Under consideration for publication in Formal Aspects of Computing

Information Gain of Black-box Testing[1]

Linmin Yang, Zhe Dang and Thomas R. Fischer
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164, USA
{lyang1, zdang, fischer}@eecs.wsu.edu

Abstract. For model-based black-box testing, test cases are often selected from the syntactic appearance of the specification of the system under test, according to a pre-given test data adequacy criterion. We introduce a novel approach that is semantics-based, independent of the syntactic appearance of the system specification. Basically, we model the system under test as a random variable, whose sample space consists of all possible behavior sets (with respect to the specification) over the known interface of the black-box. The entropy of the system is measured as the (Shannon) entropy of the random variable. In our criterion, the coverage of a test set is measured as the expected amount of entropy decrease (i.e., the expected amount of information gained) once the test set is run. Since our criterion is syntactic independent, we study the notion of information-optimal software testing where, within a given constraint, a test set is selected to gain the most information.

Keywords: black-box testing, entropy, finite automata

1. Introduction

Software testing (roughly speaking, evaluating software by running it) is still the most widely accepted approach for quality assurance of software systems with nontrivial complexity. A textbook testing procedure is shown in Fig. 1 [AmO08]. A test case is the input data fed to the system under test. When a test case is selected, the system can then be executed with the test case. In consequence, the tester decides whether the result of the execution is as expected or not (e.g., comparing the result with the system's specification). After a set of test cases are run, an error (the system does not meet its specification) is possibly identified. However, when there is no error found, one usually cannot conclude that the system does meet its specification. This is because, for a nontrivial system, there are potentially infinitely many test cases and a tester can only run finitely many of them. On the other hand, a test case is selected before the test case is run and an error can only be identified after a test case is run. This raises a great challenge in software testing: How should test cases be selected?

Test cases are typically generated according to a pre-given test data adequacy criterion [GoG75], which associates

[1] A 7-page abstract of this paper [YDF09] is in the conference proceedings of FCS'09.
Correspondence and offprint requests to: Linmin Yang, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, USA, e-mail: lyang1@eecs.wsu.edu

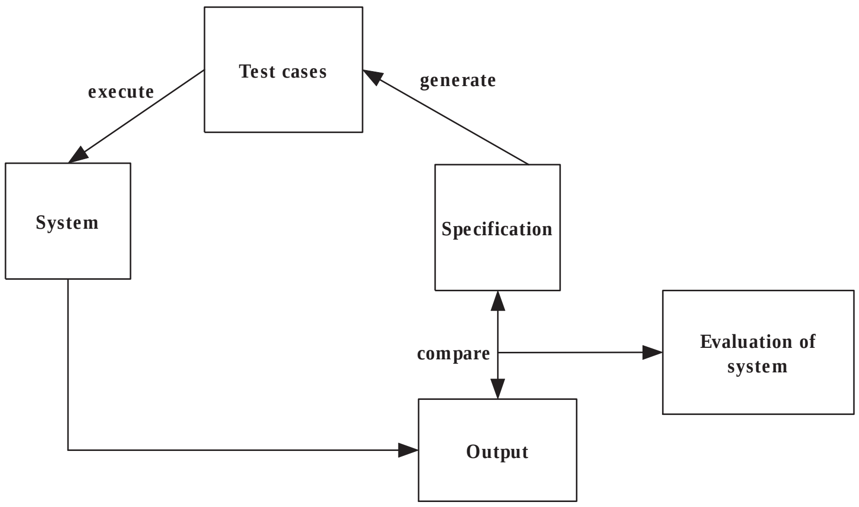

Fig. 1. A textbook testing procedure [AmO08].

a degree of adequacy with the test set (i.e., the set of generated test cases) to indicate coverage of the test set with respect to a system specification [ZHM97]. Formally, a test data adequacy criterion C is a function that maps a triple of a system under test, a specification and a test set to an (adequacy) degree in $[0, 1]$ [ZhH92]. In particular,

$$C(Sys, Spec, t) = r \tag{1}$$

indicates that, under the criterion C, the adequacy of the test set t on the system Sys with respect to the specification $Spec$ is of degree r. Naturally, as we mentioned earlier, a test data adequacy criterion, besides judging the test quality of an existing test set, is also a guideline of a test case generator [ZHM97]. In this paper, we treat the system Sys in (1) as a black-box [Bei95, MyS04, AmO08]. This is typical when the source code of the system is usually unavailable (e.g., commercial off-the-shelf (COTS) systems) or is too complex to analyze. In this case, even though the implementation details of the Sys are not clear, some attributes $Attr$ of it could be known; e.g., whether Sys is deterministic or nondeterministic, how many states the Sys has, what constitutes the input-output interface of the Sys, etc. Consequently, we can rewrite (1) into

$$C(Attr, Spec, t) = r. \tag{2}$$

Sometimes, we just omit the parameter $Attr$ when it is clearly given in the context. We shall emphasize that, since the system under test is a black-box, the degree r in (2) is independent of the specific system Sys under test and also independent of the results of executing the test set t.

Note that in (2), when the criterion C and the system attributes $Attr$ are given, the test set t is essentially generated from the system specification $Spec$ for each given r. Examples of formalisms used for the system specification are logical expressions (such as Boolean formulas and temporal logic formulas [Pnu77]) which describe the system's behaviors as a mathematical statement, C-like code (such as PROMELA [Hol97]), tables (such as SCR [Hei02]), and graphs (such as data flow graphs, control flow graphs and statecharts) which describe how a system is intended to operate. One might have already noticed that a system can be described using different formalisms. Even within one formalism, one can specify the same system in different ways such that the resulting specifications share the same semantics. For instance, suppose that Boolean formulas $A \vee B$ and $\overline{(\overline{A} \wedge \overline{B})}$ are specifications for a Boolean circuit. These two specifications have the same semantics, though their syntactic appearances are different.

Common testing criteria give the adequacy degree of a test set based on the syntactic appearance of the system specification. Hence, those criteria are *syntax-based*. This causes problems. For instance, a slight change to a specification's syntactic appearance, even if the specification still keeps its semantics, might result in a dramatically different adequacy degree for the same test set. Additionally, if a syntax-based test data adequacy criterion returns the same adequacy degree for two test sets, then the two test sets are indistinguishable with respect to the criterion. For instance, in the branch coverage criterion, each branch is born equal. This is not intuitively true. A similar problem exists for Clause Coverage [AOH03] for ground formulas (i.e., without quantifiers) in first order logic and testing criteria for temporal logic formulas [TSL04]; the criteria could not tell the difference between two test sets that achieve the same coverage.

An ideal test set will always identify an error whenever the system under test has an error. It is widely agreed that one direct measurement of the effectiveness of a test set is its fault-detecting ability [DuN84, WeJ91, ZHM97]. However, it is understood that there are simply no computable and ideal criteria to generate effective test sets [ZHM97]. In our opinion, fault-detecting in a black-box system is closely related to our knowledge about the system. This is particularly true considering the fact that faults are often not easy to find. We summarize our opinion into the following two intuitive statements:

- the more we test, the more we know about the system under test;
- the more we know about the system under test, the more likely faults can be found.

