLoops calls Locate11s, which conducts at most $2 + 10\log_2 k$ tests, by Lemma 3.4.5.

Otherwise, if $|A| = 1$, then this procedure conducts one additional test, and calls either A1No11inAprime or A1Locate11inAprime. The former conducts at most $5 + 6\log_2 k$ tests, and the latter conducts at most $2 + 6\log_2 k$ tests, by Lemma 3.4.6. Therefore if $|A| = 1$, then this procedure conducts at most $6 + 6\log_2 k$ tests.

Finally, if $|A| = 2$, then this procedure conducts at most two more tests prior to calling A2No11inAprime, which conducts at most $2\log_2 k$ tests, also by Lemma 3.4.6. Therefore if $|A| = 2$, then this procedure conducts at most $2 + 2\log_2 k$ tests.

Clearly the maximum number of tests conducted by this procedure is $2+10\log_2 k$, which occurs when $A = \emptyset$. ∎

Next, we analyze the number of tests required in Case 0.

Lemma 3.4.8 *Under the assumptions of Lemma 3.3.6, at most* $2 + 4\log_2 k$ *tests are conducted by the procedure* No11sNorLoops.

Proof: If we are in Subcase 0.1, this procedure identifies up to two endpoints via one call to SearchEndPoint, which requires at most $4\log_2 k + 2$ tests, by Lemma 3.4.3. Otherwise, we are in Subcase 0.2, and we call Locate01 twice, each time conducting one test, and subsequently calling SearchEndPoint in order to identify one endpoint, for a total of $1 + 2\log_2 k$ tests, by Lemma 3.4.2. Therefore the total number of tests conducted is at most $2 + 4\log_2 k$. ∎

We conclude by analyzing the number of tests executed by the main procedure, LocateAllErrors.

Theorem 3.4.9 *Under the assumptions of Theorem 3.3.5,* LocateAllErrors *executes at most* $2\big(1+o(1)\big)(\log k)^2 + \mathcal{O}(\log k)$ *tests.*

Proof: Step 0 calls FindPassingTest, which requires at most $2\log_2 k + 2$ tests. If we find no passing test, the algorithm terminates.

Step 1 calls SearchEndPoint up to $\big(1+o(1)\big)\log_2 k$ times; that is, at most once for each row in the covering array $\mathcal{C}$. Several calls may return the same singleton set, and each iteration of SearchEndPoint in which a singleton set is returned, requires at most $2\log_2 k$ tests, by Lemma 3.4.2. If at any point, the SearchEndPoint procedure returns a set containing two factors, we exit the loop and conclude Step 1. The iteration in which this happened would require at most $4\log_2 k + 2$ tests, by Lemma 3.4.3. Therefore Step 1 conducts at most $2\big(1+o(1)\big)(\log_2 k)^2 + \mathcal{O}(\log_2 k)$ tests.

In Step 2, we call either No11sNorLoops or Has11sOrLoops, both of which conduct $\mathcal{O}(\log k)$ tests, by Lemmas 3.4.8 and 3.4.7.

We see that the total number of tests conducted by all three steps is at most $2\big(1+o(1)\big)(\log k)^2 + \mathcal{O}(\log k)$. ∎

To summarize, the performance of LocateAllErrors is nearly identical to the performance of DiscoverEdges when at most two errors are present. However, our algorithm has a worst-case runnning time of $2\big(1+o(1)\big)(\log k)^2+\mathcal{O}(\log k)$ tests, which is the expected running time of the algorithm of Martínez et al. [32]. Furthermore, our algorithm does not require the assumption that errors are locatable.

Chapter 4

Combinatorial Group Testing and Error Location with Safe Values

Suppose we have a set of k items, and we wish to identify which items are defective by testing subsets of items; these subsets are called **groups** or **pools**. A **negative pool** contains no defective items, and a **positive pool** contains at least one defective item. This is known as **group testing**, and it has been studied since World War II, when Robert Dorfman and David Rosenblatt developed the field in order to efficiently screen pools of blood samples for syphilis.

There are two main categories of group testing. In **probabilistic group testing (PGT)**, defective items follow a particular probability distribution, and the goal is to minimize the expected number of tests required to identify all defective items. We focus on **combinatorial group testing (CGT)**, where defective items are not assumed to follow any particular probability distribution. For consistency with our earlier terminology, we refer to a negative pool of items as a **passing set**, and we refer to a positive pool of items as a **failing set**.

Now, we discuss a more general CGT problem. A failing set may contain no defective items; rather, it may fail due to the interaction among (some of) its items.

In the context of group testing, we define faulty interactions as follows.

Definition 4.0.10 *Let S be a set containing k items, and let $S' \subseteq S$ be a failing set, such that $|S'| = t$ for some $t \in [1, k]$. If S' contains no proper subset S'' that also fails, then S' is a **faulty interaction (or error) of strength** t. If S contains only errors of strength t, then a CGT algorithm which correctly returns all faulty interactions in S is a **strength-t CGT algorithm**. Similarly, if S contains errors of strengths up to t, then a CGT algorithm which correctly returns all faulty interactions in S is a **strength-$\bar{t}$ CGT algorithm**.*

Our goal is to identify all faulty interactions in a given set while minimizing the number of tests required.

In this chapter, we summarize existing results for combinatorial group testing. The algorithms which conduct the fewest tests relative to k, for strengths 1 and up to 2, are adaptive (see Definition 1.1.5). At this time of writing, the only known adaptive algorithm for a strength greater than two is Algorithm 5.1, which is a new strength-$\overline{3}$ CGT algorithm.

We briefly summarize the nonadaptive method which conducts the fewest tests relative to k. In Chapter 5, we compare our new algorithm to the aforementioned nonadaptive method. We remark here on a key advantage of an adaptive algorithm over a nonadaptive one: the latter requires knowledge of an upper bound on the number of faulty interactions, while the former does not!

For a book that covers nonadaptive testing, variations on the testing problem, and the history of group testing, see Du and Hwang [17].

In addition to summarizing relevant group testing algorithms, we show how these methods can be applied to a testing problem when safe values are known. In Section 4.1, we show how group testing can be applied to a testing problem where all errors are of strength 1. In Section 4.2, we give an adaptive algorithm for combinatorial group testing where all faulty interactions have strength at most 2. In Section 4.3, we show how the strength-$\overline{2}$ CGT algorithm can be applied to testing problems where each faulty interaction has strength up to 2. The algorithms presented in Sections 4.2 and 4.3 are adapted to our notation; the original versions are due to Martínez et al. [32].

4.1 Pointwise Group Testing and Strength-1 ELAs with Safe Values

Consider a set S containing k items, one of which is defective. It is well-known that we can easily identify the defective item in at most $\lceil \log_2 k \rceil$ tests by using a simple binary search, which is sometimes referred to as **binary splitting** in the group testing literature.

More generally, suppose that S contains d defective items. We can identify them in at most $d\lceil \log_2 k \rceil$ tests by conducting d binary searches, where each binary search identifies one defective item, which we subsequently remove from S. We note here that this process can be used even if d is not known in advance because S will be a passing set once all of its defective items have been removed.

We now show how a strength-1 CGT algorithm can be used to adaptively build a strength-1 ELA for the graph associated with a testing problem that has safe values (see Definition 2.2.10). Consider a $TP\big(k, (g_1, g_2,\ ..., g_k)\big)$ which has safe values $s_1, s_2, ..., s_k$ and contains only strength-1 errors. Suppose that T is a failing test. Then identifying all strength-1 errors covered by T is equivalent to identifying all defective items in the set $S = \{(f, T_f) : f \in [1, k]\}$. Furthermore, if we know that all defective items are in a subset $S' \subseteq S$ such that $S' = \{(f, T_f) : f \in A\}$ for some $A \subseteq [1, k]$, then those items are also covered by the test T' defined by

$$T'_f = \begin{cases} T_f & \text{if } f \in A \\ s_f & \text{otherwise} \end{cases}$$

Therefore, if there are d' strength-1 errors covered by T, they can be identified by at most $d'\lceil \log_2 k \rceil$ tests, via d' iterations of a binary search procedure.

Conversely, if we have a set of k items, finding the defective ones among them is equivalent to identifying the strength-1 errors in a $TP(k, 2)$, where each item corresponds to a pair $(f, 1)$ for $f \in [1, k]$, $T = 1^k$ is a failing test, and $s_1 = s_2 = ... = s_k = 0$ are safe values.

We remark here that, given any $TP\big(k, (g_1, g_2, ..., g_k\big)$ with safe values, we can relabel values in each factor so that $s_1 = s_2 = ... = s_k = 0$, without loss of generality. This leads us to the following definition.

Definition 4.1.1 *Let T be a test that can be applied to a $TP\big(k, (g_1, g_2, ..., g_k)\big)$. The unique set $C_T = \{f \in [1, k] : T_f \neq 0\}$ is called the **set (of factors) corresponding to** T.*

Given a set of factors $A \subseteq [1, k]$ and a test T such that $A \subseteq C_T$, the following procedure is used to translate between the two testing contexts, CGT and ELAs with Safe Values, by matching the factor-set A to a test T' such that $C_{T'} = A$.

Algorithm 4.1 Precondition: T is a test for a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ which has safe values $s_1 = s_2 = ... = s_k = 0$, and $A \subseteq C_T \subseteq [1, k]$.
This procedure matches the set A to a test T' such that $A = C_{T'}$; it returns fail if T' fails, and it returns pass, otherwise.

```
procedure TestSet($A, T$)
    return Test($T'$) where $T'$ is defined by: $T'_f \leftarrow \begin{cases} T_f & \text{if } f \in A \\ 0 & \text{otherwise} \end{cases}$
```

Now, consider a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ with safe values $s_1 = s_2 = ... = s_k = 0$, and let $g = \max\{g_1, g_2, ..., g_k\}$. We can build a set of $g - 1$ tests such that every possible strength-1 error in the $TP\big(k, (g_1, g_2, ..., g_k)\big)$ is covered by exactly one of those tests. Indeed, consider the test $T(i)$, $i \in [1, g - 1]$, defined by

$$T(i)_f = \begin{cases} i & \text{if } i \leq g_f - 1 \\ 0 & \text{otherwise} \end{cases}$$

It is easy to see that the strength-1 interaction $\{(f, v)\}$, with $f \in [1, k]$ and $v \in [1, g_f - 1]$, is covered by $T(v)$. Therefore, we can identify all errors in our given testing problem by running the procedure equivalent to CGT for each test $T(i)$, $i \in [1, g - 1]$. The pseudocode for strength-1 error location with safe values via strength-1 CGT is given as Algorithm 4.2 on Page 95.

We now analyze the maximum number of tests conducted by Algorithm 4.2.

Theorem 4.1.2 *Let $k \geq 2$, and $g_i \geq 2$ for all $i \in [1, k]$. Consider a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ that has safe values $s_1, s_2, ..., s_k$ and contains only strength-1 errors. Let $g = \max\{g_1, g_2, ..., g_k\}$ be fixed, and let d be the number of errors present.*

Then ErrorLocateWithSafeValues$\big(k, (s_1, s_2, ..., s_k), (g_1, g_2, ..., g_k)\big)$ *conducts at most $g - 1 + d\lceil \log_2 k \rceil$ tests.*

Proof: Let d_i be the number of errors covered by the test $T(i)$, $i \in [1, g - 1]$ defined within Algorithm 4.2. By construction, the sets of errors covered by each test are disjoint from each other, so $d = \sum_{i=1}^{g-1} d_i$.

By the discussion preceding this theorem, LocateErrorsInSet$\big(A, T(i)\big)$ conducts one test $T(i)$, plus $d_i\lceil \log_2 k \rceil$ tests whenever $T(i)$ is a failing test.

Hence, the total number of tests conducted over all $g-1$ iterations of the for-loop in ErrorLocateWithSafeValues is at most

$$\sum_{i=1}^{g-1} \big(d_i \lceil \log_2 k \rceil + 1\big) = d\lceil \log_2 k \rceil + g - 1.$$

∎

4.2 Pairwise Group Testing

In this section, we consider group testing with faulty interactions of strengths up to 2. Note that the following definition is applicable to faulty interactions of strengths up to any fixed integer t.

Definition 4.2.1 *We say that an error (or faulty interaction) e is* ***within*** *a set of factors $A \subseteq [1,k]$ if, for all $(f,v) \in e$, we have $f \in A$. Let $A_1, A_2, ..., A_t$ be pairwise disjoint sets such that $\cup_{i=1}^{t} A_i = A$. We say that $[A_1, A_2, ..., A_t]$ is a* ***partition*** *of A, and we call $A_1, A_2, ..., A_t$ the* ***parts*** *of the partition. We say that an error e' is* ***across*** *the partition $[A_1, A_2, ..., A_t]$ if $\nexists i \in [1,t]$ such that e' is within A_i, but e' is within a union of two or more distinct parts of $[A_1, A_2, ..., A_t]$.*

In [1], Andreae studied the problem of searching for a single strength-2 faulty interaction, in the context of searching for a specific edge e in a graph. Let G be a graph with k vertices and m edges. Andreae showed that at most $\lceil 2\log_2 m \rceil$ tests are required to identify e. Since $m \leq \binom{k}{2}$, at most $\lceil 4\log_2 k \rceil$ tests are required if the edges of G are unknown.

More recently, Bouvel et al. [4] conducted a survey on graph reconstruction algorithms. They analyzed particular classes of graphs, and determined that at most $k\log_2 k$ tests are required to identify all of the unknown edges in a Hamiltonian cycle.