From these statements, it is desirable to have a way to measure the amount of information of the black-box system Sys (about which we only know its attributes $Attr$) we gain with respect to the specification $Spec$ once the tests t (selected according to (2) using a given adequacy r) are run. Therefore, the measure concerns the system's semantics instead of its syntax. This naturally leads us to use Shannon entropy [Sha48, CoT06] to measure the information gain and, because of its syntactic-independence, we can now cross-compare the information gains of two test sets t_1 and t_2 of the same black-box system; even though t_1 and t_2 are generated from different criteria C, different specifications $Spec$, and/or different degrees r. For instance, consider a component-based system which is a nondeterministic choice $C_1 \Box C_2$ over two components C_1 and C_2. C_1 is modeled using statecharts [Har87] in standardized modeling language UML, while C_2 is modeled using logical expressions such as LTL formulas [CGP99]. Suppose that we use the branch-coverage [AmO08] criterion and the property-coverage criterion [TSL04] as testing criteria for C_1 and C_2, respectively. We also have a test set t which consists of two subsets t_1 and t_2, which are test sets for components C_1 and C_2, respectively. It would be impossible to obtain a coverage that the test set t achieves on the whole system, even if we already have the branch coverage that t_1 achieves on C_1 and the predicate coverage that t_2 achieves on C_2. On the contrary, our information-theoretic approach can overcome this problem, since our approach is syntactic independent, and it does not care whether a system is modeled as a graph or a formula, as long as its semantics remains the same. As will be shown in the paper, the information gain is calculated *before* tests are run. Therefore, the information gain also serves as a syntax-independent coverage measure once no faults are found after a test set is run (which is often the case). Our paper is outlined as follows.

We model the system under test as a reactive labeled transition system Sys whose observable behaviors are sequences of input-output pairs. We also assume that the system under test is deterministic. In some software testing literature [VaP05], an observable behavior is called a *trace*. In this context, the objective of software testing is to test whether the observable behaviors of a black-box software system conform with a set of sequences of input-output pairs. The set is called a *trace-specification*, which specifies the observable behaviors that the system under test is intended to have. Let P be a set of sequences of input-output pairs, which is a trace-specification that the system under test is intended to conform with. This P is the whole or part of the system specification. Since in practice, we can only test finitely many test cases, here we assume that P is a finite (but could be huge) set. We use a tree T, called the *trace-specification tree*, to represent the trace-specification, where each edge is labeled with an input-output pair. Clearly, it is possible that not every path (from the root of T to some node in T) is an observable behavior of the Sys; it is testing that tells which path is and which path is not. When, through testing, a path is indeed an observable behavior of the Sys, we mark each edge on the path as "connected". When, however, the path is not an observable behavior of the Sys, the test result shows the longest prefix of the path such that the prefix is an observable behavior of the Sys. In this case, we mark every edge (if any) in the prefix as "connected" and the remaining edges on the path as "disconnected". Hence, an edge is marked connected (resp. disconnected) when the path from the root of T to the edge itself (included) is (resp. is not) an observable behavior. Notice that observable behaviors of the Sys are prefix-closed and hence, when an edge is marked disconnected, all its offspring edges are also disconnected. Therefore, running tests is a procedure of marking the edges of T.

Before any tests are performed, we do not know whether, for each edge in T, it is connected or disconnected. (It is the tests that tell which is the case for each edge.) After testing a sufficient number of test cases, every edge in T is marked either connected or disconnected. At this moment, the *system tree* is the maximal subtree of T such that every edge of the system tree is marked connected. The system tree represents all the observable behaviors of the Sys with respect to the trace-specification P. Hence, before any tests are run, there is uncertainty in what the system tree would be. Adopting the idea of entropy in information theory, we model the system tree (which we do not clearly know before the testing) as a random variable X_T, whose sample space is the set of all subtrees that share the same root with the tree T. The entropy , written $H(T)$, of the system is measured as the (Shannon) entropy of the random variable X_T. In order to calculate $H(T)$, we need probabilities of edges being connected or not. Those probabilities could be pre-assigned. However, usually probabilities of edges are simply unknown. In that case, we can calculate the probabilities of edges such that $H(T)$ reaches the maximum (i.e., we do not have any additional information).

After a set t of test cases is executed, we know a little more about the system tree from the execution results. In consequence, the entropy of the system decreases from $H(T)$ to $H(T|\text{after testing } t)$, the conditional entropy given the tests. That is, the information *gain* of running tests t is

$$G(t) = H(T) - H(T|\text{after testing } t).$$

This gain is calculated before the testing begins, and hence we can use the gain as a guideline to develop information-optimal testing strategies that achieve the most gain. In other words, we can pick the test set t that can achieve $\max G(t)$ subject to some constraint (e.g., the size of t is bounded by a certain number).

Notice that the aforementioned information gain $G(t)$ is syntactic independent; i.e., it is independent of the formalism that is used to describe the behaviors in P. This is because the entropy of a random variable keeps unchanged after a one-to-one function is applied [CoT06]. More intuitively, the amount of Shannon information in an object is the same no matter whether one describes it in English or in French.

Shannon entropy has been used in various areas in Computer Science. For instance, in data mining, a neural network-like classification network with hidden layers is constructed in analyzing a software system's input-output relation through a training set [LFK03]. The algorithm in constructing the network stops when further adding new layers would not make the entropy of the network significantly decrease. The resulting classification network is then used to help select "non-redundant" test cases. Note that our work is completely different from [LFK03], as, we study "information-optimal" testing processes, and our information-optimal test strategy is pre-computed *without* running any training set. Also, our work emphasizes the semantic dependency between test cases, whereas in many cases, researchers in testing treat test cases as a set (all members are born equal) [Gau95]. Reference [NVS04] studies optimal testing strategies for nondeterministic systems, while using a game theory approach.

The concept of Shannon entropy, in some historical literature [Lan61, Lad07], is closely related to the second law of thermodynamics in physics. This law requires that the process that brings down the entropy of a thermal system causes a positive heat flow from the system to its environment. Landauer's principle [Lan61] states a similar principle in the digital world. That is, in a computational device, the erasure of the Shannon information is accompanied by a generation of heat. While running the test cases, the Shannon entropy of the system under test is decreasing. From Landauer's principle, there should be a heat flow from the system under test to the environment during the software testing process. Hence, the software system is cooling down during a software testing process. From this point of view, intuitively, our information-optimal testing strategy cools down the system under test fastest.

The rest of the paper is organized as follows. We formally give our definitions and terminologies on languages, trees, transition systems, and finite automata in Section 2, together with an array selection algorithm that will be used throughout this paper. In Section 3, we formally define the entropy of a trace-specification represented as a tree and the information-optimality of a testing strategy, and develop algorithms to calculate information-optimal testing strategies of the system under test. In section 4, we study information-optimal testing strategies when the trace-specification is represented as a finite automaton. We summarize our study and briefly introduce our future work in Section 5.

2. Preparations

In this section, we provide definitions and terminologies on languages, finite automata, trees, (labeled) transition systems, and an algorithm, called `MAX-SELECT`, on selecting certain numbers from arrays, which will be used later in the paper.

2.1. Languages and Finite Automata

Let Σ be an alphabet. A language L is a set of words on the alphabet; i.e., $L \subseteq \Sigma^*$. For two words ω and ω', we use $\omega' \prec \omega$ to denote the fact that ω' is a (not necessarily proper) prefix of ω. L is *prefix-free* if, for any $\omega \in L$ and any $\omega' \prec \omega$, we have

$$\omega' \in L \text{ implies } \omega = \omega'.$$

L is *prefix-closed* if, for any ω and any $\omega' \prec \omega$, we have

$$\omega \in L \text{ implies } \omega' \in L.$$

Let $A = \langle S, s_{init}, F, \Sigma, R\rangle$ be a *deterministic finite automaton* (DFA), where S is the set of states with s_{init} being the initial state, F is the set of accepting states, $\Sigma = \{a_1, \cdots, a_k\}$ is the alphabet, and $R \subseteq S \times \Sigma \times S$ is the set of state transitions, satisfying that from a state and a symbol, one can at most reach one state (formally, $\forall s \in S, a \in \Sigma$, there is at most one s' with $(s, a, s') \in R$). A word $\omega = x_1 \cdots x_i$ in Σ^* is *accepted* by A if there is a sequence of states $s_0 s_1 \cdots s_i$, such that $s_0 = s_{init}$, $s_i \in F$, and $(s_{j-1}, x_j, s_j) \in R$ for $1 \le j \le i$. The language $L(A)$ that A accepts is the set of words accepted by A. Without loss of generality, we assume that A is cleaned up. That is, every state in A is reachable from the initial state, and can reach an accepting state.