Algorithm 4.2 [32] Precondition: $TP\big(k,(g_1,g_2,...,g_k)\big)$ has safe values $s_1,s_2,...,s_k$ ($k \geq 2$ is an integer), every error has strength 1, and $g = \max\{g_1,g_2,...,g_k\}$ is fixed. This algorithm returns the set E of all errors in our $TP\big(k,(g_1,g_2,...,g_k)\big)$.

```
procedure ErrorLocateWithSafeValues($k, (s_1, s_2, ..., s_k), (g_1, g_2, ..., g_k)$)
    ▷ Relabel values in each factor so that $s_1 = s_2 = ... = s_k = 0$.
    $S \leftarrow \emptyset$
    for each factor $f \in [1,k]$ such that $s_f \neq 0$ do
        $S \leftarrow S \cup \{f\}$, and swap labels $(f,0)$ and $(f,s_f)$.
    Define $T(i)$ by $T(i)_f = \begin{cases} i & \text{if } i \leq g_f - 1 \\ 0 & \text{otherwise} \end{cases}$

    ▷ Identify the defective items via CGT.
    $E \leftarrow \emptyset$, and let $g = \max\{g_1, g_2, ..., g_k\}$.
    for $i \leftarrow 1$ to $g-1$ do
        $E' \leftarrow$ LocateErrorsInSet$\big(C_{T(i)}, T(i)\big)$
        for $f \in E'$ do $E \leftarrow E \cup \big\{\{(f, T(i)_f)\}\big\}$
    ▷ Restore the original labels to every factor whose label we previously changed.
    for $f \in S$ do swap labels $(f,0)$ and $(f,s_f)$.
    return $E$

▷ Precondition: All errors covered by $T$ are elements of $\big\{\{(f,T_f)\} : f \in A \subseteq C_T\big\}$.
▷ This procedure returns the set $\big\{f \in A : \{(f,T_f)\}$ is an error$\big\}$.
procedure LocateErrorsInSet($A, T$)
    if TestSet($A, T$) = fail then
        $[a, X] \leftarrow$ BinarySearch($A, T$)
        return $\{a\} \cup$ LocateErrorsInSet$\big(A \setminus (\{a\} \cup X), T\big)$
    else return $\emptyset$

▷ Precondition: $T$ is a failing test, and $D \subseteq C_T$.
▷ This procedure returns one factor $a$ corresponding to a defective item $(a, T_a)$, and
▷ a set $X$ of factors corresponding to nondefective items.
procedure BinarySearch($D, T$)
    $X \leftarrow \emptyset$
    while $|D| > 1$ do
        Partition $D$ as evenly as possible into parts $D', D''$.
        if TestSet($D', T$) = fail then $D \leftarrow D'$
        else $X \leftarrow X \cup D'$, and $D \leftarrow D''$
    Let $D = \{a\}$
    return $[a, X]$.
```

Their goal was to reconstruct an unknown graph $G(V, E)$ by discovering its edges via a process called graph testing, which we define (more generally) below.

Definition 4.2.2 ***Hypergraph Testing:*** *Given a hypergraph $H(V, E)$, a test conducted on a subset $V' \subseteq V$ fails if $H[V']$ contains a hyperedge, and passes otherwise. If H is simply a graph (i.e. for every $e \in E$, $|e| \leq 2$), we simply refer to this process as* ***Graph Testing****.*

It is easy to see that hypergraph testing is equivalent to combinatorial group testing; vertices correspond to items, and hyperedges of cardinality t correspond to faulty interactions of strength t.

In [32], Martínez et al. give an adaptive, strength-$\overline{2}$ ELA-building algorithm for a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ which has safe values $s_1, s_2, ..., s_k$, supposing that no error in the given system is of strength greater than two, and there are at most d errors. We observe that their algorithm consists of two strength-$\overline{2}$ algorithms: a CGT algorithm that requires $\mathcal{O}\big(d \log_2 k + d^2\big)$ tests, and an ELA-building algorithm that iteratively applies the aforementioned CGT algorithm $\mathcal{O}(\log_2 k)$ times, requiring a total of $\mathcal{O}\big(d(\log_2 k)^2 + d^2 \log_2 k\big)$ tests.

In this section, we present their CGT algorithm, LocateErrorsInTest [32, Section 5], adapted to our notation. Our version, called LocateErrorsInSet, is presented here as Algorithm 4.3. Let T be a test applied to a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ that has safe values, and errors of strength up to 2. LocateErrorsInSet(A, T) requires that $A \subseteq C_T$ (see Definition 4.1.1). Since the test T remains fixed within LocateErrorsInSet(A, T), there is a one-to-one correspondence between the factors $f \in A$ and the items (f, T_f) covered by T. Therefore, in the context of Combinatorial Group Testing, items can be labeled by their factors directly, and the distinction between an item-set and a factor-set becomes unimportant. However, for the sake of notational consistency, we describe this algorithm (and, in Chapter 5, its strength-$\overline{3}$ generalization, Algorithm 5.1) as if it is being applied in the context of a

$TP\big(k, (g_1, g_2, ..., g_k)\big)$.

When T is a failing test, LOCATEERRORSINSET(A, T) works as follows. If $A = \{a\}$, then $\{(a, T_a)\}$ is a strength-1 error. Otherwise $|A| \geq 2$, and we partition A as evenly as possible into two sets A' and A''. Since the strength of each error covered by T is at most two, every error is either within one of A', A'', or is across the partition $[A', A'']$.

If either A' or A'' is a set of factors corresponding to a failing set of items, we recursively call LOCATEERRORSINSET to identify the error sets E' and E'' of errors within A' and A'', respectively. We then call ACROSSLOCATE in order to identify every error across $[A', A'']$ while avoiding the errors already found within each of A', A'' (which are already stored in E' and E'').

By Definition 4.0.10, strength-2 errors cannot contain strength-1 errors as subsets, so ACROSSLOCATE begins by considering only subsets $B' \subseteq A'$ and $B'' \subseteq A''$ such that each of B', B'' contains all factors in A', A'' except for those that correspond to strength-1 errors. It then further partitions each of B', B'' such that no error is contained within any part of either B' or B'', that is, every error within B' is across the partition $[B'_1, B'_2, \ldots, B'_{c'}]$, and every error within B'' is across the partition $[B''_1, B''_2, \ldots, B''_{c''}]$. These partitions can be determined in the following way. Let $G = (V, \mathcal{E})$ be a graph such that $V = [1, k]$ and every edge in $\mathcal{E}$ is a set of factors corresponding to an error. Then a proper colouring of $G[B']$ and $G[B'']$ determines the partitions mentioned above as follows.

It is well-known that a greedy algorithm can assign a proper $\big(\Delta(H)+1\big)$-colouring to a graph H. Since $\Delta(H) \leq d$ (where d is the maximum number of errors in our testing problem) for $H \in \big\{G[B'], G[B'']\big\}$, we have a $(d+1)$-colouring of $G[B']$ with colour classes $B'_1, B'_2, \ldots, B'_{c'}$, and a $(d+1)$-colouring of $G[B'']$ with colour classes $B''_1, B''_2, \ldots, B''_{c''}$. Clearly, no edges of $G[B']$ and $G[B'']$ lie within a colour class, and $c', c'' \leq d + 1$.

Algorithm 4.3 [32] Martínez et al.'s strength-$\overline{2}$ CGT algorithm.
Precondition: T is a failing test that covers errors of strengths up to 2, and $A \subseteq C_T$. Let TestSet(A) = TestSet(A, T). This algorithm returns the set of all factor-sets corresponding to the errors within A.

```
procedure LocateErrorsInSet($A, T$)
    if $|A| = 1$ then return $\{A\}$
    else
        Partition $A$ into $[A', A'']$ as evenly as possible.
        if TestSet($A'$) = fail then $E' \leftarrow$ LocateErrorsInSet($A'$)
        if TestSet($A''$) = fail then $E'' \leftarrow$ LocateErrorsInSet($A''$)
        $E''' \leftarrow$ AcrossLocate($A', A'', E', E''$)
        return $E' \cup E'' \cup E'''$

▷ Precondition: $E'$ is the set of all errors within $A'$, and $E''$ is the set of all
▷ errors within $A''$. This subprocedure returns the set $E'''$ of all errors
▷ that are across the partition $[A', A'']$ of $A' \cup A''$.
procedure AcrossLocate($A', A'', E', E''$)
    Let $B' = A' \setminus S'$, where $S' = \{x \in A' : \{x\} \in E'\}$.
    Partition $B'$ as $[B'_1, B'_2, \ldots, B'_{c'}]$ such that $\forall e' \in E', \forall i \in [1, c']$,
    $e'$ is not within $B'_i$ .
    Let $B'' = A'' \setminus S''$, where $S'' = \{y \in A'' : \{y\} \in E''\}$.
    Partition $B''$ as $[B''_1, B''_2, \ldots, B'_{c''}]$ such that $\forall e'' \in E'', \forall j \in [1, c'']$,
    $e''$ is not within $B''_j$.
    $E''' \leftarrow \emptyset$
    for $i \leftarrow 1$ to $c'$ do
        for $j \leftarrow 1$ to $c''$ do
            if TestSet($B'_i \cup B''_j$) = fail then
                $E''' \leftarrow E''' \cup$ AcrossLocateAux($B'_i, B''_j$)
    return $E'''$

▷ Precondition: TestSet($A \cup B$) = fail, neither $A$ nor $B$ has an error within it.
procedure AcrossLocateAux($A, B$)
    if $|A| = |B| = 1$ then return $\{A \cup B\}$
    else
        ▷ Partition the larger set first.
        if $|A| \geq |B|$ then $[C, D] \leftarrow [A, B]$
        else $[C, D] \leftarrow [B, A]$

        $E' \leftarrow E'' \leftarrow \emptyset$
        Partition $C$ into $[C', C'']$ as evenly as possible.
        if TestSet($C' \cup D$) = fail then $E' \leftarrow$ AcrossLocateAux($C', D$)
        if TestSet($C'' \cup D$) = fail then $E' \leftarrow$ AcrossLocateAux($C'', D$)
        return $E' \cup E''$
```

No part of either $[B'_1, B'_2, \ldots, B'_{c'}]$ or $[B''_1, B''_2, \ldots, B''_{c''}]$ contains an error, and all errors across these partitions are known. Therefore the remaining errors must be across partitions $[B'_i, B''_j]$ of sets $B'_i \cup B''_j$, where $i \in [1, c']$ and $j \in [1, c'']$. For each set $B'_i \cup B''_j$ corresponding to a failing set of items, we call the procedure AcrossLocateAux to identify all of the errors that are across $[B'_i, B''_j]$.

If B'_i and B''_j are both singleton sets, then the strength-2 error across $[B'_i, B''_j]$ must correspond to the factor-set $B'_i \cup B''_j$, which AcrossLocateAux then returns. Otherwise we let C and D be the larger and smaller of the given sets, respectively, and we partition C as evenly as possible into $[C', C'']$. Since no errors are within C, each error within $C \cup D$ must be across one of the following partitions: $[C', D]$ of $C' \cup D$, or $[C'', D]$ of $C'' \cup D$.

If $C' \cup D$ is a set of factors corresponding to a failing set of items, we identify each of the errors across its partition $[C', D]$ by recursively calling AcrossLocateAux. We proceed similarly if $C'' \cup D$ corresponds to a failing set of items; we identify each of the errors across $[C'', D]$ by recursively calling AcrossLocateAux.

We omit the proof of correctness here, since we prove the correctness of a more generalized version of this algorithm in the next chapter.

In Section 4.1, we gave a strength-1 CGT algorithm which is logarithmic in k and linear in d. This strength-$\overline{2}$ generalization is also logarithmic in k.

Theorem 4.2.3 *Algorithm 4.3 performs $\mathcal{O}\big(d \log_2 k + d^2\big)$ tests.*

For a detailed proof of the above theorem, see [32, Section 5].

4.3 Strength-$\overline{2}$ ELAs with Safe Values via CGT

The strength-$\overline{2}$ version of ErrorLocateWithSafeValues builds a strength-2 MCA $\mathcal{M}$ (by, for instance, the method of Bryce and Colbourn [5]). By Definition 1.3.1, the failing tests corresponding to the rows of $\mathcal{M}$ collectively cover all errors in the given

$TP\big(k,(g_1,g_2,...,g_k)\big)$. For every such failing test T, the algorithm calls the strength-$\overline{2}$ CGT algorithm, LocateErrorsInSet(C_T, T), which identifies the errors involving factors in C_T (see Definition 4.1.1).

We remark here on a slight improvement over the original version. Since the first for-loop of Algorithm 4.4 relabels all safe values to be zeroes, we need an MCA whose rows collectively cover all strength-2 interactions except for those containing $(f,0)$ where $f \in [1,k]$. We possibly reduce the number of tests conducted by the ErrorLocateWithSafeValues procedure by using an MCA with alphabets of decreased cardinality. We accomplish this by constructing an $MCA\big(N;2,k,(g_1 - 1, g_2-1,...,g_k-1)\big)$ instead of an $MCA\big(N;2,k,(g_1,g_2,...,g_k)\big)$, and then for each row (test) T of the array, we assign the value of $g_f - 1$ to T_f whenever $T_f = 0$.

Corollary 4.3.1 *Let* $g = \max\{g_1,g_2,...,g_k\}$ *be fixed. Then Algorithm 4.4 performs* $\mathcal{O}\big(d(\log_2 k)^2 + d^2\log_2 k\big)$ *tests.*

Proof: There exists an $MCA\big(N;2,k,(g_1-1,g_2-1,...,g_k-1)\big)$ with $N \sim \frac{g-1}{2}\log_2 k$ as $k \to \infty$, but this bound is not constructive (see [20, 12]). However, the method of Bryce and Colbourn [5] builds an $MCA\big(N;2,k,(g_1-1,g_2-1,...,g_k-1)\big)$ with $N \leq 2(g-1)^2\ln k + \mathcal{O}(1)$. Since Algorithm 4.4 calls LocateErrorsInSet N times, and $N = \mathcal{O}(\log_2 k)$, the result follows directly from Theorem 4.2.3. ∎

4.4 Higher-Strength Nonadaptive Group Testing

Higher-strength combinatorial group testing is introduced by Torney in the context of sets pooling designs [40]. He gives a brief description of a higher-strength CGT algorithm for the special case when there is only one error. Given a set S of items, sequentially remove elements of S one-at-a-time, and after each removal, if S becomes

Algorithm 4.4 [32] Adaptation of Martínez et al.'s strength-$\overline{2}$ ELA algorithm. Precondition: $TP(k,(g_1,g_2,...,g_k))$ has safe values $s_1,s_2,...,s_k$ ($k \geq 2$), the set of all errors is independent, each error has strength at most 2, and $g = \max\{g_1,g_2,...,g_k\}$ is fixed. This algorithm returns the set E of all errors in the $TP\big(k,(g_1,g_2,...,g_k)\big)$.

```
procedure ErrorLocateWithSafeValues($k, (s_1, s_2, ..., s_k), (g_1, g_2, ..., g_k)$)
    ▷ Relabel values in each factor so that $s_1 = s_2 = ... = s_k = 0$.
    $S \leftarrow \emptyset$
    for each factor $f \in [1,k]$ such that $s_f \neq 0$ do
        $S \leftarrow S \cup \{f\}$, and swap labels $(f,0)$ and $(f,s_f)$.

    ▷ Identify the defective items via CGT.
    $E \leftarrow \emptyset$, and let $g = \max\{g_1, g_2, ..., g_k\}$
    Let $\mathcal{M}$ be an $MCA(N; 2, k, (g_1-1, g_2-1, ..., g_k-1))$.
    Denote the $i$th row (test) of $\mathcal{M}$ by $T(i) = T(i)_1 T(i)_2 ... T(i)_k$.
    for $i \leftarrow 1$ to $N$ do
        for all $f \in [1,k]$ such that $T(i)_f = 0$ do $T(i)_f \leftarrow g_f - 1$
        if Test$\big(T(i)\big)$ = fail then
            $E' \leftarrow$ LocateErrorsInSet$\big(C_{T(i)}, T(i)\big)$
            for $D \in E'$ do $E \leftarrow E \cup \big\{\{(f, T(i)_f) : f \in D\}\big\}$

    ▷ Restore the original labels to every factor whose label we previously changed.
    for $f \in S$ do swap labels $(f,0)$ and $(f,s_f)$.
    return $E$
```

a passing set, conclude that the removed item must be involved in the higher-strength error, replace it in S, and continue. This is a very restrictive case, and some progress has been made since then.

Higher-strength group testing is often referred to as hypergraph testing (see Definition 4.2.2) or **group testing for complexes**, with the latter term being motivated by applications in molecular and cellular biology (see [40]). Chen et al. have remarked that, in such applications, a test may correspond to an experiment that occurs over the course of hours or, worse yet, days [9]. In such situations, nonadaptive testing has a clear advantage since it allows for tests to be conducted in parallel.

All higher-strength CGT algorithms considered prior to this thesis are nonadaptive, and they require advance knowledge of an upper bound on d, the number of errors in the given set. In [19], Gao et al. describe how to identify all hyperedges in an unknown hypergraph H containing up to d hyperedges, by constructing a special type of incidence matrix for H.

Definition 4.4.1 *[19] Let H be a hypergraph with k vertices labeled $1, ..., k$ and at most $d \geq 1$ hyperedges. Let M be a binary matrix with k columns.*

*We say that M is $d(H)$**-disjunct** if for any set of $d+1$ potential hyperedges $\{e_0, e_1, ..., e_d\}$ (i.e. subsets of $[1, k]$) of H there exists at least one row (test) T of M such that $T_v = 1$ for all $v \in e_0$ and $T_v = 0$ for some v in each of $e_1, e_2, ..., e_d$.*

A $d(H)$-disjunct matrix for H is used to identify d or fewer hyperedges in H in the following way: every subset of vertices corresponding to a passing test is a non-edge, and the remaining maximal subsets of $V(H)$ are all edges (see Definition 4.2.2).

Theorem 4.4.2 *[9, Section 3] Let $d, k,$ and t be positive integers, and let H be a hypergraph with k vertices and at most d hyperedges of cardinalities up to t, such that $d + t \leq k$. Then there exists a $d(H)$-disjunct matrix with at most N rows, such that*

$$N < \left(\frac{d+t}{t}\right)^t \left(\frac{d+t}{d}\right)^d \left(1 + (d+t)\left(1 + \ln\left(1 + \frac{k}{d+t}\right)\right)\right)$$

Other bounds on $d(H)$-disjunct matrices are given in [8, 19], but they are greater than the above bound when $\ln k$ is relatively large with respect to d and t, since each bound includes a factor of $(\ln k)^{t+1}$ in addition to exponential factors involving d and t.

Macula and Popyack [30] give a construction of a $d(H)$-disjunct matrix whose number of rows is logarithmic in k, and has a factor of td^t. However, their method is probabilistic, and identifies an expected number $(1 - o(1))d$ faulty interactions of strengths at most t. Furthermore, they require two additional assumptions on the structure of the errors: each error has cardinality exactly t, and for any pair of errors e, e', $|e \setminus e'| = h$ for some fixed integer $h \in [1, t]$.

In the next chapter, we generalize the strength-$\overline{2}$ CGT algorithm of Martínez et al. [32] (Algorithm 4.3) to strength $\overline{3}$. We show that it identifies all errors in a given set, without any prior knowledge of the number of errors present. We also show that it requires fewer tests than the current best nonadaptive method of Chen et al. [9]. We also generalize error location with safe values (Algorithm 4.4) to strength $\overline{3}$.

Chapter 5

Strength-$\overline{3}$ Group Testing and Error Location

In the preceding chapter, we introduced combinatorial group testing (CGT), and we explained how it can be used to identify all faulty interactions in a given set of k items. We presented the known adaptive algorithms for pointwise and pairwise combinatorial group testing, which are logarithmic in k. We also showed how error location with safe values, as found in Martínez et al. [32], relates to CGT. We then summarized existing nonadaptive methods for CGT with strengths greater than one; we also summarized the performance of each algorithm in terms of the maximum number of tests conducted with respect to k, the number of items in the given set, and d, the number of faulty interactions among the k items. We note that for strengths greater than two, the method with the fewest tests with respect to k is the nonadaptive, disjunct matrix method of Chen et al. (see Theorem 4.4.2).

We begin this chapter by giving a new adaptive algorithm for strength-$\overline{3}$ CGT in Section 5.1. We show that it conducts $\mathcal{O}\big(d^3 \ln k\big)$ tests, fewer than the current best nonadaptive algorithm of Chen et al. [9], which conducts $\mathcal{O}\Big(d^4\big(1+\frac{3}{d}\big)^d \ln\big(\frac{k}{d}\big)\Big)$ tests when applied to a set containing up to d errors of strength at most 3. In addition, our algorithm, unlike nonadaptive ones, does not require foreknowledge of an upper bound on d. We give a comprehensive comparison of the two strength-$\overline{3}$ CGT methods in Section 5.4.

In Section 5.3, we generalize the algorithm for error location with safe values to strength $\overline{3}$. This algorithm adaptively generates a strength-$\overline{3}$ ELA for a $TP(k, (g_1, g_2, ..., g_k))$ that has $g = \max\{g_1, g_2, ..., g_k\}$ fixed, and safe values $s_1, s_2, ..., s_k$. The ELA consists of $\mathcal{O}\big(d^3(\ln k)^2 + \ln k\big)$ tests, where d is the number of faulty interactions among the items in the $TP(k, (g_1, g_2, ..., g_k))$.

We remind the reader that in the context of group testing, errors are defined as failing sets that are minimal with respect to set inclusion (see Definition 4.0.10).

5.1 Combinatorial Group Testing, Strength $\overline{3}$

The new algorithm LocateErrorsInSet (given as Algorithm 5.1) has a structure similar to the algorithm from Martínez et al. [32] called LocateErrorsInTest. Given a test T and a set (of factors) $A \subseteq C_T$ (see Definition 4.1.1), our algorithm generates a set of tests which identifies all errors among items associated with factors in A, assuming that all errors present are of strength at most 3. For brevity, we sometimes refer to 3-subsets of A as **triples**.

Algorithm 5.1 works as follows. We store all 3-subsets of A in the set U, and whenever we identify a triple τ as corresponding to either a passing or failing set of items, we remove τ from U. At each step, we construct a set (of factors) $A' \subseteq A$ which contains many triples from U. Whenever the cardinality of A' is not a power of 3, we take the union of A' with a set X of "dummy factors", so that $|A'| = 3^l$ for some integer $l \geq 2$ for all subsequent tests within this iteration. This process greatly simplifies our later analysis.

If A' corresponds to a failing set of items, then the procedure LocateTriple(A') identifies a triple $\tau \subseteq A'$ such that at least one error is within τ. Next, we identify the set E' containing all factor-sets corresponding to errors within τ, via the procedure LocateErrorsInTriple(τ). We then remove from U each triple that corresponds to an error $e \in E'$.

If A' corresponds to a passing set of items, then we remove all 3-subsets of A' from U. More generally, throughout the algorithm, whenever TestSet(D, T) passes for some set D of cardinality at least three, we remove each 3-subset of D from U.

The algorithm terminates when all 3-subsets of items corresponding to triples in A have been tested.

We now proceed to prove the correctness of Algorithm 5.1. We begin with the subprocedure MaxUnidentifiedTriples(A, U, E), which does the following:

1. It constructs a list M of sets such that every triple in U is a subset of at least one set in M.

2. No set in M contains a factor-set $e \in E$ (those factor-sets correspond to previously-identified errors).

3. The set R returned by the procedure contains at least as many triples $\tau \in U$ as are contained in any other set in M. In particular, R contains at least $\frac{1}{\alpha}|U|$ triples that are also in U, for some α. In order to determine the value of the parameter α, we first need some results involving antichains, which we define below.

Definition 5.1.1 *An **antichain** is a collection $\mathcal{A}$ of subsets of an n-set such that for all $A, B \in \mathcal{A}$ with $A \neq B$, we have $A \not\subseteq B$.*

We remark here on the relation between Definitions 5.1.1 and 2.1.1. An independent set of interactions is clearly an antichain, and an antichain of subsets that are interactions is an independent set of interactions.

The following lemma has been shown independently by Lubell [29], Mešalkin [34], and Yamamoto [42], establishing the so-called LYM inequality.

Algorithm 5.1 Precondition: T is a test that may cover errors of strength at most 3, and $A \subseteq C_T$ is a set of $k \geq 3$ factors. Let $\textsc{TestSet}(A) = \textsc{TestSet}(A, T)$. This algorithm returns the set
$E = \big\{(e \subseteq A : \textsc{TestSet}(e) = \text{fail}) \text{ and } (\nexists e' \subset e : \textsc{TestSet}(e') = \text{fail})\big\}$.

```
procedure LocateErrorsInSet($A, T$)
    $E \leftarrow \emptyset$, and $U \leftarrow \{\tau \subseteq A : |\tau| = 3\}$.
    ▷ While there are triples corresponding to sets (of items) whose pass/fail
    ▷ results are still unknown, keep testing.
    while $U \neq \emptyset$ do
        $A' \leftarrow$ MaxUnidentifiedTriples($A, U, E$)
        if $|A'| \neq 3^l$ for some integer $l \geq 1$ then
            Let $X$ be a set of "dummy factors" that do not affect test outcomes,
            such that $A' \cap X = \emptyset$, $|X| < 2|A'|$, and $|A' \cup X| = 3^l$ for some $l \geq 2$.
            $A' \leftarrow A' \cup X$
        if TestSet($A'$) = fail then
            $\tau \leftarrow$ LocateTriple($A'$)
            $E' \leftarrow$ LocateErrorsInTriple($\tau$)
            RemoveErrors($U, E'$)
            $E \leftarrow E \cup E'$
        else RemoveNonErrors($U, A'$)
    return $E$

▷ Precondition: $E$ is a set of factor-sets corresponding to errors. This subprocedure
▷ removes from $U$ each triple $\tau$ that has $e \subseteq \tau$ for some $e \in E$.
procedure RemoveErrors($U, E$)
    for $e \in E$ do
        for $\tau \in U$ do
            if $e \subseteq \tau$ then $U \leftarrow U \setminus \{\tau\}$

▷ Precondition: $P$ is a factor-set corresponding to a passing set of items, and
▷ $|P| \geq 3$. This subprocedure removes each triple contained in $P$ from $U$.
procedure RemoveNonErrors($U, P$)
    for every 3-subset $\tau \subseteq P$ do $U \leftarrow U \setminus \{\tau\}$

▷ Precondition: $\tau$ is a triple corresponding to a failing set of items. This
▷ subprocedure returns the set $E$ of all subsets of $\tau$ that correspond to errors.
procedure LocateErrorsInTriple($\tau$)
    $E \leftarrow \emptyset$
    for $a \in \tau$ do
        if TestSet($\{a\}$) = fail then $E \leftarrow E \cup \big\{\{a\}\big\}$
    for each $\tau' \subseteq \tau$ such that $|\tau'| = 2$ do
        if $e' \not\subseteq \tau'$ for all $e' \in E$ and TestSet($\tau'$) = fail then $E \leftarrow E \cup \{\tau'\}$
    if $E = \emptyset$ then $E \leftarrow \{\tau\}$
    return $E$
```

Lemma 5.1.2 *[29, 34, 42] Let n be a positive integer, and let $\mathcal{A}$ be an antichain of subsets of an n-set. Then,*

$$\sum_{A \in \mathcal{A}} \frac{1}{\binom{n}{|A|}} \leq 1.$$

We now use the preceding lemma to prove a generalization of Sperner's Theorem [39] for the case of antichains with subsets of bounded cardinality.

Theorem 5.1.3 *Let $l \leq n$ be a positive integer, and let $\mathcal{A}$ be an antichain of subsets of an n-set such that $|A| \leq l$ for all $A \in \mathcal{A}$. Then,*

$$|\mathcal{A}| \leq \begin{cases} \binom{n}{l} & \textit{if } l \leq \lceil n/2 \rceil \\ \binom{n}{\lfloor n/2 \rfloor} & \textit{if } l \geq \lfloor n/2 \rfloor. \end{cases}$$

Proof: Let

$$\mathcal{C} = \begin{cases} \binom{n}{l} & \text{if } l \leq \lceil n/2 \rceil \\ \binom{n}{\lfloor n/2 \rfloor} & \text{otherwise.} \end{cases}$$

Take any $A \in \mathcal{A}$. Clearly $\binom{n}{|A|} \leq \mathcal{C}$. A slight rearrangement of that inequality yields $\frac{1}{\mathcal{C}} \leq \frac{1}{\binom{n}{|A|}}$. It is then easy to see that

$$|\mathcal{A}| \frac{1}{\mathcal{C}} \leq \sum_{A \in \mathcal{A}} \frac{1}{\binom{n}{|A|}} \leq 1,$$

where the final inequality comes from Lemma 5.1.2. Thus, $|\mathcal{A}| \leq \mathcal{C}$. For the values $l = \lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, it is easy to see that $\binom{n}{l} = \binom{n}{\lfloor n/2 \rfloor} = \binom{n}{\lceil n/2 \rceil}$, i.e. both formulas for $\mathcal{C}$ coincide, and the proof is complete. ∎

We now prove the correctness of the procedure MaxUnidentifiedTriples.

Lemma 5.1.4 *Let $A = [1,k]$ be a set of $k \geq 3$ factors. Let U be a set of 3-subsets of A, and let E be a collection of subsets of A, $|E| \leq d$, such that for all $e \in E$,*

(1) e corresponds to a failing set of items, $|e| \leq 3$, and

(2) for all $\tau \in U$, we have $e \nsubseteq \tau$.

Then the procedure MaxUnidentifiedTriples(A, U, E) *(given as Algorithm 5.2 on Page 110) correctly returns a factor-set $R \subseteq A$ such that*

$$\big|\{\tau \subseteq R : |\tau| = 3\} \cap U\big| \geq \frac{1}{\alpha}|U|, \text{ where } \alpha = \begin{cases} \binom{3d}{3} & \text{if } d \geq 2 \\ 3 & \text{otherwise.} \end{cases}$$

Proof: First, suppose that $E = \emptyset$. Then $R = A$, and clearly

$$\big|\{\tau \subseteq R : |\tau| = 3\} \cap U\big| = |U| \geq \frac{1}{\alpha}|U|$$

since $\alpha \geq 1$.

Next, consider the case where $E \neq \emptyset$. Let $m = |M|$, where M is the list constructed by MaxUnidentifiedTriples(A, U, E). Let $\mathcal{T}_i$ be the set of triples in U covered by $M[i]$, i.e. let

$$\mathcal{T}_i = \big\{\tau \subseteq M[i] : |\tau| = 3\big\} \cap U, \text{ where } i \in [1, m].$$

We claim that for each triple $\tau \in U$, we have $\tau \in \mathcal{T}_i$ for some $i \in [1, m]$.