When a language is finite, a tree can also be employed to represent it, as in below.

2.2. Trees

Let $L \subseteq \Sigma^*$ be a finite language and $\widehat{L}$ be its maximal prefix-free subset. Naturally, one can use a tree T to *represent* L. Every edge in T is labeled with a symbol in Σ, and any two distinct child edges (a child edge of a node N is the edge from N to a child node of N) of the same node cannot have the same label. Furthermore, T has $|\widehat{L}|$ leaves and for each leaf, the sequence of the labels on the path from the root to the leaf is a word in $\widehat{L}$. We use the following terminologies for the tree T:

- an *edge* from a node N to its child node N' is denoted by $\langle N, N'\rangle$ where N is the *source* of $\langle N, N'\rangle$ and N' is the *a-child* of N when the edge is labeled with $a \in \Sigma$;
- the *parent edge* of an edge $\langle N, N'\rangle$ (N is not the root) is the edge from the parent node of N to N itself;
- a *sibling edge* of edge e is an edge that shares the same source with e;
- t is a *subtree* of T if t is a tree, and every node and every edge in t is a node and an edge, respectively, in T;
- the *root path of node* N is the path (of edges) from the root of T all the way down to N itself. When we collect the labels of the edges on the path, a word in Σ^* is obtained. Sometimes, we simply use the word to (uniquely) identify the path;
- the *root path of subtree* t is the root path of the root of the subtree t in T;
- an *empty tree* is one that contains exactly one node.We use $\emptyset$ to denote an empty tree when the node in the tree is clear from the context;
- we use $t \prec T$ to denote that t is a subtree of T and shares the same root with T. Intuitively, when $t \prec T$, the t can be obtained by dropping some leaves repeatedly from T;
- a *subtree at node* N is a subtree of T with root N. We use the *maximal subtree at* N to denote the maximal such subtree at N;

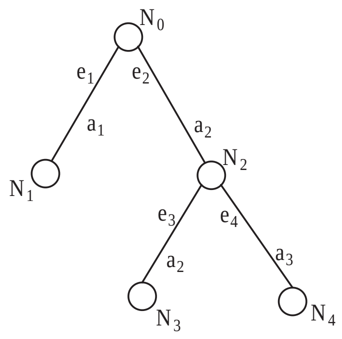

Fig. 2. A tree represents the language $L = \{a_1, a_2, a_2a_2, a_2a_3\}$ over the alphabet $\Sigma = \{a_1, a_2, a_3\}$.

- a *subtree under edge* $\langle N, N' \rangle$ is a subtree at node N';
- the *child-tree under edge* $\langle N, N' \rangle$ is the maximal subtree at node N';
- a *child-tree of node* N is the child-tree under edge $\langle N, N' \rangle$ for some N'. In this case, the child-tree is also called the a*-child-tree* of N, when N' is the a-child of N, for some $a \in \Sigma$;

Example 1. Fig. 2 gives a tree T representing $L = \{a_1, a_2, a_2a_2, a_2a_3\}$ over the alphabet $\Sigma = \{a_1, a_2, a_3\}$. The subtree with the set of edges $\{e_1, e_2\}$ is a subtree $t \prec T$. The a_2-child-tree of the root of T is the one with the set of edges $\{e_3, e_4\}$; the a_1-child-tree with the set of node(s) $\{N_1\}$ is simply an empty tree. □

2.3. Transition Systems

Let Sys be the software system under test. Here we modify the formal model in [XiD05]. A similar model can be found in a later paper [Tre08]. Sys is a transition system that changes from one state to another while executing a transition labeled by a symbol. Formally,

$$Sys = \langle S, s_{init}, \nabla, R \rangle$$

is a *(labeled) transition system*, where S is the set of states with $s_{init} \in S$ being the initial state, ∇ is a finite set of symbols, and R will be explained in a moment. More specifically, $\nabla = \Pi \cup \Gamma$, where Π and Γ are two disjoint subsets, and Π is the set of input symbols, while Γ is the set of output symbols. In particular, we call $\Pi \times \Gamma$ the set of *observable (input-output) pairs* (sometimes, we also call such pairs as observable events), and (Π, Γ) is the *interface* of Sys. $R \subseteq S \times (\Pi \times \Gamma) \times S$ is the set of state transitions. An *executable sequence* of Sys, $\omega = x_1 \cdots x_i$ for some i, is a word on alphabet $(\Pi \times \Gamma)$, such that there is a sequence of states, say, $s_0 \cdots s_i$, with $(s_{j-1}, x_j, s_j) \in R$ for $1 \leq j \leq i$, and s_0 is the initial state s_{init}. We call such a word ω an *(observable input-output) behavior* of the system. Note that a general form of executable sequence (i.e., a sequence of input, output and internal symbols) can be modeled as ours, if one has at most a fixed number of inputs, followed by an output. For instance, we can encode the sequence, say, $input_1, \cdots, input_i, output$, as one pair $\langle (input_1, \cdots, input_i), output \rangle$ in an expanded input alphabet. Our system here is actually a reactive system [HaP89], and the theoretical root of our model is Mealy machines [HMU07] and I/O automata [LyT89]. Note that the labeled transition system is universal since the number of states could be infinite. Also note that behaviors of Sys are prefix-closed. In this paper, Sys is a *black-box* system (i.e., we assume that we only know its interface).

In our context, software testing is to test whether the (black-box) software system Sys conforms with a trace-specification. No matter what formalism is used, the trace-specification, semantically, is commonly a set of sequences in $(\Pi \times \Gamma)^*$. This set is a language that specifies the (observable) behaviors that the system under test is intended to have. The trace-specification, denoted as $P_{original} \subseteq (\Pi \times \Gamma)^*$, could be an infinite language. For instance, when testing a TV remote control, theoretically, there are an infinite number of button combination sequences to test. However, in the real world, we can only test up to a given bound d on the length of the input sequence. The trace-specification, denoted as P, that in practice we plan to test against, is a "truncation" of the original trace-specification $P_{original}$. That is, $P = \{\omega : \exists \omega' \in P_{original},\ \omega \prec \omega' \text{ and } |\omega| \leq d\}$. In other words, P is the set of all prefixes (up to the given length d) of words in $P_{original}$. Hence, P is simply a prefix-closed finite language.

In practice, this $P_{original}$ (as well as P) can be the whole or part of the *system specification* that describes the expected behaviors of the system under test. Such a specification can be drawn from, for instance, the design documents and requirement documents of the system Sys under test. How to derive such a $P_{original}$ and P is out of the scope of this paper; the reader is referred to [BJK05] for, e.g., model-based software testing. In this paper, we simply assume that the $P_{original}$ and the P are given.

The transition system Sys defined earlier is in general *output-nondeterministic*. That is, it is possible that, for some observable behavior $\omega \in (\Pi \times \Gamma)^*$ and some input symbol $b \in \Pi$, there are more than one output symbol $c \in \Gamma$ such that $\omega(b, c)$ (i.e., the concatenation of the string ω and the symbol (b, c)) is also an observable behavior of the Sys. In other words, one can possibly observe more than one output from an input symbol. The source of output-nondeterminism comes from such things as a highly nondeterministic implementation (such as a concurrent program) of the Sys under test, or a partial specification of the interface. It is still an on-going research issue how to test an output-nondeterministic system Sys [NVS04, TrB03, PYH03].

On the other hand, *output-deterministic* systems constitute a most important and most common class of software systems in practice; this is particularly true when such a system is used in a safety-critical application in which nondeterministic outputs are intended to be avoided. Formally, the transition system Sys is *output-deterministic* if for each $\omega \in (\Pi \times \Gamma)^*$ and input symbol $b \in \Pi$, there is at most one output symbol $c \in \Gamma$ such that $\omega(b, c)$ is an observable behavior of Sys. Implicitly, we have an option of "crash" after applying an input, which makes our approach more general than other models. For instance, suppose that, for a traffic light system with sensors, when a car is approaching at midnight (so no other cars are around), it is desirable that the light turns yellow or turns red. In here, we use a_{yellow} and a_{red} to represent the input-output pairs $(approaching, yellow)$ and $(approaching, red)$, respectively. When the system is not assumed output-deterministic, a test shows that a_{yellow} is actually observable does not necessarily conclude that a_{red} is not actually observable. However, when we assume that the system is output-deterministic, this additional knowledge will conclude exactly one of the following three scenarios: when the car is approaching,

(i) the light turns yellow;

(ii) the light turns red;

(iii) neither (i) or (ii); e.g. the light system crashes.

Hence, the positive test result of, say, (i), immediately implies that outcomes like (ii) and (iii) are not possible.

In this paper, we focus on output-deterministic systems. Actually, if a test execution engine can be built for output-nondeterministic systems, we can see that testing output-nondeterministic systems is a special case of testing output-deterministic systems, which will be discussed later in this paper.

2.4. A Technical Algorithm MAX-SELECT

We now present an algorithm to solve the following selection problem, which will be used in several algorithms later in the paper.

Let k be a number. Suppose that we are given q arrays of numbers, $Y_1, \cdots, Y_q$, each of which has $k + 1$ entries. Each array Y_j is nondecreasing, i.e., $Y_j[\texttt{index}] \leq Y_j[\texttt{index} + 1]$, $0 \leq \texttt{index} \leq k - 1$, and the first entry $Y_j[0]$ is 0. Let $I \leq k$ be a number. We would like to select indices $\texttt{index}_1, \cdots, \texttt{index}_q$, satisfying $\sum_{1 \leq j \leq q} \texttt{index}_j = I$, of the arrays $Y_1, \cdots, Y_q$, respectively, such that the sum

$$\sum_{1 \leq j \leq q} Y_j[\texttt{index}_j] \tag{3}$$

is maximal. The instance of the problem is written as MAX-SELECT over $(\{Y_1, \cdots, Y_q\}, I)$. The result includes the desired indices and the sum in (3).

We use $SUM(\{Y_1, \cdots, Y_q\}, I)$ to denote the maximal sum reached in (3) for the problem instance. Suppose that the set $\mathcal{Y} = \{Y_1, \cdots, Y_q\}$ is partitioned into two nonempty subsets $\mathcal{Y}'$ and $\mathcal{Y}''$. One can show

$$SUM(\mathcal{Y}, I) = \max_{0 \le i \le I}(SUM(\mathcal{Y}', i) + SUM(\mathcal{Y}'', I - i)). \tag{4}$$

The algorithm MAX-SELECT that solves the problem is as follows. We first build, in linear time, a balanced binary tree t with q leaves, leaf$_1$, $\cdots$, leaf$_q$, and with roughly $2q$ nodes. Each leaf$_j$ corresponds to the array Y_j, $1 \le j \le q$. In the sequel, we simply use a set of arrays to denote a set of leaves. Let N be a node in t. We associate it with a table of $k+1$ entries. The i^{th} entry, $0 \le i \le k$, contains a number $\mathtt{SUM}_N[i]$ and a set $\mathtt{IND}_N[i]$ of pairs: each pair is a number $1 \le j \le q$ and an index to the array Y_j. Initially, $\mathtt{SUM}_N[i] = 0$ and $\mathtt{IND}_N[i] = \emptyset$, for all i. When N is a leaf Y_j, we further initialize $\mathtt{SUM}_N[i] = Y_j[i]$ and $\mathtt{IND}_N[i] = \{(j, i)\}$, for all $0 \le i \le k$. We now explain the meaning of the table. Let N be a nonleaf node in t. We use $\mathcal{Y}_N$ to denote the set of leaf nodes of which N is an ancestor. After the algorithm is run, $\mathtt{SUM}_N[i]$ is the value $SUM(\mathcal{Y}_N, i)$, and $\mathtt{IND}_N[i]$ records the desired indices for the MAX-SELECT instance over $(\mathcal{Y}_N, i)$. That is,

$$\mathtt{SUM}_N[i] = \sum_{(j, \mathtt{index}_j) \in \mathtt{IND}_N[i]} Y_j[\mathtt{index}_j].$$

Let N_1 and N_2 be the two child nodes of N (every nonleaf node in a balanced binary tree has exactly two children). From (4), $\mathtt{SUM}_N[i] = \max_{0 \le l \le i}(\mathtt{SUM}_{N_1}[l] + \mathtt{SUM}_{N_2}[i-l])$, which provides a way to calculate $\mathtt{SUM}_N[\cdot]$ and $\mathtt{IND}_N[\cdot]$ presented in the following algorithm:

```
MAX-SELECT($\{Y_1, \cdots, Y_q\}, I$):
//To find solutions to MAX-SELECT problem instances over $(\{Y_1, \cdots, Y_q\}, i)$,
//for all $i \le I$, and return the solution for $i = I$.
//Suppose that the balanced binary tree $t$ is already built with the arrays
//$Y_1, \cdots Y_q$ being the leaves. Each node $N$ is associated with $\mathtt{SUM}_N[\cdot]$ and $\mathtt{IND}_N[\cdot]$
//that are already initialized as described earlier in the subsection.
1. For $level := 1$ to (height of $t$)
   //a node of level (height of $t$) is the root
2.     For each nonleaf node $N$ of level $level$
       //suppose that $N_1$ and $N_2$ are the two child nodes of $N$
3.         For $i := 0$ to $I$
4.             $\mathtt{SUM}_N[i] := \max_{0 \le l \le I}(\mathtt{SUM}_{N_1}[l] + \mathtt{SUM}_{N_2}[i-l])$;
5.             Suppose that $1 \le l^* \le I$ reach the maximal in line 4;
6.             $\mathtt{IND}_N[i] := \mathtt{IND}_N[l^*] \cup \mathtt{IND}_N[i - l^*]$;
7. Return $\mathtt{SUM}_{root}[I]$ and $\mathtt{index}_1, \cdots, \mathtt{index}_q$.
   //$root$ is the root of $T$
   //$\mathtt{SUM}_{root}[I]$ is the value $SUM(\{Y_1, \cdots, Y_q\}, I)$
   //$\mathtt{IND}_{root}[I]$ is a set of pairs $(j, \mathtt{index}_j)$, for each $1 \le j \le q$
```

Clearly, the algorithm runs in worst-case time $O(k^2 q)$, recalling that $k+1$ is the size of each of the arrays $Y_1, \cdots, Y_q$.

3. Information Gain of Tests and Information-Optimal Testing Strategies

In this section, we assume that the system Sys under test is a black-box transition system with its interface known. Recall that Π and Γ are the input symbols and output symbols of Sys, respectively. Throughout this section, the system Sys under test is assumed to be output-deterministic. Before we develop algorithms for information-optimal testing of such systems, we need definitions on the test oracle.