Take any $\tau \in U$, and let $S = \bigcup_{e \in E} e$, as calculated by the algorithm; also, let $\tau' = \tau \cap S$. We can write $\tau = \tau' \cup \big(\tau \cap (A \setminus S)\big)$. Observe that $|\tau'| \leq 3$.

If $|\tau'| = 0$, then $\tau \subseteq A \setminus S$, in which case $\tau \subseteq M[i]$ for all $i \in [1, m]$. Assume now that $1 \leq |\tau'| \leq 3$. By the assumption of this lemma, there is no $e \in E$ with $e \subseteq \tau'$, so the second for-loop will add the elements of a set $S' \supseteq \tau'$ to $M[i]$ for some $i \in [1, m]$.

Algorithm 5.2 Under the assumptions of Lemma 5.1.4, this procedure returns a set of factors $R \subseteq A$ such that $\left|\{\tau \subseteq R : |\tau| = 3\} \cap U\right| \geq \frac{1}{\alpha}|U|$ where $\alpha = \binom{3d}{3}$ if $d \geq 2$, and $\alpha = 3$ otherwise.

```
procedure MaxUnidentifiedTriples($A, U, E$)
    ▷ Special Case
    if $E = \emptyset$ then $R \leftarrow A$
    else
        ▷ Store in $S$ all factors involved in factor-sets $e \in E$. Then store in $M$ all
        ▷ subsets $S' \subset S$ such that $|S'| \in [1, 3]$, $S'$ does not contain any $e \in E$, and
        ▷ $S'$ is maximal (w.r.t. set inclusion) among the sets with these properties.
        $S \leftarrow \emptyset$

        ▷ $M$ will be a list of $m$ sets whose $i$th set is $M[i]$, where $i \geq 1$ is an integer.
        $m \leftarrow 0$, and let $M$ be the empty list.
        for $e \in E$ do $S \leftarrow S \cup e$
        for $l \leftarrow 3$ down to 1 do
            for $S' \subseteq S$ such that $|S'| = l$ do
                if $\left(\nexists i \in [1, m] \text{ such that } S' \subseteq M[i]\right)$ and $\left(\forall e \in E, e \nsubseteq S'\right)$ then
                    $m \leftarrow m + 1$
                    $M[m] \leftarrow S'$

        ▷ Take the union of each set in $M$ with the set of all factors not contained
        ▷ in any factor-set $e \in E$.
        for $i \leftarrow 1$ to $m$ do
            $M[i] \leftarrow M[i] \cup (A \setminus S)$

        ▷ Find a set $M[i]$ that has as many triples in common with $U$ as possible.
        $max \leftarrow 0$
        $i_{max} \leftarrow 1$
        for $i \leftarrow 1$ to $m$ do
            $j \leftarrow \left|\left\{\tau \subseteq M[i] : |\tau| = 3\right\} \cap U\right|$
            if $j > max$ then
                $max \leftarrow j$
                $i_{max} \leftarrow i$
        $R \leftarrow M[i_{max}]$
    return $R$
```

The third (main) for-loop will add the elements of $A \setminus S$ to $M[i]$, so

$$\tau = \tau' \cup \big(\tau \cap (A \setminus S)\big) \subseteq S' \cup \big(A \setminus S\big) = M[i].$$

Thus $\tau \in \mathcal{T}_i$ for some $i \in [1, m]$, i.e. $\tau \in \bigcup_{i=1}^{m} \mathcal{T}_i$. It then easily follows that $\bigcup_{i=1}^{m} \mathcal{T}_i = U$.

Now, let i_{max} be as computed by the algorithm. Then $\left|\mathcal{T}_{i_{max}}\right| \geq |\mathcal{T}_i|$ for all $i \in [1, m]$. We conclude that

$$|U| = \left|\bigcup_{i=1}^{m} \mathcal{T}_i\right| \leq \sum_{i=1}^{m} \left|\mathcal{T}_i\right| \leq m\left|\mathcal{T}_{i_{max}}\right|.$$

Therefore, $\left|\mathcal{T}_{i_{max}}\right| \geq \frac{1}{m}\left|U\right|$.

We conclude the proof by showing that $m \leq \alpha$.

Let S_i' be the set S' whose elements are added to $M[i]$ in the body of the second for-loop, $i \in [1, m]$. Consider $\mathcal{S} = \{S_1', S_2', ..., S_m'\}$. By construction, $\mathcal{S}$ is an antichain, since sets are added to M in non-increasing order of cardinality, and whenever S' is added to M, no proper subset of S' is subsequently added to M.

Since E is a collection of factor-sets corresponding to errors, each of which has strength at most 3, we have $|E| \leq d$ and hence $s = |S| \leq 3|E| \leq 3d$. Therefore $\mathcal{S}$ is an antichain of subsets of an s-set with $s \leq 3d$, such that each $S_i' \in \mathcal{S}$ satisfies $|S_i'| \leq 3$.

By Theorem 5.1.3, we obtain

$$m \leq \begin{cases} \binom{s}{3} & \text{if } 3 \leq \lceil s/2 \rceil \\ \binom{s}{\lfloor s/2 \rfloor} & \text{if } 3 \geq \lfloor s/2 \rfloor. \end{cases}$$

If $s \leq 3$, then $3 \geq \lfloor s/2 \rfloor$ and hence $m \leq \binom{s}{\lfloor s/2 \rfloor} \leq 3$. Suppose that $d = 1$. Then $s \leq 3$ and by the above inequality, $m \leq 3$.

Hence assume that $d \geq 2$ and $s \geq 4$. If $s \geq 5$, then $m \leq \binom{s}{3} \leq \binom{3d}{3}$ as required.

The remaining case is $d \geq 2, s = 4$. Then $m \leq \binom{4}{2} = 6 < 20 = \binom{6}{3} \leq \binom{3d}{3}$.

Hence

$$m \leq \begin{cases} \binom{3d}{3} & \text{if } d \geq 2 \\ 3 & \text{if } d \leq 1. \end{cases}$$

That is, $m \leq \alpha$. Therefore $\left|\{T \subseteq R : |T| = 3\} \cap U\right| = \left|\mathcal{T}_{i_{max}}\right| \geq \frac{1}{m}|U| \geq \frac{1}{\alpha}|U|$. ∎

Next, we prove the correctness of the procedure LocateErrorsInTriple.

Lemma 5.1.5 *Let τ be a triple corresponding to a failing set S containing three items. Then* LocateErrorsInTriple(τ) *correctly returns the set of all subsets of τ that correspond to errors in S.*

Proof: By Definition 4.0.10, no error can contain an error of lesser strength as a subset. Any failing singleton subset of S is clearly a strength-1 error; every singleton subset of S is tested, via its corresponding factor-set, by LocateErrorsInTriple.

Let E be the collection of factor-sets corresponding to errors found after the first for-loop, and note that E contains every subset of τ that corresponds to a strength-1 in S. The second for-loop tests only those 2-subsets of τ that contain no factor-set already stored in E; any failing set of items corresponding to a 2-subset $\tau' \subset \tau$ is a strength-2 error, in which case the algorithm stores τ' in E.

The set τ itself corresponds to a strength-3 error if and only if no error of lesser strength corresponds to a proper subset $\tau' \subset \tau$, so E is empty after the two for-loops in this procedure if and only if τ corresponds to a strength-3 error, in which case we let $E = \{\tau\}$.

Therefore LocateErrorsInTriple(τ) correctly returns the set E of all subsets of τ corresponding to errors. ∎

We now proceed to show that LocateTriple(D) finds a 3-subset of D corresponding to a failing set of items.

Lemma 5.1.6 *Assume that D is factor-set corresponding to a failing set of items containing errors of strength at most* 3, *and* $|D| = 3^l$ *for some integer* $l \geq 1$. *Then* LocateTriple(D) *(given in Algorithm 5.3) correctly returns a* 3*-subset of D that also corresponds to a failing set of items.*

Proof: We proceed by induction on l. First, consider the base case where $l = 1$. Then $|D| = 3$, so LocateTriple(D) returns D, and the thesis holds.

Assume that for some integer $l \geq 1$, LocateTriple(B) correctly returns a failing 3-subset of B, for any failing set B such that $|B| = 3^l$.

Suppose that $|D| = 3^{l+1}$, and LocateTriple(D) is called. Since $|D| \geq 9$, the "else" block is executed. The procedure first partitions D into thirds $[D', D'', D''']$. Any error within D must be within one of D', D'', or D''', or across the partition $[D', D'', D''']$.

If an error is within one of D', D'', or D''', then we call LocateTriple(D^*), where $D^* \in \{D', D'', D'''\}$ is a factor-set of cardinality 3^l that corresponds to a failing set of items. By the induction hypothesis, LocateTriple(D^*) returns a triple $\tau \subseteq D^*$ corresponding to a failing set of items. Since $D^* \subset D$, the thesis holds.

Otherwise there is an error across the partition $[D', D'', D''']$, in which case we call the procedure AcrossLocate, which depends upon three auxiliary subprocedures: HalveFirstSet, HalveSecondSet, and OneBallPerBin in the following way. If one of $D' \cup D''$, $D' \cup D'''$, or $D'' \cup D'''$ corresponds to a failing set of items, then we have a set $D^{**} \in \{D' \cup D'', D' \cup D''', D'' \cup D'''\}$ such that $|D^{**}| = \frac{2}{3}|D| = 2 \cdot 3^l$. We refer to this as **Case 1**.

When $D' \cup D''$, $D' \cup D'''$, and $D'' \cup D'''$ all correspond to passing sets, there must be at least one strength-3 error within D that has one factor in each of D', D'', D''', in which case we call OneBallPerBin(D', D'', D'''). We refer to this as **Case 2**.

Algorithm 5.3 Precondition for LocateTriple(D): the factor-set D corresponds to a failing set of items containing errors of strengths at most 3, such that $|D| = 3^l$ for some integer $l \geq 1$. This algorithm returns a 3-subset of D that also corresponds to a failing set of items.

Precondition for AcrossLocate(D', D'', D'''): the partition $[D', D'', D'']$ of D is such that $|D'| = |D''| = |D'''| = 3^l$ for some integer $l \geq 1$, and no error within D is also within one of D', D'', or D''. This subprocedure, in combination with HalveFirstSet and HalveSecondSet, determines that an error within D is either across the partition $[B, C]$ of a set $B \cup C$ (where each of B, C is one half of D', D'', or D''), or has one factor in each of D', D'', and D'''.

```
procedure LocateTriple($D$)
    if $|D| = 3$ then return $D$
    else
        Partition $D$ into thirds $[D', D'', D'']$.
        if TestSet($D'$) = fail then return LocateTriple($D'$)
        else
            RemoveNonErrors($U, D'$)
            if TestSet($D''$) = fail then return LocateTriple($D''$)
            else
                RemoveNonErrors($U, D''$)
                if TestSet($D'''$) = fail then return LocateTriple($D'''$)
                else
                    RemoveNonErrors($U, D'''$)
                    return AcrossLocate($D', D'', D'''$)

procedure AcrossLocate($D', D'', D'''$)
    if TestSet($D' \cup D''$) = fail then return HalveFirstSet($D', D''$)
    else
        RemoveNonErrors($U, D' \cup D''$)
        if TestSet($D' \cup D'''$) = fail then return HalveFirstSet($D', D'''$)
        else
            RemoveNonErrors($U, D' \cup D'''$)
            if TestSet($D'' \cup D'''$) = fail then return HalveFirstSet($D'', D'''$)
            else
                RemoveNonErrors($U, D'' \cup D'''$)
                return OneBallPerBin($D', D'', D'''$)
```

Case 1: Without loss of generality, suppose that $D^{**} = D' \cup D''$. Let $B = D', C = D''$, and let c be an arbitrary element of C. The procedure HalveFirstSet takes the union of B with $\{c\}$ and removes c from C so that each of B, C contains an even number of elements, $|B| = 3^l + 1$, and $|C| = 3^l - 1$. It then partitions each of B, C into halves as $[B', B''], [C', C'']$, respectively.

The procedures HalveFirstSet and HalveSecondSet collectively determine whether one of $B' \cup C'$, $B' \cup C''$, $B'' \cup C'$, or $B'' \cup C''$ corresponds to a failing set of items. First suppose that one of those sets does correspond to a failing set, and let it be $B' \cup C'$, without loss of generality. The sets B' and C' are disjoint, so

$$|B' \cup C'| = |B'| + |C'| = \frac{1}{2}|B| + \frac{1}{2}|C| = \frac{1}{2}(3^l + 1 + 3^l - 1) = 3^l.$$

Therefore, by the induction hypothesis, LocateTriple$(B' \cup C')$ returns a triple $\tau \subseteq B' \cup C' \subset D^{**}$ that corresponds to a failing set of items. Since $D^{**} \subset D$, the thesis holds.

Alternatively, suppose that $B' \cup C'$, $B' \cup C''$, $B'' \cup C'$, and $B'' \cup C''$ all correspond to passing sets. Then any error within $B \cup C$ must have one factor in each of three known disjoint subsets $X, Y, Z \in \{B', B'', C', C''\}$, i.e. we are in Case 2, and we call OneBallPerBin(X, Y, Z).

Case 2: The procedure OneBallPerBin(X, Y, Z) identifies the factors in a triple (corresponding to a strength-3 error) factor-by-factor via a binary search subprocedure called BinarySearch, which is conducted on each of X, Y, Z. A binary search on X iteratively halves X while keeping $X \cup Y \cup Z$ in correspondence with a failing item set. This reduces the cardinality of X to 1, while still having a strength-3 error across $[X, Y, Z]$ such that each of X, Y, Z contains one factor involved in that error. Subsequent binary searches reduce the cardinalities of Y and Z in a similar way, finally returning a factor-set $\{x, y, z\}$ that corresponds to an error, where $x \in X, y \in Y$, and $z \in Z$. ∎

We remark here on two subprocedures which occur within LocateErrorsInSet. First, RemoveNonErrors(U, P) clearly removes from U precisely all 3-subsets of the factor-set P that corresponds to a passing set of items. Second, the procedure RemoveErrors(U, E) clearly removes from U precisely each triple containing a factor-set $e \in E$ that corresponds to an error. These remarks can be easily verified by a careful reading of the respective subprocedures.

Next, we show that LocateErrorsInSet(A, T) identifies each subset of A that corresponds to an error exactly once.

Theorem 5.1.7 *Let T be a test, and let A be a set of k factors such that $A \subseteq C_T$, and assume that every error within A has strength at most* 3. *Then the following two statements hold.*

(a) At each iteration of its while-loop, the procedure LocateErrorsInSet(A, T) *removes either at least one triple from U that corresponds to a failing item-set, or it removes at least $\frac{1}{\alpha}|U|$ triples from U, each corresponding to a passing item-set, where α is given in Lemma 5.1.4.*

(b) LocateErrorsInSet(A, T) *returns the set E of all subsets of A that correspond to faulty interactions.*

Proof: LocateErrorsInSet begins with U containing all 3-subsets of A, and $E = \emptyset$. At each step, triples are removed from U as their corresponding item-sets are identified as either passing or failing, and the factor-sets corresponding to errors are stored in E.

We first prove (a). We remark here that the while-loop iterates as long as there are triples corresponding to item-sets whose pass/fail status is unknown (i.e. $U \neq \emptyset$). MaxUnidentifiedTriples(A, U, E) returns a set A' containing at least one triple that is also in U, by Lemma 5.1.4, since $|U| > 0$. If TestSet(A') returns fail, then by Lemma 5.1.6, LocateTriple(A') returns a triple τ that corresponds to a failing set, and by Lemma 5.1.5, LocateErrorsInTriple(τ) returns the set E' of all subsets

Algorithm 5.4 Precondition: $|B| = |C| = 3^l$ for some integer $l \geq 1$, and there is an error across the partition $[B, C]$ of the set $B \cup C$. In combination with HalveSecondSet, this procedure determines that an error is within a subset $S \subseteq B \cup C$ such that $|S| = \frac{1}{2}|B \cup C| = 3^l$ for some integer $l \geq 1$, or that a strength-3 error has one factor in each of three disjoint sets X, Y, and Z, such that each of X, Y, and Z is a subset of either B or C.

```
procedure HalveFirstSet($B, C$)
    ▷ Move one element from $C$ to $B$ so that we can halve each of $B$ and $C$.
    Choose one element $c \in C$. $B \leftarrow B \cup \{c\}$, and $C \leftarrow C \setminus \{c\}$.
    Partition each of $B, C$ into halves $[B', B'']$, $[C', C'']$, respectively.
    if TestSet($B' \cup C$) = fail then return HalveSecondSet($B', C', C''$)
    else
        RemoveNonErrors($B' \cup C$)
        if TestSet($B'' \cup C$) = fail then return HalveSecondSet($B'', C', C''$)
        else
            RemoveNonErrors($B'' \cup C$), return OneBallPerBin($B', B'', C$)

▷ Precondition: $B^* \in \{B', B''\}$, and there is an error across the partition $[B^*, C]$
▷ of the set $B^* \cup C$. This subprocedure is a continuation of HalveFirstSet.
procedure HalveSecondSet($B^*, C', C''$)
    if TestSet($B^* \cup C'$) = fail then return LocateTriple($B^* \cup C'$)
    else
        RemoveNonErrors($B^* \cup C'$)
        if TestSet($B^* \cup C''$) = fail then return LocateTriple($B^* \cup C''$)
        else
            RemoveNonErrors($B^* \cup C''$), return OneBallPerBin($B^*, C', C''$)

▷ Precondition: $X \cup Y \cup Z$ corresponds to a failing set of items, and there is a
▷ strength-3 error with one factor in each of the disjoint sets $X, Y, Z$. This
▷ subprocedure identifies each factor and returns the factor-set corresponding to
▷ the strength-3 error.
procedure OneBallPerBin($X, Y, Z$)
    $\{x\} \leftarrow$ BinarySearch($X, Y \cup Z$), and $\{y\} \leftarrow$ BinarySearch($Y, \{x\} \cup Z$)
    $\{z\} \leftarrow$ BinarySearch($Z, \{x, y\}$), and return $\{x, y, z\}$

▷ Precondition: $W \cup R$ corresponds to a failing item-set, and $w \in W$ is one factor
▷ involved in a strength-3 error, where the other two factors are in $R$.
▷ This subprocedure conducts a binary search in $W$, and returns $\{w\}$.
procedure BinarySearch($W, R$)
    if $|W| = 1$ then return $W$
    else
        Partition $W$ into $[W', W'']$ as evenly as possible.
        if TestSet($W' \cup R$) = fail then return BinarySearch($W', R$)
        else
            RemoveNonErrors($W' \cup R$), return BinarySearch($W'', R$)
```

of τ corresponding to faulty interactions. Next, $\textsc{RemoveErrors}(U, E')$ removes from U each triple that contains a factor-set $e \in E'$ that corresponds to an error. In particular, τ is removed from U by $\textsc{RemoveErrors}(U, E')$. Therefore, at least one triple is removed from U in each iteration of the while-loop in which $\textsc{TestSet}(A')$ returns fail.

If $\textsc{TestSet}(A')$ returns pass, then $\textsc{RemoveNonErrors}(U, A')$ removes every 3-subset of A' from U, since they all correspond to passing sets of items. By Lemma 5.1.4, there are at least $\frac{1}{\alpha}|U|$ such 3-subsets of A'.

Therefore (a) holds.

It remains to show that every subset of A that corresponds to a failing interaction is contained in the set E returned by the algorithm. Before we do so, we prove that the following statement is a loop invariant for the while-loop in $\textsc{LocateErrorsInSet}$:

$\mathcal{P}(E, U) =$"For each $e \subseteq A$ that corresponds to a faulty interaction, either $e \in E$, or $e \subseteq \tau$ for some triple $\tau \in U$."

First, notice that $\mathcal{P}(E, U)$ is true before the first iteration of the while-loop. Every $e \subseteq A$ that corresponds to a faulty interaction has cardinality at most 3, therefore $e \subseteq \tau$ for some $\tau \in U = \left\{\tau' \subseteq A : |\tau'| = 3\right\}$, and obviously $e \notin E = \emptyset$.

Consider an arbitrary iteration of the while-loop and let E_B and U_B be the values of E and U, respectively, before this iteration. Similarly, let E_A and U_A be the values of E and U after this iteration. Assume that $\mathcal{P}(E_B, U_B)$ holds; we will show that $\mathcal{P}(E_A, U_A)$ also holds.

Now, let $e \subseteq A$ be a factor-set corresponding to a faulty interaction. Then either $e \in E_B$ or $e \subseteq \tau$ for some $\tau \in U_B$, since $\mathcal{P}(E_B, U_B)$ holds. We consider two cases as follows.

<u>Case 1:</u> $e \in E_B$. In this case, $e \in E_A$ since $E_B \subseteq E_A$, therefore $\mathcal{P}(E_A, U_A)$ holds.

<u>Case 2:</u> $e \notin E_B$, and $e \subseteq \tau$ for some $\tau \in U_B$.

Let A' be the set returned by $\textsc{MaxUnidentifiedTriples}(A, U_B, E_B)$.

If $\textsc{TestSet}(A')$ returns fail, then the set $\tau \subseteq A'$ returned by $\textsc{LocateTriple}(A')$

corresponds to a failing set of items. If $e \subseteq \tau$, then $e \in E'$, since every error within τ has its corresponding factor-set stored in E', by Lemma 5.1.5. In this case, $e \in E_B \cup E' = E_A$. Moreover, RemoveErrors(U, E') guarantees that there exists no $\tau' \in U_A$ such that $e \subseteq \tau'$. Therefore $\mathcal{P}(E_A, U_A)$ holds in Case 2 when A' corresponds to a failing set of items.

On the other hand, if TestSet(A') returns pass, then $e \not\subseteq A'$, and $U_A = U_B \setminus \{\tau'' \subseteq A' : |\tau''| = 3\}$ after we call RemoveNonErrors(U_B, A'). We see that $\tau \in U_A$, since $e \not\subseteq A'$ and $e \subseteq \tau \in U_B$. Furthermore, $E_A = E_B$, so $e \notin E_A$. Therefore $\mathcal{P}(E_A, U_A)$ also holds for Case 2 when A' corresponds to a passing set of items.

Since $\mathcal{P}(E, U)$ holds before and after an iteration of the while-loop, we conclude that $\mathcal{P}(E, U)$ is a loop invariant.

Now, we show that the while-loop terminates after a finite number of iterations. During each iteration, $|U|$ strictly decreases, since by (a) at least one triple is removed from U by one of RemoveErrors(U, E') and RemoveNonErrors(U, A'). Since the stopping condition of the loop is $|U| = 0$, we conclude that the loop terminates. At the end of the last iteration of the while-loop, the loop invariant $\mathcal{P}(E, \emptyset)$ holds, which implies (b). ∎

This concludes the proof of correctness of Algorithm 5.1; in the next section we analyze its performance.

5.2 Performance Analysis

We measure each algorithm's cost as we did in previous chapters: by the number of times it calls Test (note that every call to TestSet is also a call to Test). We begin by counting the maximum number of tests required to identify all subsets of a 3-set of factors that correspond to errors.

Lemma 5.2.1 *Under the assumptions of Lemma 5.1.5, at most* 6 *tests are conducted by* LocateErrorsInTriple(T).

Proof: The procedure LocateErrorsInTriple(τ) conducts one test for each element of τ, and at most one test per pair, for a total of at most $\binom{3}{1} + \binom{3}{2} = 6$ tests. ∎

Next, we give the maximum number of tests conducted by LocateTriple. However, we first need the following auxiliary lemma.

Lemma 5.2.2 *Let T_l be a sequence of integers satisfying*

$$T_l = \begin{cases} 0 & \textit{if } l = 1 \\ 10 + \max\left\{T_{l-1}, 3\lceil (l-1)\log_2 3\rceil\right\} & \textit{if } l > 1. \end{cases}$$

Then $T_l < 10l$ for all $l \geq 1$.

Proof: We first prove, by induction on l, that the following claim holds for all $l \geq 3$.

$\mathcal{C}(l)$: $T_{l-1} > 3\lceil (l-1)\log_2 3\rceil$.

To see that $\mathcal{C}(3)$ holds, we first calculate

$$T_2 = 10 + \max\left\{T_1, 3\lceil \log_2 3\rceil\right\} = 10 + \max\left\{0, 6\right\} = 16 > 12 = 3\lceil 2\log_2 3\rceil.$$

Hence $\mathcal{C}(l)$ holds for $l = 3$.

Now, assume that $\mathcal{C}(l)$ holds for some integer $l \geq 3$, and consider $\mathcal{C}(l+1)$.

By the induction hypothesis, we have

$$T_l = 10 + T_{l-1} > 10 + 3\left\lceil (l-1)\log_2 3\right\rceil.$$

Next, we see that

$$\lceil (l-1)\log_2 3 \rceil = \lceil l \log_2 3 - \log_2 3 \rceil \geq \lceil l \log_2 3 - 2 \rceil = \lceil l \log_2 3 \rceil - 2.$$

Hence

$$T_l > 10 + 3\big(\lceil l \log_2 3 \rceil - 2\big) = 4 + 3\lceil l \log_2 3 \rceil > 3\lceil l \log_2 3 \rceil,$$

i.e. $\mathcal{C}(l+1)$ holds.

We conclude that $\mathcal{C}(l)$ holds for all $l \geq 3$. Consequently, $T_l = 10 + T_{l-1}$ for all $l > 2$. We now prove, also by induction on l, that $T_l \leq 10l$ for all $l \geq 1$.

We have already seen that $T(1) = 0 < 10$ and $T(2) = 16 < 20$, therefore the inequality holds for $l = 1, 2$.

Now, assume that $T_l < 10l$ for some $l \geq 2$. Since $\mathcal{C}(l+1)$ holds, we have $T_{l+1} = 10 + T_l$, and applying the induction hypothesis, we see that

$$T_{l+1} < 10 + 10l = 10(l+1).$$

The result follows by induction. ∎

Now, we give the maximum number of tests conducted by LocateTriple (see Algorithm 5.3 on Page 114).

Lemma 5.2.3 *Under the assumptions of Lemma 5.1.6,* LocateTriple(D) *conducts fewer than* $10\log_3 n$ *tests, where* $n = |D|$.

Proof: Let $T(l)$ be the maximum number of tests conducted by LocateTriple(D) when $n = |D| = 3^l$. We proceed by induction on $l = \log_3 n$.

It is easy to see that LOCATETRIPLE(D) returns D after conducting no tests at all when $l = 1$, therefore $T(1) = 0$.

When $l > 1$, we see that LOCATETRIPLE(D) conducts up to 10 tests before calling either LOCATETRIPLE(D^*) for some $D^* \subset D$ such that $|D^*| = \frac{1}{3}|D|$, or ONEBALLPERBIN(X, Y, Z) for some $X, Y, Z \subset D$ such that $|X|, |Y|, |Z| \leq n/3$ (these 10 tests consist of 3 tests in LOCATETRIPLE, 3 tests in ACROSSLOCATE, 2 tests in HALVEFIRSTSET, and 2 tests in HALVESECONDSET). It is easy to see (e.g. by induction) that the number of tests conducted by each call to BINARYSEARCH within ONEBALLPERBIN(X, Y, Z) is at most

$$\lceil \log_2(n/3) \rceil = \lceil (l-1)\log_2 3 \rceil.$$

So, in total ONEBALLPERBIN(X, Y, Z) conducts at most $3\lceil (l-1)\log_2 3 \rceil$ tests.

Then $T(l)$ satisfies the recurrence relation given in Lemma 5.2.2, with $T_l = T(l)$. Therefore $T(l) < 10\log_3 n$. ∎

The main while-loop of LOCATEERRORSINSET reduces $|U|$, the number of triples corresponding to 3-sets of items whose pass/fail status is not yet known, by a constant ratio $\frac{1}{\alpha}$ ($\alpha > 1$ is an integer) at each iteration in which the factor-set A' corresponds to a passing set of items (see the proof of Theorem 5.1.7(a)). It has been shown [10] that, in such cases, the number of iterations is bounded above by $\alpha \ln u_0$, where u_0 is the initial value of $|U|$. We give a proof of this bound in the following lemma.

Lemma 5.2.4 *Let $(u_i)_{i=0}^{\infty}$ be a sequence of non-negative integers, $u_0 > 1$, and let $\alpha > 1$ be an integer. Suppose that $u_i \leq u_{i-1}\left(1 - \frac{1}{\alpha}\right)$ for all $i \geq 1$.*

Let N be the smallest value of i such that $u_i = 0$. Then $N \leq \alpha \ln u_0$.