As explained before, we use $P_{original} \subseteq (\Pi \times \Gamma)^*$ to denote a set of intended input-output behaviors of the Sys under test. The trace-specification $P \subseteq P_{original}$ is a finite set that we will actually test against, which is the set of all prefixes (up to length of a given number d) of sequences in $P_{original}$. In the sequel and without loss of generality, we simply assume that P is a finite set $\subseteq (\Pi \times \Gamma)^*$. We use a tree T, i.e., the trace-specification tree, to represent P, where every edge on T is labeled with a pair of input and output symbols. When we drop output symbols from every

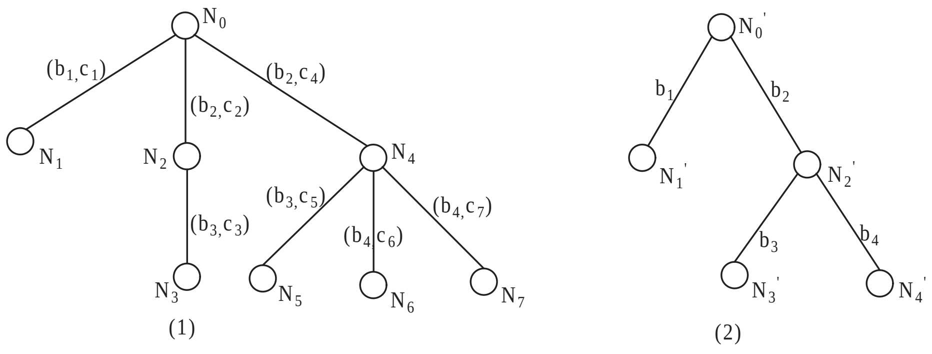

Fig. 3. (1) A trace-specification tree T. (2) The corresponding input testing tree T_Π of T.

sequence in P, we obtain a set $P_\Pi \subseteq \Pi^*$ of input symbol sequences; i.e., $P_\Pi = \{b_1 \cdots b_n : (b_1, c_1) \cdots (b_n, c_n) \in P,$ for some n and each $c_i \in \Gamma\}$. The tree T_Π that represents P_Π is called the *input testing tree*.

Example 2. In Fig. 3 (1), we show the trace-specification tree T representing the trace-specification $P = \{(b_1,c_1),(b_2,c_2),\ (b_2,c_4),\ (b_2,c_2)(b_3,c_3),\ (b_2,c_4)(b_3,c_5),\ (b_2,c_4)(b_4,c_6),\ (b_2,c_4)(b_4,c_7)\}$. The trace-specification P specifies that the system under test (which is output-deterministic) is expected to have the following behaviors:

- Initially, when button b_1 is pressed, color c_1 is shown.
- Initially, when button b_2 is pressed, either color c_2 or color c_4 is shown.
  - when color c_2 is shown, further pressing button b_3 will show color c_3.
  - when color c_4 is shown, we can further press button b_3 or b_4.
    - when pressing button b_3, color c_5 is shown.
    - when pressing button b_4, either color c_6 or c_7 is shown.

From P, we can obtain the set $P_\Pi = \{b_1, b_2, b_2b_3, b_2b_4\}$, and the corresponding input testing tree is shown in Fig. 3 (2). □

We further assume that Sys is *sequentially testable*; that is, there is a test execution engine *oracle* such that, if we send an input sequence $\pi = b_1 \cdots b_n$ on Π to the oracle, it will "read" symbols in π, one by one, from b_1 up to b_n while running the black-box Sys (this is a common assumption for black-box testing [PVY01]). As a result, the oracle returns an output symbol right after each input symbol read. The run stops when the Sys crashes on some input, or, the last symbol in the sequence is read and the corresponding output symbol is returned. Formally, a *test case* π is a word in Π^*. The oracle is equipped with a deterministic program $Test$ that runs on Sys and π and always halts with an output $Test(Sys, \pi)$. The run is said testing π. The oracle, as usual, honestly relays the outputs from the Sys. That is, for all n and $b_1 \cdots b_n \in \Pi^*$,

- $Test(Sys, b_1 \cdots b_n) = c_1 \cdots c_n$ if $(b_1,c_1)\cdots(b_n,c_n)$ is an observable behavior of Sys;
- $Test(Sys, b_1 \cdots b_n) = c_1 \cdots c_i \bot$, for some $i < n$, if $(b_1,c_1)\cdots(b_i,c_i)$ is an observable behavior of Sys, and $(b_1,c_1)\cdots(b_i,c_i)(b_{i+1},c)$ is not an observable behavior of Sys, for any $c \in \Gamma$. The symbol $\bot \notin \Gamma$ indicates "Sys crashes".

Notice that, for empty string ϵ, $Test(Sys, \epsilon) = \epsilon$ by definition. Since Sys is output-deterministic, $Test(Sys, \pi)$ is unique.

We say that the Sys *conforms with* P if, for every input symbol sequence $b_1 \cdots b_n \in P_\Pi$, for some n, we have $Test(Sys, b_1 \cdots b_n) = c_1 \cdots c_n \in \Gamma^*$ and $(b_1,c_1)\cdots(b_n,c_n) \in P$ (in some literature, this is called $\leq_{iot}$*-correct* with respect to P [VaP05]). That is, in terms of the trace-specification tree, for each path from the root in the input testing tree T_Π, suppose that the sequence of labels on the path is $b_1 \cdots b_n$, one can find a path in the trace-specification tree T such that the label sequence on the latter path is $(b_1, c_1) \cdots (b_n, c_n)$ for some $c_1, \cdots, c_n$ satisfying that $(b_1, c_1) \cdots (b_n, c_n)$ is an observable behavior of Sys.

Running a test case can be considered as a process of marking on the trace-specification tree T as follows. An edge in T is marked with "connected" if the sequence of the input-output labels on the path from the root to the edge (included) is an observable behavior of Sys; it is marked with "disconnected" if otherwise. Let the edge e be labeled with $(b, c) \in \Pi \times \Gamma$. One can observe that:

(1) once e is marked disconnected, every edge in the child-tree under edge e must be also marked disconnected (this is because the set of observable behaviors of Sys is prefix-closed);

(2) once e is marked connected, then every sibling edge e that is labeled with (b, c') for some $c' \in \Gamma$ must be marked disconnected (this is because Sys is output-deterministic), and hence edges in the child-trees under such sibling edges must also be marked disconnected.

Therefore, once e is marked, we implicitly assume that markings are already propagated further to its siblings (that share the same input symbol with e) and the offspring edges of itself and the siblings, using the above observation. Now, suppose that we run $Test(Sys, b_1 \cdots b_n)$ and obtain $c_1 \cdots c_n \in \Gamma^*$ as the result. Clearly by definition, the edges on the path (from the root of T) labeled with $(b_1, c_1) \cdots (b_n, c_n)$ are all marked connected. When $Test(Sys, b_1 \cdots b_n)$ results in $c_1 \cdots c_i \perp$ for some $i < n$, the edges on the path (from the root of T) labeled with $(b_1, c_1) \cdots (b_i, c_i)$ are all marked connected but all the edges in the child-tree under the last edge on the path are marked disconnected.

Initially, none of the edges in T is marked. As we test more and more test cases in P_Π, more and more edges in T are marked. Clearly, when all test cases in P_Π are tested, every edge in T is marked. Notice that, in this case, the *system tree* is the maximal subtree t of T such that $t \prec T$ and every edge in t is marked connected. The system tree exactly characterizes all the observable behaviors of Sys for all input sequences in P_Π. Consider a subtree t of T. We say that t is *output-deterministic* if, for any edge e with some label $(b, c) \in \Pi \times \Gamma$, e does not have a sibling edge with label (b, c') for any $c' \in \Gamma$. Observe that the system tree must be an output-deterministic subtree $t \prec T$.

3.1. Entropy of a Trace-specification Tree

Before any testing is performed, we do not know exactly what the system tree is except that it is an output-deterministic subtree $t \prec T$. We now treat the system tree as a random variable X_T and first study the algorithm in calculating its entropy.

Consider a path (labeled by) $(b_1, c_1) \cdots (b_n, c_n)$ in the trace-specification tree T, and the set E of the child edges of the last edge (b_n, c_n) on the path. Suppose that $e_1, \cdots, e_l$ for some l, are all the edges in E that are labeled by $(b, c^1), \cdots, (b, c^l)$, respectively, for some $b \in \Pi$ and some $c^1, \cdots c^l \in \Gamma$. We use E_b to denote $\{e_1, \cdots, e_l\}$. Let $p(e_i)$ be the probability that edge e_i is marked connected when all its ancestor edges are marked connected, and all other edges in E_b are marked disconnected. That is, $p(e_i)$ is the probability that $(b_1, c_1) \cdots (b_n, c_n)(b, c^i)$ is the only observable input-output behavior of the Sys (with $(b_1, c_1) \cdots (b_n, c_n)$ as prefix and with length $n+1$) given that $(b_1, c_1) \cdots (b_n, c_n)$ is an observable behavior of the Sys. Probabilities $p(e_i)$ could be pre-assigned (e.g., obtained from usage study [ALR09]). However, usually probabilities of edges are simply unknown. In that case, we can calculate the probabilities of edges such that $H(T)$ reaches the maximum (i.e., we do not have any additional information), which will be discussed later. Since Sys is output-deterministic, the $p(\cdot)$ must obviously satisfy the following additional constraint, for each b,

$$\sum_{e_i \in E_b} p(e_i) \leq 1.$$

We use $p(t, T)$ to denote the probability of t being the system tree (that shares the same root with T; i.e., $t \prec T$). Clearly, $p(t, T) = 0$ when t is not output-deterministic. Observe that

$$\sum_{\substack{t \prec T \text{ and} \\ t \text{ is output-deterministic}}} p(t, T) = 1$$

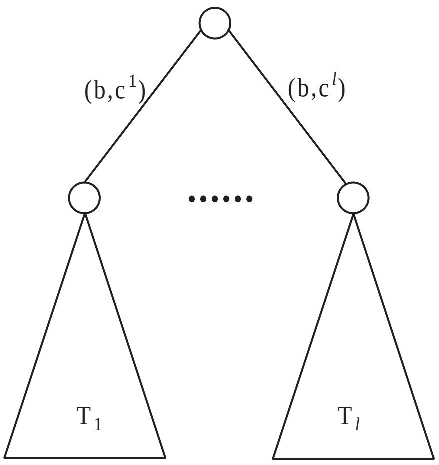

Fig. 4. A $(b,\cdot)$-component tree T_b of T.

and hence $p(\cdot, T)$ is a distribution. Now, the entropy of X_T, simply written $H(T)$, is

$$H(T) = -\sum_{\substack{t \prec T \text{ and} \\ t \text{ is output-deterministic}}} p(t,T)\log p(t,T).$$

Similarly, we can also define $H(t)$ for a subtree t (while keeping the probability assignments of the edges) of T as:

$$H(t) = -\sum_{\substack{t' \prec t \text{ and} \\ t' \text{ is output-deterministic}}} p(t',t)\log p(t',t). \tag{5}$$

Note that throughout this paper, the base of the logarithm is 2. In other words, we measure entropy in *bits*. By definition, $H(\emptyset) = 0$, since for the empty tree $\emptyset$, the system tree has only one choice that is the empty tree itself. Before we show how to calculate $H(T)$, some more definitions are needed. Let $b \in \Pi$ and E be the set of child edges of T's root. The $(b,\cdot)$-component tree T_b of T exactly consists of each edge e in E that is labeled with (b,c) for some c and the child-tree under the edge e. We use $C < T$ to denote that C is a $(b,\cdot)$-component tree of T for some $b \in \Pi$.

Example 3. For the trace-specification tree T shown in Fig. 3, the edge (b_1,c_1) forms the $(b_1,\cdot)$-component tree of T; edges (b_2,c_2) and (b_2,c_4), together with the child-trees under them, form the $(b_2.\cdot)$-component tree of T. □

One can show the following proposition,

Proposition 1.

$$H(T) = \sum_{C<T} H(C).$$

That is, the entropy of T is the summation of the entropy of each $(b,\cdot)$-component trees, $b \in \Pi$.

For a $(b,\cdot)$-component tree $C = T_b$, suppose that it consists of child trees $T_1,\cdots,T_l$ for some l, and edges $e_1,\cdots e_l$ with labels $(b,c^1),\cdots,(b,c^l)$ from the root of T to the root of $T_1,\cdots,T_l$, respectively (see Fig. 4). We use $p_1,\cdots,p_l$ to denote the probability assignments $p(e_1),\cdots,p(e_l)$, respectively. One can show,

Proposition 2. For the $(b,\cdot)$-component tree $C = T_b$ shown in Fig. 4,

$$H(C) = H(T_b) = \sum_{1\le i\le l} p_i H(T_i) - \sum_{1\le i\le l} p_i \log p_i - (1 - \sum_{1\le i\le l} p_i)\log(1 - \sum_{1\le i\le l} p_i).$$

The entropy $H(T_b)$, according to Proposition 2, is the "average" entropy of $H(T_i)$'s, together with the uncertainty introduced by output-determinism: among edges $(b,c^1),\cdots,(b,c^l)$, at most one of them is contained in the system tree since at most one of $c^1, \cdots, c^l$ can be the output from the input b. This additional entropy,

$$-\sum_{1\leq i\leq l} p_i \log p_i - (1 - \sum_{1\leq i\leq l} p_i)\log(1 - \sum_{1\leq i\leq l} p_i),$$

is exactly the entropy of a random variable with $(l+1)$ outcomes, and each outcome with probability assignments $p_1, \cdots, p_l, 1 - \sum_{1\leq i\leq l} p_i$, respectively. From the two propositions above, an algorithm that calculates the entropy of an output-deterministic trace-specification tree T is immediately given as follows, with time complexity $O(n)$, where n is the size of T (i.e., the number of edges in T).

```
ALG-entropy-tree($T$):
//To calculate the entropy $H(T)$ of the trace-specification tree $T$ with
//given probability assignments $p(\cdot)$ for an output-deterministic system.
//The return value of this algorithm is the entropy $H(T)$.
1. If $T = \emptyset$
2.     $H(T) := 0$;
3.     Return $H(T)$;
4. If $T$ only has one $(b,\cdot)$-component tree
       //$T$ now is in the form of a $(b,\cdot)$-component tree for some $b$,
       //shown in Fig. 4, consists of $l$ edges $(b,c^1),\cdots,(b,c^l)$ along with
       //child-trees $T_i$ under edges $(b,c^i)$ $(1 \leq i \leq l)$; each edge $(b,c^i)$ with
       //probability assignment $p_i$
5.     $H(T_i) :=$ ALG-entropy-tree($T_i$);
6.     $H(T) := \sum_{1\leq i\leq l} p_i H(T_i) - \sum_{1\leq i\leq l} p_i \log p_i - (1 - \sum_{1\leq i\leq l} p_i)\log(1 - \sum_{1\leq i\leq l} p_i)$;
7.     Return $H(T)$;
8. $H(T) := \sum_{C<T}$ ALG-entropy-tree($C$);
9. Return $H(T)$.
```