Proof: From the hypothesis we get $u_i \leq u_0\left(1-\frac{1}{\alpha}\right)^i$. We claim that

$$N \leq -\frac{\ln u_0}{\ln\left(1-\frac{1}{\alpha}\right)} + 1.$$

Indeed, for any $i > -\frac{\ln u_0}{\ln\left(1-\frac{1}{\alpha}\right)}$, we have

$$u_i < u_0\left(1-\frac{1}{\alpha}\right)^{-\frac{\ln u_0}{\ln\left(1-\frac{1}{\alpha}\right)}}.$$

Taking logarithms of both sides, and applying well-known properties of logarithmic functions, we get

$$\ln u_i < \ln\left(u_0\left(1-\frac{1}{\alpha}\right)^{-\frac{\ln u_0}{\ln\left(1-\frac{1}{\alpha}\right)}}\right) = \ln u_0 - \frac{\ln u_0}{\ln\left(1-\frac{1}{\alpha}\right)}\ln\left(1-\frac{1}{\alpha}\right) = 0.$$

Therefore, $u_i < 1$. Since u_i is an integer, we have $u_i = 0$. Thus,

$$N \leq \frac{\ln u_0}{-\ln\left(1-\frac{1}{\alpha}\right)} + 1.$$

Next, we apply the McLaurin expansion $\ln(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + ...$, which converges for all real numbers $-1 < x \leq 1$. Since $\alpha > 1$, we have $-1 < \frac{-1}{\alpha} < 0$. Hence $-\ln\left(1-\frac{1}{\alpha}\right) = \sum_{i\geq 1}\frac{1}{i}\left(\frac{1}{\alpha}\right)^i = \sum_{i\geq 1}\frac{1}{i\alpha^i}$. Using this expansion in the above inequality we obtain

$$N \leq \frac{\ln u_0}{\sum_{i\geq 1}\frac{1}{i\alpha^i}} + 1.$$

We then notice that $\sum_{i\geq 1}\frac{1}{i\alpha^i} > \frac{1}{\alpha}$, since α is a positive integer, and hence

$$\frac{\ln u_0}{\sum_{i\geq 1}\frac{1}{i\alpha^i}} < \frac{\ln u_0}{\frac{1}{\alpha}} = \alpha \ln u_0.$$

Therefore, $N < \alpha \ln u_0 + 1$. Since N is an integer, the result follows. ∎

We now give an upper bound on the number of tests conducted by Algorithm 5.1.

Theorem 5.2.5 *Let A be a set of $k \geq 4$ factors whose corresponding item-set contains $d \geq 0$ faulty interactions, each having strength at most* 3. *Let N^* be the maximum number of tests conducted by* LocateErrorsInSet(A, T).
If $d = 0$, then $N^* = 1$.
If $d = 1$, then $N^* < 3 \ln \binom{k}{3} + 7 + \frac{10}{\ln 3} \ln k < 12.55 \log_2 k + 7$.
If $d \geq 2$, then $N^* < \big(4.5d^3 - 4.5d^2 + d\big) \ln \binom{k}{3} + d\big(7 + \frac{10}{\ln 3} \ln k\big) < 9.36d^3 \log_2 k$

Proof: At each iteration of the while-loop, some subset of A is assigned to A'. Let N_p be the number of times that TestSet(A') returns pass, and let N_f be the number of times that TestSet(A') returns fail. Then the total number of iterations of the while-loop is $N = N_p + N_f$.

Index the iterations where TestSet(A') returns pass by the integer $i \in [1, N_p]$. Consider the iteration labeled i. Let U_i be the set U at iteration i, and let $u_i = |U_i|$. Also, let $\mathcal{T}_i = \big\{\tau \subseteq A' : |\tau| = 3\big\} \cap U_i$, and let $t_i = |\mathcal{T}_i|$.

By Lemma 5.1.4, $t_i \geq \frac{1}{\alpha} u_i$, for the value of α specified in that lemma. Since A' corresponds to a passing set at each of these iterations and $\mathcal{T}_i \subseteq U$, exactly t_i triples are removed from U each time there are no iterations between the ith and $(i+1)$st in which TestSet(A') returns fail, and we have $u_{i+1} = u_i - t_i$ in this case. If there is at least one such iteration, then at least one more triple is removed from U, and we have $u_{i+1} < u_i - t_i$. Therefore $u_{i+1} \leq u_i - t_i$, and hence

$$u_{i+1} \leq \left(1 - \frac{1}{\alpha}\right) u_i.$$

By Lemma 5.2.4, we have $N_p \leq \alpha \ln u_0$. Since $u_0 = \binom{k}{3}$, we have

$$N_p \leq \alpha \ln \binom{k}{3}.$$

At each iteration in which $\textsc{TestSet}(A')$ returns fail, this algorithm conducts one test for A', plus the number of tests conducted within $\textsc{LocateTriple}(A')$ and, subsequently, $\textsc{LocateErrorsInTriple}(\tau)$. Then the total number of tests conducted within one iteration where A' corresponds to a failing set of items is at most $7 + 10\log_3 k$, by Lemmas 5.2.3 and 5.2.1.

It is easy to see that $N_f \leq d$. Let N^* be the maximum number of tests conducted by this algorithm. Then we have

$$N^* \leq N_p + N_f(7 + 10\log_3 k) \leq \alpha \ln \binom{k}{3} + d(7 + 10\log_3 k).$$

If $d \geq 2$, then $\alpha = \binom{3d}{3}$, and

$$N^* \leq \binom{3d}{3} \ln \binom{k}{3} + d(7+10\log_3 k) = \big(4.5d^3 - 4.5d^2 + d\big) \ln \binom{k}{3} + d\bigg(7 + \frac{10}{\ln 3}\ln k\bigg).$$

Since $\binom{k}{3} < k^3$, we have $\ln \binom{k}{3} < \ln k^3 = 3\ln k$. We apply this inequality to the preceding expression to get

$$N^* < 3\big(4.5d^3 - 4.5d^2 + d\big)\ln k + 7d + \frac{10d}{\ln 3}\ln k.$$

We notice that $7d < 7d\ln k$ since $k \geq 4$. The inequality then becomes

$$N^* < \left(13.5d^3 - 13.5d^2 + \left(10 + \frac{10}{\ln 3}\right)d\right)\ln k.$$

Next, we see that $10 + \frac{10}{\ln 3} < 19.11$, and for all $d \geq 2$ we have $-13.5d^2 + 19.11d < 0$. Therefore $13.5d^3 - 13.5d^2 + \big(10 + \frac{10}{\ln 3}\big)d < 13.5d^3 - 13.5d^2 + 19.11d < 13.5d^3$, and hence,

$$N^* < 13.5d^3 \ln k = 13.5(\ln 2)d^3 \log_2 k < 9.36d^3 \log_2 k.$$

If $d \leq 1$, then $\alpha = 3$. It is easy to see that exactly one test is conducted when $d = 0$, so suppose that $d = 1$. Then

$$N^* \leq 3 \ln \binom{k}{3} + 7 + \frac{10}{\ln 3} \ln k.$$

We apply the inequality $\ln \binom{k}{3} < 3 \ln k$, and combine like terms to get

$$N^* < \left(9 + \frac{10}{\ln 3}\right) \ln k + 7.$$

Since $9 + \frac{10}{\ln 3} < 18.11$, we have

$$N^* < 18.11 \ln k + 7 = 18.11 (\ln 2) \log_2 k + 7 < 12.55 \log_2 k + 7.$$

We conclude that N^* satisfies the conditions presented in the statement of this theorem. ∎

In the next section, we generalize Algorithm 4.4 to strength $\overline{3}$.

5.3 Strength-$\overline{3}$ ELAs with Safe Values via CGT

In the preceding chapter, we presented two algorithms for adaptively building an ELA for a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ that has $g = \max\{g_1, g_2, ..., g_k\}$ fixed, and safe values $s_1, s_2, ..., s_k$. The first, Algorithm 4.2, can be applied to systems with only strength-1 errors, and the second, Algorithm 4.4, can be applied to systems with errors of strengths up to two.

We now give an algorithm that generalizes this procedure up to strength 3.

Algorithm 5.5 Strength-$\overline{3}$ generalization of Algorithm 4.4 on Page 101.
Precondition: $TP(k, (g_1, g_2, ..., g_k))$ has safe values $s_1, s_2, ..., s_k$ ($k \geq 2$ is an integer), the set of all errors is independent, each error has strength at most 3, and $g = \max\{g_1, g_2, ..., g_k\}$ is fixed. This algorithm returns the set E of all errors in the $TP\big(k, (g_1, g_2, ..., g_k)\big)$.

```
procedure ErrorLocateWithSafeValues($k, (s_1, s_2, ..., s_k), (g_1, g_2, ..., g_k)$)
    ▷ Relabel values in each factor so that $s_1 = s_2 = ... = s_k = 0$.
    $S \leftarrow \emptyset$
    for each factor $f \in [1, k]$ such that $s_f \neq 0$ do
        $S \leftarrow S \cup \{f\}$, and swap labels $(f, 0)$ and $(f, s_f)$.

    ▷ Identify the defective items via CGT.
    $E \leftarrow \emptyset$, and let $g = \max\{g_1, g_2, ..., g_k\}$
    Let $\mathcal{M}$ be an $MCA(N; 3, k, (g_1 - 1, g_2 - 1, ..., g_k - 1))$.
    Denote the $i$th row (test) of $\mathcal{M}$ by $T(i) = T(i)_1 T(i)_2 ... T(i)_k$.
    for $i \leftarrow 1$ to $N$ do
        for all $f \in [1, k]$ such that $T(i)_f = 0$ do $T(i)_f \leftarrow g_f - 1$
        if Test$\big(T(i)\big)$ = fail then
            Let $A = \big\{f \in [1, k] : T(i)_f \neq 0\big\}$.
            $E' \leftarrow$ LocateErrorsInSet$\big(A, T(i)\big)$
            for $D \in E'$ do $E \leftarrow E \cup \big\{\{(f, T(i)_f) : f \in D\}\big\}$

    ▷ Restore the original labels to every factor whose label we previously changed.
    for $f \in S$ do swap labels $(f, 0)$ and $(f, s_f)$.
    return $E$
```

Corollary 5.3.1 *Let $g = \max\{g_1, g_2, ..., g_k\}$ be fixed. Then the number of tests performed by Algorithm 5.5, as $k \to \infty$, satisfies*

$$N^* \leq \big(3(g-1)^3 \ln k + \mathcal{O}(1)\big)\left(\big(4.5d^3 - 4.5d^2 + d\big) \ln \binom{k}{3} + d\left(7 + \frac{10}{\ln 3} \ln k\right) + 1\right)$$

Proof: As in Corollary 4.3.1, an $MCA\big(N; 3, k, (g_1 - 1, g_2 - 1, ..., g_k - 1)\big)$ can be built, by the method of [6], with $N \leq 3(g-1)^3 \ln k + \mathcal{O}(1)$.

ErrorLocateWithSafeValues performs N tests in the main for-loop (even when $d = 0$), and it calls the procedure LocateErrorsInSet at most N times (once for each failing test in the covering array). By Theorem 5.2.5, the result follows. ∎

5.4 Comparison to $d(H)$-Disjunct Matrix Method

We now compare Algorithm 5.1 with the $d(H)$-disjunct matrix method of Chen et al. [9], in terms of maximum number of tests conducted on a set A of k items (vertices) with d errors (edges) of strength (cardinality) at most 3. By Theorem 4.4.2, if $d \leq k-3$, then there exists a $d(H)$-disjunct matrix that has at most N rows (tests), where

$$N < \left(\frac{d+3}{3}\right)^3 \left(\frac{d+3}{d}\right)^d \left(1 + (d+3)\left(1 + \ln\left(1 + \frac{k}{d+3}\right)\right)\right).$$

We first note that when $d = 0$, our adaptive algorithm conducts exactly one test, by Theorem 5.2.5. There is no $0(H)$-disjunct matrix; however, one could always test the entire set A prior to executing any CGT algorithm, to determine whether or not $d = 0$. Therefore, we compare the two methods for $d = 1$ and $d \geq 2$. In the tables that follow, we use $N_{adaptive}$ and $N_{d(H)}$ to denote the maximum number of tests conducted by Algorithm 5.1 and the $d(H)$-disjunct matrix method as stated in Theorems 5.2.5 and 4.4.2, respectively.

If $d = 1$, then by Theorem 5.2.5, $N_{adaptive} = 3 \ln \binom{k}{3} + 7 + \frac{10}{\ln 3} \ln k$ tests. The case where $k = 3$ is trivial, so we give a table for values of k beginning with $k = 4$.

k	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$
4	24	74	0.323
5	29	78	0.365
6	32	82	0.393
7	**35**	86	0.412
8	38	89	0.427
9	40	92	0.437
10	42	**95**	0.446
20	55	115	0.480
30	63	129	0.489
40	68	138	0.493
50	72	146	0.495
100	85	171	**0.497**
1,000	127	257	0.493
10,000	168	344	0.489
100,000	210	431	0.487
1,000,000	252	519	0.485
1,000,000,000	377	781	0.483
1,000,000,000,000	502	1,043	0.481
googol	4,170	8,728	0.478

Table 5.1: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $d = 1$.

Consider Table 5.1. We used bold font to indicate the first time each method conducts fewer tests than an exhaustive method, which would conduct at most $\binom{k}{3}+6$ tests (one test per triple, plus one call to LocateErrorsInTriple once a failing triple is found).

Furthermore, our adaptive algorithm conducts at most half as many tests as the disjunct matrix method, if the latter is given the foreknowledge that $d = 1$ (the ratio is greatest when $k = 112$; $\frac{N_{adaptive}}{N_{d(H)}} \approx 0.4965866 < 0.497$). However, our algorithm does not need to know d in advance; a $\hat{d}(H)$-disjunct matrix method with $\hat{d} > d$ conducts a far greater number of tests as $\hat{d}/d$ increases, which we shall see in subsequent tables.

If $d \geq 2$, then by Theorem 5.2.5, we have

$$N_{adaptive} = \big(4.5d^3 - 4.5d^2 + d\big) \ln \binom{k}{3} + d\bigg(7 + \frac{10}{\ln 3} \ln k\bigg).$$

Consider Table 5.2. If $k = 10$, an exhaustive method would conduct at most $\binom{10}{3} + 6d$ tests (one test per triple, plus one call to LocateErrorsInTriple per failing triple). We see that if $k = 10$, then either method may conduct more than $\binom{10}{3} + 6d$ tests, even for $d = 2$.

d	$\binom{10}{3} + 6d$	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$
2	132	152	333	0.456
3	138	486	825	0.589
4	144	1,165	1,693	0.688

Table 5.2: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $k = 10, d \in [2, 4]$.

For small values of k, the exhaustive method (i.e. every triple is tested, and every subset of every failing triple is also tested) may be superior to both Algorithm 5.1 and the use of a $d(H)$-disjunct matrix. In practice, however, both our algorithm and the method of Chen et al. may conduct significantly fewer tests than the upper bounds given in Theorems 5.2.5 and 4.4.2, depending on the structure of the error hypergraph H.