Before we proceed further, we need more notation. Given a trace-specification tree T and its corresponding input testing tree T_Π, for a path (labeled by) $\omega = (b_1, c_1)\cdots(b_i, c_i)$ in T, by definition, we have a path $\omega_\Pi = b_1 \cdots b_i$ in T_Π; in this case, we write $\omega \sim \omega_\Pi$. Similarly, for a node N in T, suppose that the path from the root to N is ω. Correspondingly, in T_Π, we have a path ω_Π from the root to some node N_Π, such that $\omega \sim \omega_\Pi$; in this case, we say $N \sim N_\Pi$. For a subtree t of T, we can find a minimal subtree t_Π of T_Π, such that for each node N in t, there is some node N_Π in t_Π such that $N \sim N_\Pi$; in this case, we say $t \sim t_\Pi$. For a subtree t_Π of T_Π, let t be a maximal subtree t of T such that $t \sim t_\Pi$; in this case, we say $t \simeq t_\Pi$ (note that there could be more than one such t, depending on the location of the root of t in T). Intuitively, a subtree t that satisfies $t \simeq t_\Pi$ is a maximal subtree of T, such that once every edge in t_Π is tested, every edge in t will be marked.

Example 4. For the trace-specification tree T and its corresponding input testing tree T_Π shown in Fig. 3, considering the subtree t of T that consists of nodes N_4, N_5, N_6 and N_7, and the subtree t' of T_Π that consists of nodes N'_2, N'_3 and N'_4, we have $t \simeq t'$. For the subtree t'' of T that consists of nodes N_2 and N_3, we also have $t'' \simeq t'$. □

Now, consider an edge e in T_Π. In the following, we will define the entropy "gain" $G(e)$ after the edge e is tested. To do this, consider all those subtrees t of T that satisfy $t \simeq e$ (here e is treated as the subtree of T_Π that only has the edge e). Recall that the t's are those t's that are marked after e is tested.

Example 5. Consider the trace-specification tree T and its corresponding input testing tree T_Π shown in Fig. 3. For the edge $e = \langle N'_2, N'_3\rangle$ in T_Π, we have $t_1 \simeq e$ and $t_2 \simeq e$, where t_1 consists of nodes N_2 and N_3, and t_2 consists of nodes N_4 and N_5 in T. Similarly, for the edge $e' = \langle N'_2, N'_4\rangle$, the only subtree t' of T such that $t' \simeq e'$ is the one consisting of nodes N_4, N_6 and N_7. □

For a subtree t of T with $t \simeq e$, we use $\omega_t = e^1 \cdots e^j$, for some j, to denote the root path of t in the trace-specification tree T. Also, we define $p(\omega_t) = p(e^1)\cdots p(e^j)$ when ω_t is not empty, and when ω_t is an empty path, $p(\omega_t) = 1$. Since $t \simeq e$, it is clear that t must be in the form of a subtree only containing a number edges, say, $e_1, \cdots, e_l$, for some l, that share the same source (the root of t), shown in Fig. 5. By definition in (5), we have

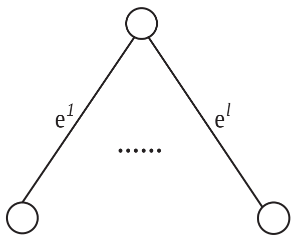

Fig. 5. A subtree t of T such that $t \simeq e$ for some edge e in T_Π.

$H(t) = -(\sum_{1\le i\le l} p_i \log p_i + (1 - \sum_{1\le i\le l} p_i)\log(1 - \sum_{1\le i\le l} p_i))$. Now, we define the entropy gain of the edge e in T_Π as

$$G(e) = \sum_{t\simeq e} p(\omega_t)H(t). \tag{6}$$

One can show, using Propositions 1 and 2, that the entropy $H(T)$ of the trace-specification tree T can be expressed as the summation of the entropy gains of the edges in the input testing tree T_Π:

Proposition 3.

$$H(T) = \sum_{e \text{ in } T_\Pi} G(e).$$

That is, once all the edges in the input testing tree is tested, the total gain is exactly the uncertainty $H(T)$ of the trace-specification tree; i.e., no uncertainty is left.

Example 6. Consider the trace-specification tree T and its corresponding input testing tree T_Π shown in Fig. 3. Suppose that the probability assignments of edges in T are as follows: $p(\langle N_0, N_1\rangle) = p(\langle N_2, N_3\rangle) = p(\langle N_4, N_5\rangle) = 1/2$, $p(\langle N_0, N_2\rangle) = 2/9$, $p(\langle N_0, N_4\rangle) = 2/3$, and $p(\langle N_4, N_6\rangle) = p(\langle N_4, N_7\rangle) = 1/3$. We have $H(T) = \log 18$ bits by applying the algorithm `ALG-entropy-tree(`T`)`. We can also calculate $H(T)$ using Proposition 3. For the edge $e = \langle N'_2, N'_3\rangle$ in T_Π, we have $t_1 \simeq e$ and $t_2 \simeq e$, where t_1 consists of nodes N_2 and N_3, and t_2 consists of nodes N_4 and N_5 in T. By definition, $H(t_1) = H(t_2) = 1$. By (6), we have $G(e) = p(\langle N_0, N_2\rangle)H(t_1) + p(\langle N_0, N_4\rangle)H(t_2) = 2/9 + 2/3 = 8/9$. Similarly, for all other edges in T_Π, we have $G(\langle N'_0, N'_1\rangle) = 1$, $G(\langle N'_0, N'_2\rangle) = 4/3\log 3 - 8/9$, and $G(\langle N'_2, N'_4\rangle) = 2/3\log 3$. From Proposition 3, we have $H(T) = 1 + 2\log 3 = \log 18$, which coincides with the results obtained from `ALG-entropy-tree(`T`)`. □

3.2. Testing Strategies and Gain

By running test cases in P_Π, one can check whether a software system Sys conforms with the given trace-specification P, which is a set of intended observable behaviors that Sys is supposed to have. Let T be the trace-specification tree that represents P. Recall that, running test cases using the oracle resembles the process of marking some edges in T. Since the gain of information on the initially unknown system tree of T after running a number of test cases in P_Π corresponds to the reduction of entropy of the testing tree by marking edges in T, a strategy that specifies the ordering of marking edges also represents the process of gaining the information while running tests.

A testing strategy $\mathcal{C}$ is a sequence of edges in the input testing tree T_Π, in the form of

$$e_{\mathcal{C}(1)}, \cdots, e_{\mathcal{C}(g)},$$

for some $g \le m$ (m is the number of edges in T_Π), satisfying the constraint that parent edges should precede their child edges in $\mathcal{C}$. That is, for any $1 \le i \le g$, if e is the parent edge of $e_{\mathcal{C}(i)}$, then there is a j with $1 \le j < i \le g$ such that $e = e_{\mathcal{C}(j)}$. Naturally, the strategy gives the ordering that test cases (i.e., input symbol sequences) should be run.