Next, consider Table 5.3. Let N_e be the maximum number of tests conducted by the exhaustive method. If $k = 100$, then $N_e = \binom{100}{3} + 6d$ and for values of d contained in Table 5.3 (that is, $d \in [2, 15]$), we have $161,712 \leq N_e \leq 161,790$. When $k = 100$, each method may conduct more tests than the exhaustive method if $d \geq 15$.

d	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$	$N_{(2d)(H)}$	$\frac{N_{adaptive}}{N_{(2d)(H)}}$
2	338	614	0.550	3,228	0.105
3	1,154	1,551	0.744	9,979	0.116
4	2,834	3,228	0.878	23,574	0.120
5	5,702	5,930	0.962	47,239	**0.121**
6	10,080	9,979	1.010	84,680	0.119
7	16,294	15,731	1.036	140,047	0.116
8	24,666	23,574	1.046	**217,913**	0.113
9	35,521	33,927	**1.047**	$> N_e$	0.110
10	49,183	47,239	1.041	$> N_e$	0.107
11	65,975	63,988	1.031	$> N_e$	0.103
12	86,221	84,680	1.018	$> N_e$	0.100
13	110,245	109,846	1.004	$> N_e$	0.097
14	138,370	140,047	0.988	$> N_e$	0.095
15	**170,922**	**175,866**	0.972	$> N_e$	0.092

Table 5.3: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $k = 100, d \in [2, 15]$.

We notice that, even when the exact value of d is foreknown, $N_{adaptive} \leq 1.047 N_{d(H)}$. Now, let $\hat{d} \geq d$ be an upper bound on d. From the entries in the table, it is easy to see that for all $\hat{d} \geq d + 1$, we have $N_{adaptive} < N_{\hat{d}(H)}$. If $\hat{d}$ is overestimated by a factor of 2, then $N_{adaptive} < 0.121 N_{\hat{d}(H)}$. Furthermore, a $16(H)$-disjunct matrix may contain as many as $217,913 > 162,000$ tests, that is, far more than the exhaustive testing method, even though only 8 errors are present.

Consider Table 5.4. If $k = 1,000$, then $N_e = \binom{1,000}{3} + 6d \approx 166 \times 10^6$ for values of d contained in Table 5.4 (that is, $2 \leq d \leq 126$). We remark here that when $k = 1,000$, $N_{adaptive}$ first exceeds N_e when $d = 126$, and $N_{d(H)}$ first exceeds N_e earlier with respect to d, that is, when $d = 88$.

d	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$	$N_{(2d)(H)}$	$\frac{N_{adaptive}}{N_{(2d)(H)}}$
2	518	941	0.5505	5,097	0.1016
3	1,800	2,415	0.7453	16,139	0.1115
4	4,444	5,097	0.8719	38,880	**0.1143**
5	8,962	9,483	0.9451	79,224	0.1131
6	15,865	16,139	0.9830	144,098	0.1101
7	25,664	25,702	0.9986	241,407	0.1063
8	38,870	38,880	**0.9997**	379,986	0.1023
9	55,995	56,442	0.9921	569,569	0.0983
10	77,548	79,224	0.9788	820,754	0.0945
11	104,043	108,123	0.9623	1,144,979	0.0909
12	135,988	144,098	0.9437	1,554,494	0.0875
13	173,896	188,168	0.9242	2,062,339	0.0843
14	218,278	241,407	0.9042	2,682,327	0.0814
15	269,644	304,950	0.8842	3,429,022	0.0786
20	649,131	820,754	0.7909	9,623,387	0.0675
30	2,225,817	3,429,022	0.6491	41,866,556	0.0532
40	5,318,676	9,623,387	0.5527	119,664,329	0.0444
50	10,438,778	21,575,354	0.4838	$> N_e$	0.0385
88	57,394,799	**169,595,300**	0.3384	$> N_e$	0.0267
126	**169,047,461**	631,874,470	0.2675	$> N_e$	0.0211

Table 5.4: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $k = 1,000$, some values of $d \in [2, 126]$.

From the table, it is apparent that $N_{adaptive} < N_{d(H)}$ for all values of d in the table when $k = 1,000$. If $\hat{d}$ is overestimated by a factor of 2, then $N_{adaptive} \leq 0.1143 N_{d(H)}$.

Next, consider Table 5.5. If $k = 10,000$, then $N_e = \binom{10,000}{3} + 6d \approx 166.6 \times 10^9$ for values of d contained in Table 5.5 (that is, $2 \leq d \leq 1,128$). We remark here that when $k = 10,000$, $N_{adaptive}$ first exceeds N_e when $d = 1,128$, and $N_{d(H)}$ first exceeds N_e much earlier with respect to d, that is, when $d = 483$.

d	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$	$N_{(2d)(H)}$	$\frac{N_{adaptive}}{N_{(2d)(H)}}$
2	698	1,273	0.5483	7,012	0.0995
3	2,443	3,297	0.7410	22,490	0.1086
4	6,048	7,012	0.8625	54,765	**0.1104**
5	12,211	13,134	0.9297	112,632	0.1084
6	21,630	22,490	0.9618	206,559	0.1047
7	35,002	36,018	**0.9718**	348,626	0.1004
8	53,025	54,765	0.9682	552,491	0.0960
9	76,396	79,884	0.9563	833,344	0.0917
10	105,815	112,632	0.9395	1,207,876	0.0876
20	886,026	1,207,876	0.7335	14,797,949	0.0599
50	14,249,561	33,720,869	0.4226	448,282,453	0.0318
100	115,124,242	448,282,453	0.2568	6,081,130,416	0.0189
200	925,574,940	6,081,130,415	0.1522	82,530,408,310	0.0112
300	3,129,004,060	27,982,974,551	0.1118	$> N_e$	0.0083
400	7,423,063,565	82,530,408,310	0.0899	$> N_e$	0.0067
483	13,074,670,182	**167,501,393,845**	0.0781	$> N_e$	0.0058
1,128	**166,736,249,646**	$> N_e$	0.0418	$> N_e$	0.0032

Table 5.5: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $k = 10,000$, some values of $d \leq 1,128$.

Again, we have $N_{adaptive} < N_{d(H)}$ for all values of d in the table when $k = 10,000$; in particular, $N_{adaptive} \leq 0.9718 N_{d(H)}$. If $\hat{d}$ is overestimated by a factor of 2, then $N_{adaptive} \leq 0.1104 N_{d(H)}$.

We now consider Table 5.6. If $k = 100,000$, then $N_e = \binom{100,000}{3} + 6d \approx 166.66 \times 10^{12}$ for values of d contained in Table 5.5 (that is, $2 \leq d \leq 10,236$). We remark here that when $k = 100,000$, $N_{adaptive}$ first exceeds N_e when $d = 10,420$, and $N_{d(H)}$ first exceeds N_e much earlier with respect to d, that is, when $d = 2,631$.

d	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$
2	879	1,606	0.5473
3	3,086	4,181	0.7381
4	7,652	8,932	0.8567
5	15,459	16,796	0.9204
6	27,392	28,861	0.9491
7	44,336	46,369	**0.9562**
8	67,174	70,711	0.9500
9	96,791	103,425	0.9359
10	134,071	146,194	0.9171
100	145,902,281	632,012,933	0.2309
1,000	147,214,224,265	4,207,070,302,481	0.0350
2,631	2,682,757,183,342	**166,690,234,140,180**	0.0161
5,000	18,416,496,832,237	$> N_e$	0.0098
10,420	**166,703,738,743,611**	$> N_e$	0.0057

Table 5.6: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $k = 100,000$, some values of $d \leq 10,420$.

Yet again, we have $N_{adaptive} < N_{d(H)}$ for all values of d in the table when $k = 100,000$; in particular, $N_{adaptive} \leq 0.9562 N_{d(H)}$. We also notice that $\frac{N_{adaptive}}{N_{d(H)}}$ decreases rapidly as d increases past 10.

If $k = 1,000,000$, then $N_e = \binom{1,000,000}{3} + 6d \approx 166.666 \times 10^{15}$ for values of d contained in Table 5.7 (that is, $2 \leq d \leq 97,750$). We remark here that when $k = 1,000,000$, $N_{adaptive}$ first exceeds N_e when $d = 97,750$, and $N_{d(H)}$ first exceeds N_e when $d = 14,367$.

d	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$
2	1,059	1,940	0.5459
3	3,729	5,065	0.7362
4	9,255	10,853	0.8528
5	18,707	20,458	0.9144
6	33,155	35,234	0.9410
7	53,670	56,724	**0.9462**
8	81,323	86,663	0.9384
9	117,185	126,975	0.9229
10	162,326	179,770	0.9030
100	176,679,238	816,407,667	0.2164
1,000	178,268,187,369	5,926,134,514,674	0.0301
14,367	529,146,573,279,085	**166,703,447,853,592,306**	0.0032
50,000	22,305,361,570,089,421	$> N_e$	0.0012
97,750	**166,668,603,171,383,725**	$> N_e$	0.0007

Table 5.7: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $k = 1,000,000$, some values of $d \leq 97,750$.

When $k = 100,000$, we have $N_{adaptive} \leq 0.9462 N_{d(H)}$ for all values of d in the table. Again, $\frac{N_{adaptive}}{N_{d(H)}}$ decreases rapidly as d increases past 10.

Next, we try fixing d to some values greater than one, and varying k from values as low as 100 up to as high as one googol. In Table 5.8, we compare $N_{adaptive}$ with $N_{d(H)}$ and $N_{(2d)(H)}$ for the cases in which $d = 10^p, p \in [2, 4]$.

d	k	$N_{adaptive}$	$N_{d(H)}$	$\frac{N_{adaptive}}{N_{d(H)}}$	$\frac{N_{adaptive}}{N_{(2d)(H)}}$
10	100	49,183	47,239	1.0412	0.1066
10	1,000	77,548	79,224	0.9788	0.0599
10	10,000	105,815	112,632	0.9395	0.0407
10	100,000	134,071	146,194	0.9171	0.0308
10	1,000,000	162,326	179,770	0.9030	0.0247
10	10^9	247,091	280,505	0.8809	0.0156
10	10^{12}	331,856	381,240	0.8705	0.0114
10	googol	2,818,303	3,336,131	0.8448	0.0013
100	1,000	84,335,365	270,846,403	0.3114	0.0155
100	10,000	115,124,242	448,282,453	0.2568	0.0088
100	100,000	145,902,281	632,012,933	0.2309	0.0060
100	1,000,000	176,679,238	816,407,667	0.2164	0.0045
100	10^9	269,009,760	1,369,806,297	0.1964	0.0026
100	10^{12}	361,340,270	1,923,213,162	0.1879	0.0019
100	googol	3,069,701,877	18,156,481,439	0.1691	0.0002
1,000	10,000	116,159,168,645	2,545,436,931,660	0.0456	0.0016
1,000	100,000	147,214,224,265	4,207,070,302,481	0.0350	0.0009
1,000	1,000,000	178,268,187,369	5,926,134,514,674	0.0301	0.0006
1,000	10^9	271,429,726,012	11,102,762,581,525	0.0244	0.0003
1,000	10^{12}	364,591,251,196	16,280,140,519,714	0.0224	0.0002
1,000	googol	3,097,329,322,860	168,149,915,408,123	0.0184	0.0000

Table 5.8: Algorithm 5.1 vs. Chen et al.'s $d(H)$-disjunct matrix method [9] for $d = 10^p, p \in [2, 4]$.

We see that the ratio $\frac{N_{adaptive}}{N_{d(H)}}$ steadily decreases as k increases. Furthermore, $\frac{N_{adaptive}}{N_{(2d)(H)}}$ decreases quite rapidly as k increases. We conclude that the larger the value of k, the more advantageous it is to choose Algorithm 5.1 over the $d(H)$-disjunct matrix method [9], particulary if d is large, and even more so if d is not known in advance.

Chapter 6

Conclusion

In this thesis, we have given two new algorithms for identifying faulty interactions in testing problems. Each algorithm applies to a particular type of testing problem, and we discuss the results, and relevant open problems related to each result, in Sections 6.1 and 6.2, respectively. We then discuss some additional related open problems in Section 6.3.

6.1 Robust Error Location for Binary Alphabets

In Chapter 3, we gave a new adaptive algorithm, called LocateAllErrors, for constructing a strength-$\overline{2}$ error locating array (ELA) for a $TP(k, 2)$ with at most two faulty interactions, without knowledge of safe values (see Algorithm 3.3 on Page 64). For the special case where at most two errors are present, LocateAllErrors is an improvement on the DiscoverEdges algorithm of Martínez et al. [32] in two key ways.

1. Their algorithm assumes that all errors are locatable, and ours does not.
2. Their algorithm finds a passing test by using a randomized selection process, while ours finds a passing test by an efficient deterministic process.

We analyzed four types of graphs (a, b, c, and d) corresponding to nonlocatable errors, as seen in Figure 3.1 on Page 39 (and originally defined in [32]). We gave some refinements to the notion of locatability (see Definitions 3.1.2 and 3.1.3 on Pages 40 and 41); these new insights led us to see that some faulty interactions are "more nonlocatable" than others, as shown in Corollary 3.1.4 on Page 41. In particular, we discovered that a passing test exists for a $TP(k, 2)$ if and only if its associated error graph G does not contain a type-d subgraph (see Theorem 3.2.2 on Page 44). Subsequently, we gave an efficient deterministic procedure for either finding a passing test, or determining that one does not exist (see Version 1 of Algorithm 3.1 on page 51).

Our insights into the particularities of nonlocatability have enabled our algorithm to handle the cases where errors corresponding to a nonlocatable subgraph (of type a, b, or c) are present in the $TP(k, 2)$. In all cases where a passing test exists for the given $TP(k, 2)$, Algorithm 3.3 returns the set of errors corresponding to the edges of either G or a graph that is location-equivalent to G.

The number of tests conducted by both our algorithm and the algorithm of Martínez et al. is related to the quantity $N = 2\big(1+o(1)\big)(\log k)^2+\mathcal{O}(\log k)$. However, the number of tests conducted by their algorithm, DiscoverEdges, is *expected* to be N, due to the randomized process of finding a passing test, while Algorithm 3.3 conducts *at most* N tests, since our algorithm uses a deterministic procedure for finding a passing test. Therefore, if at most two faulty interactions are present in a $TP(k, 2)$ without known safe values, Algorithm 3.3 is both more robust and more efficient than the previously-known algorithm that can be applied to such a testing problem, DiscoverEdges [32].

Several open problems exist for identifying errors in testing problems for which safe values are not known.

One could generalize either DiscoverEdges or Algorithm 3.3 for testing problems with larger alphabets, i.e. $TP(k, g)$s with $g \geq 3$, or for testing problems with mixed alphabets, i.e. $TP\big(k,(g_1,g_2,...,g_k)\big)$s with $g_i \neq g_j$ for some $i \neq j$. Since our algorithm proceeds case-by-case, and relies on the fact that the number of cases to consider is relatively small for a $TP(k,2)$ with at most two faulty interactions, it may not easily generalize to testing problems with larger or mixed alphabets.

However, it may not need to be generalized directly. That is, a more complicated case-by-case algorithm may not be needed. Consider a $TP\big(k,(g_1,g_2,...,g_k)\big)$ with $g_i \geq 3$ for some $i \in [1,k]$, and several faulty interactions of strengths up to 2. We can construct a $TP(k,2)$ from the $TP\big(k,(g_1,g_2,...,g_k)\big)$ in the following way.

Let G be the (unknown, k-partite) error graph associated with the given $TP\big(k,(g_1,g_2,...,g_k)\big)$, and let $V' \subset V(G)$ such that exactly two vertices from each of the k parts of G are stored in V'. Then $G[V']$ is a $G_{k,2}$ whose edges are unknown. Now, suppose that we have an algorithm that, when applied to the $TP\big(k,(g_1,g_2,...,g_k)\big)$, can determine that for some $V' \subset V(G)$, the induced subgraph $G[V']$ has at most two edges. In this case, Algorithm 3.3 can be applied to the $TP(k,2)$ associated with $G[V']$, after a temporary and appropriate relabeling of some vertices in G.

We believe that the procedure for finding a passing test will easily generalize to a larger alphabet due to its symmetrical nature. This procedure, or some variation of it, may yield enough information about errors present in a $TP\big(k,(g_1,g_2,...,g_k)\big)$ in order to allow error identification in such a testing problem, via the method suggested in the preceding two paragraphs.

To generalize Algorithm 3.3 to identify more than two errors may require an approach other than a case-by-case analysis, since the presence of more than two errors could greatly increase the number of subcases. More research is needed here.

Development of higher-strength versions of DiscoverEdges and Algorithm 3.3 also seem to be significant problems, particularly since adaptive error location with safe values has only been done for strength $\overline{3}$ in this thesis, and identifying errors is significantly more difficult when safe values are not known.

6.2 Combinatorial Group Testing and Error Location for Strengths Greater than Two

In Chapter 5, we gave the first adaptive CGT algorithm that can identify faulty interactions of strengths up to three. In particular, LocateErrorsInSet (given as Algorithm 5.1 on Page 107) is a strength-$\overline{3}$ generalization of the strength-$\overline{2}$ CGT algorithm by Martínez et al. [32].

We have compared LocateErrorsInSet to the current-best nonadaptive CGT method that can be applied to item-sets with faulty interactions of strengths up to three, and showed that our method excels at identifying more faulty interactions in fewer tests than the competing method of Chen et al. [9], particularly as both k, the number of items in the given set, and d, the number of faulty interactions among those items, grow very large.

The nonadaptive method of Chen et al. [9] requires advance knowledge of an upper bound on d, while adaptive algorithms such as ours do not require such knowledge. This is a key advantage, as can be seen in Tables 5.1 through 5.7 on Pages 129 through 135. In particular, the maximum number of tests conducted by Algorithm 5.1 is lower than the maximum number of tests conducted by the nonadaptive, disjunct-matrix method of Chen et al. given the following parameters:

1. $k \geq 4$ and $d = 1$.
2. $k \geq 1,000$ and $d \geq 2$.

Furthermore, if the upper bound on d is overestimated even by one, then the maximum number of tests conducted by Algorithm 5.1 is lower than the maximum number of tests conducted by the method of Chen et al., and far lower if the upper bound on d is overestimated by a factor of two.

Further generalizations of Algorithm 5.1 to strengths greater than three are needed, particularly since testing with strengths up to $t \in [4, 6]$ is needed in several applications; Kuhn et al. [27] give empirical results showing that 4-way testing is needed to detect at least 95% of faulty interactions in the applications they tested, and 6-way testing is needed to identify all faulty interactions. We believe that the next step is to generalize Algorithm 5.1 to strength $\bar{t}$ for any fixed integer t that is reasonably small with respect to k.

We also gave a new adaptive algorithm for generating a strength-$\overline{3}$ error locating array (ELA) for a graph associated with a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ that has safe values. We showed how such an ELA can be generated by applying Algorithm 5.1 to each failing test in an $MCA\big(N; 3, k, (g_1 - 1, g_2 - 1, ..., g_k - 1)\big)$. Algorithm 5.5 is a generalization of the strength-$\overline{2}$ ELA-building algorithm by Martínez et al. [32], and it will easily generalize to strength-$\bar{t}$ for some fixed integer t, once there is an adaptive strength-$\bar{t}$ CGT algorithm.

6.3 Other Related Open Problems

Recall that Error Locating Arrays are related to Mixed Covering Arrays (MCAs), by Theorems 2.2.12 and 2.2.13 on Pages 33 and 34. Bryce and Colbourn give a method that adaptively generates an $MCA\big(N; t, k, (g_1, g_2, ..., g_k)\big)$ for a given $TP\big(k, (g_1, g_2, ..., g_k)\big)$, such that N, the number of rows, is logarithmic in k if t and $g = \max_i\{g_i\}$ are fixed [5].

We have shown that Algorithm 5.5 generates an $ELA(N'; 3, H)$ for a hypergraph H associated with a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ such that N' is also logarithmic in k if $g = \max_i\{g_i\}$ is fixed. Assuming further generalizations of Algorithms 5.1 and 5.5 to strength $\bar{t}$ for some fixed positive integer t, it will soon be possible to adaptively generate an $ELA(N'; t, H')$ for a hypergraph H' whose edges have cardinalities up to t, such that N' is also logarithmic in k.

However, these arrays can only be applied to testing problems where every interaction may be tested. Forbidden configurations often arise out of product constraints in the context of computer hardware, software, and network testing; this has been studied by Cohen et al. [11]. A new type of array has been created for use in troubleshooting such systems with constraints. Covering Arrays avoiding Forbidden Edges (CAFEs), and their close relation to ELAs, have been studied by Danziger et al. [16] and Maltais [31]. In particular, Danziger et al. give a recursive construction for CAFEs, and an explicit equation that, given an error graph G associated with a testing problem, relates the minimum number of rows in an $ELA(G)$ with the minimum number of rows in a $CAFE(G)$.

Loosely speaking, CAFEs can be described as strength-2 covering arrays with restrictions. That is, certain specified interactions must *not* be covered by any row of the CAFE, but every other pairwise interactions must appear in at least one row of the CAFE. In other words, the rows of an ELA that are passing tests collectively form a CAFE whose forbidden edges are precisely the edges identified by the ELA. Now that an efficient, adaptive algorithm for constructing a strength-$\overline{3}$ ELA is known (in the case that safe values are also known), further research may yield a generalization to CAFEs so that certain hyperedges (in a hypergraph associated with a given testing problem) may be avoided - i.e. a Covering Array avoiding Forbidden Hyperedges (CAFH).

Now, consider a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ that has safe values and whose associated hypergraph H has (forbidden) hyperedges whose cardinalities are at most 3. Also consider an iteration of Algorithm 5.5 in which $E \neq \emptyset$. In all subsequent iterations, every test conducted by this algorithm avoids all errors in E corresponding to hyperedges in H. That is, given safe values for a $TP\big(k, (g_1, g_2, ..., g_k)\big)$ and an initial set of errors E that correspond to forbidden hyperedges, Algorithm 5.5 adaptively constructs a CAFH for H. Therefore we have given a method for constructing a CAFH when safe values are known. Constructing a CAFH when safe values are not known may be a significantly more challenging problem.

An additional problem would be to actually identify all faulty interactions in a given testing problem, even in the presence of constraints yielding forbidden interactions; an array that can be applied in such a situation would be either an Error Locating Array avoiding Forbidden Edges (ELAFE) or, similarly, an ELAFH if the testing problem is associated with a hypergraph, rather than just a graph.

It is our hope that the results developed here enable further progress in the study of testing problems with faulty interactions of strengths greater than two.

Bibliography

[1] T. Andreae, A search problem on graphs which generalizes some group testing problems with two defectives, *Discrete Math.* **88** (1991), 121–127.

[2] J. Azar, R. Motwani and J. Naor, Approximating probability distributions using small sample spaces, *Combinatorica* **18** (1998), 151–171.

[3] J. A. Bondy and U. S. R. Murty, Graph Theory With Applications, *Springer*, New York, 2008.

[4] M. Bouvel, V. Grebinski and G. Kucherov, Combinatorial search on graphs motivated by bioinformatics applications: a brief survey, *Graph Theoretic Concepts in Computer Science*, 2005, Springer, 16–27.

[5] R.C. Bryce and C.J. Colbourn, A density algorithm for pairwise interaction testing, *Softw. Test. Verif. Reliab.* **17** (2007), 159–182.

[6] R.C. Bryce and C.J. Colbourn, A density-based greedy algorithm for higher-strength covering arrays, *Softw. Test. Verif. Reliab.* **19** (2009), 37–53.

[7] K. Burr and W. Young, Combinatorial test techniques: Table-based automation, test generation, and code coverage, *Proc. Intl. Conf. on Soft. Test. Anal. and Rev.*, New York, October 1998, ACM, 503–513.

[8] H. B. Chen, D. Z. Du and F. K. Hwang, An unexpected meeting of four seemingly unrelated problems: graph testing, DNA complex screening,

superimposed codes and secure key distribution, *J. Comb. Optim.* **14** (2007), 121–129.

[9] H. B. Chen, H. L. Fu and F. K. Hwang, An upper bound of the number of tests in pooling designs for the error-tolerant complex model, *Optim. Letters* **2** (2008), 425–431.

[10] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, The AETG System: An Approach to Testing Based on Combinatorial Design, *IEEE Transactions On Software Engineering* **23** (1997), 437–444.

[11] M. B. Cohen, M. B. Dwyer, J. Shi, Interaction testing of highly-configurable systems in the presence of constraints, *Inter. Symp. Softw. Test. and Analysis, ISSTA '07* (London) **23** (2007), 129–139.

[12] C.J. Colbourn, Combinatorial aspects of covering arrays, *Le Matematiche* (Catania) **58** (2004), 121–167.

[13] C.J. Colbourn, S.S. Martirosyan, G.L. Mullen, D.E. Shasha, G.B. Sherwood, and J.L Yucas, Products of mixed covering arrays of strength two, *J. Comb. Des.* **14** (2006), 124–138.

[14] C.J. Colbourn and D.W. McClary, Locating and detecting arrays for interaction faults, *J. Comb. Optim.* **15** (2008), 17–48.

[15] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, Model-based Testing in Practice, *Proc. of the Intl. Conf. on Software Engineering, ICSE '99* (New York), 1999, 285–294.

[16] P. Danziger, E. Mendelsohn, L. Moura, and B. Stevens, Covering arrays avoiding forbidden edges, *Theor. Comput. Sci.* **410** (2009), 5403-5414.

[17] D.Z. Du and F.K. Hwang, Combinatorial group testing and its applications, *World Scientific Publishing Cp. Inc.*, River Edge, NJ, 2000.

[18] P. Flajolet and R. Sedgewick. Analytic Combinatorics, *Cambridge University Press*, 2009.

[19] H. Gao, F.K. Hwang, M.T. Thai and W. Wu, Construction of $d(H)$-disjunct matrix for group testing in hypergraphs, *J. Comb. Optim.* **12** (2006), 297–301.

[20] L. Gargano, J. Korner, and U. Vaccaro, Sperner Capacities, *Graphs Combin.* **9** (1993), 31–46.

[21] A. P. Godbole, D. E. Skipper, and R. A. Sunley. t-Covering arrays: upper bounds and Poisson approximations, *Combinatorics, Probability and Computing* **5** (1996), 105–118.

[22] A. Hedayat, N. Sloane, and J. Stufken, Orthogonal Arrays, *Springer-Verlag* New York, 1999.

[23] F.K. Hwang, A method for detecting all defetive members in a population by group testing, *J. Amer. Statist. Assoc.* **67** (1972), 605–608.

[24] G. Katona. Two applications (for search theory and truth functions) of Sperner type theorems. *Periodica Math.* **3** (1973), 19–26.

[25] D. Kleitman and J. Spencer, Families of k-independent sets, *Discrete Math.* **6** (1973), 255–262.

[26] D.R. Kuhn and M. Reilly, An investigation of the applicability of design of experiments to software testing, *In Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, Los Alamitos, CA, October 2002, IEEE, 91–95.

[27] D.R. Kuhn, D.R. Wallace and A.M. Gallo, Software fault interactions and implications for software testing, *IEEE Trans. Soft. Eng.* **30** (2004), 418–421.

[28] J. Lawrence, R. N. Kacker, Y. Lei, D. R. Kuhn, M. Forbes, A Survey of Binary Covering Arrays, *The Electronic Journal of Combinatorics* **18** (2011), 84.

[29] D. Lubell, A short proof of Sperner's lemma, *J. Comb. Theory* **1** (1966), 299.

[30] A.J. Macula and L.J. Popyack, A group testing method for finding patterns in data, *Discrete Appl. Math.* **144** (2004), 149–157.

[31] E. Maltais, Covering Arrays Avoiding Forbidden Edges And Edge Clique Covers, *MSc Thesis, University of Ottawa*, Ottawa, 2009.

[32] C. Martínez, L. Moura, D. Panario and B. Stevens, Locating errors using ELAs, covering arrays, and adaptive testing algorithms. *SIAM J. Discrete Math* **4** (2009), 1776–1799.

[33] K. Meagher, Covering Arrays on Graphs: Qualitative Independence Graphs and Extremal Set Partition Theory, *PhD Thesis, University of Ottawa*, Ottawa, 2005.

[34] L.D. Mešalkin, A generalization of Sperner's theorem on the number of subsets of a finite set, *Teor. Verojatnost i Primenen* **8** (1963), 219–220.

[35] L. Moura, J. Stardom, B. Stevens, and A. Williams, Covering arrays with mixed alphabet sizes, *J. Comb. Des.* **11** (2003), 413–432.

[36] J. Naor and M. Naor, Small-bias probability spaces: efficient constructions and applications. *SIAM J. Computing* **22** (1993), 838–856.

[37] M. Naor, L.J. Schulman, and A. Srinivasan, Splitters and near-optimal randomization, *Proc. 36th Annual Symp. Foundations of Computer Science (FOCS), IEE*, (1996), 182–191.

[38] K. H. Rosen, Discrete Mathematics and Its Applications, *McGraw-Hill*, New York, 2003.

[39] E. Sperner, Ein Satz über Untermengen einer endlichen Menge, *Math. A.* **27** (1928), 544–548.

[40] D.C. Torney, Sets pooling designs, *Ann. Comb.* **3** (1999), 95–101.

[41] A.W. Williams and R.L. Probert, A measure for component interaction test coverage, *Proc. ACS/IEEE Intl. Conf. Comput. Syst. & Applic.*, 2001, 301–311.

[42] K. Yamamoto, Logarithmic order of free distributive lattice. *J. Math. Soc. Japan* **6** (1954), 343–353.