Let α be a prefix of $\mathcal{C}$, which corresponds to a subtree t_α (i.e., t_α exactly contains all the edges in α) of the input testing tree T_Π with $t_\alpha \prec T_\Pi$. Let $T_\alpha \prec T$ be the maximal subtree of the trace-specification tree T such that, for each

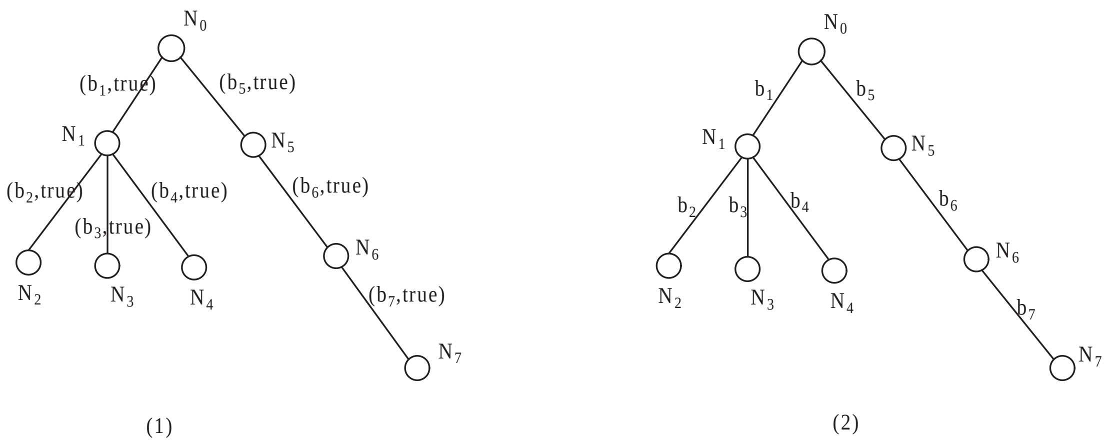

Fig. 6. (1) A trace-specification tree T. (2) The corresponding input testing tree T_Π of T.

path $(b_1, c_1) \cdots (b_i, c_i)$, for some i, in T_α, we have that $b_1 \cdots b_i$ is a path in the aforementioned subtree t_α of T_Π that represents α; i.e., $T_\alpha \simeq t_\alpha$. That is, after α is tested (i.e., every test case represented in t_α is tested), the uncertainty in T_α is gone completely, since, in this case, every edge in T_α is marked already using the test results. We use $G(\alpha)$ to denote the expected entropy reduction of T after α is tested, which is also the information gained on the system tree of T. The gain $G(\alpha)$ is defined as $H(T_\alpha)$. From Proposition 3 (taking T_α as T and t_α as T_Π), we have

$$G(\alpha) = \sum_{e \text{ in } t_\alpha} G(e), \tag{7}$$

and naturally, $G(\alpha) = H(T)$ when α is the entire strategy $\mathcal{C}$ when $g = m$ (that is, $\mathcal{C}$ covers all the edges in T_Π). When all the edges in T_Π are tested, the entropy gain is $H(T)$; i.e., no uncertainty is left. Let $Pre_{\mathcal{C}}(k)$ be the prefix of $\mathcal{C}$ of length k. One can easily show that $G(Pre_{\mathcal{C}}(k_1)) \le G(Pre_{\mathcal{C}}(k_2))$, for any $k_1 \le k_2$. That is, as we test more edges in the input testing tree, more information is gained. Sometimes, we abuse the notation G and use

$$G(t_\alpha) = G(\alpha) \tag{8}$$

to denote the gain of the subtree t_α of T_Π. In fact, (8) already gives the information gain of a test set $tests$ as follows. Recall that the set can be represented as a subtree t of T_Π; the gain of $tests$ is defined as

$$G(tests) = G(t). \tag{9}$$

In later sections, we will show how to select test cases that achieve the maximal amount of information gain under certain constraints.

3.3. Information-Optimal Testing Strategies with Pre-given Probability Assignments

In this subsection, unless stated otherwise, we assume that a testing strategy $\mathcal{C}$ covers all the edges in T_Π. An *information-optimal* testing strategy, as explained before, tries to make a maximal reduction of entropy in the testing tree. Formally,

Definition 1. Let T be a trace-specification tree and T_Π be the corresponding input testing tree with m edges. $\mathcal{C}^*$ is a `global information-optimal testing strategy` if, for any testing strategy $\mathcal{C}$, $G(Pre_{\mathcal{C}^*}(k)) \ge G(Pre_{\mathcal{C}}(k))$, for each $0 \le k \le m$.
That is, for any given length of prefix, a global information-optimal testing strategy reduces more uncertainties than any other strategy. Note that a global information-optimal testing strategy may not necessarily exist.

Example 7. Consider the trace-specification tree T in Fig. 6 (1) with $\Pi = \{b_1, \cdots, b_7\}$ and $\Gamma = \{true\}$. Its input testing tree is shown in Fig. 6 (2). Let e_i be the edge in T labeled with $(b_i, true)$, and e'_i be the edge in T_Π labeled with b_i, $1 \leq i \leq 7$. The probability assignments of edges are as follows: $p(e_1) = 8/9$, $p(e_2) = p(e_3) = p(e_4) = p(e_7) = 1/2$, $p(e_5) = 3/4$ and $p(e_6) = 2/3$. The input testing tree T_Π does not have a global information-optimal testing strategy, since one can easily check that, assuming that $\mathcal{C}^*$ were a global information-optimal testing strategy, $Pre_{\mathcal{C}^*}(1) = e'_5$, while $Pre_{\mathcal{C}^*}(3) = e'_1, e'_2, e'_3$, which leads to a contradiction. □

Now, we define a weaker form of information-optimality, when we are only allowed to test for k edges, and we try to maximize the entropy reduction after testing the k test cases.

Definition 2. Let T be a trace-specification tree, T_Π be the corresponding input testing tree with m edges and k be a number $\leq m$. $\mathcal{C}^*$ is a k-`information-optimal testing strategy` if, for any testing strategy $\mathcal{C}$, $G(Pre_{\mathcal{C}^*}(k)) \geq G(Pre_{\mathcal{C}}(k))$.

Note that the testing ordering within the first k edges would not matter (of course, parents should be tested before their children) when the k edges are given, since the gain is always the entropy of the subtree consisting of those k edges. Hence, finding a k-information-optimal testing strategy is equivalent to picking the first k edges to test in T_Π. Those k edges form a *k-information-optimal* subtree of T_Π, which is a subtree sharing the same root with T_Π that has the maximal gain among all the subtrees (sharing the same root with T_Π) which have k edges. Clearly, enumerating all possible subtrees $t \prec T_\Pi$ with k edges and picking the one with the maximal gain will result in an exponential time algorithm in k. In below, we will present an efficient algorithm that finds a k-information-optimal testing strategy $\mathcal{C}^*$ for a given number k.

We associate each node N in T_Π with a table of $k+1$ entries. The i^{th} entry, $0 \leq i \leq k$, contains a number $G_N[i]$ and a set $OPT_N[i]$ of edges, where $G_N[i]$ is the gain of the i-information-optimal subtree at node N, and $OPT_N[i]$ records the set of edges in that i-information-optimal subtree at node N. Suppose that N has q child nodes $N'_1, \cdots, N'_q$, for some $q \geq 1$. $G_N(i)$ can be calculated from $G_{N'_1}[\texttt{index}_1], \cdots, G_{N'_q}[\texttt{index}_q]$, for some $\texttt{index}_1, \cdots, \texttt{index}_q < i$. We define another array $Y_j[\cdot]$ for each N'_j. Each $Y_j[i+1]$ records the gain of the $(i+1)$-information-optimal subtree of the component tree $T_{N'_j}$, which consists of the edge $\langle N, N'_j \rangle$ together with the child-tree under the edge. Clearly, the $(i+1)$-information-optimal subtree of $T_{N'_j}$ exactly contains the edge $\langle N, N'_j \rangle$, together with the i-information-optimal subtree at node N'_j. From (7), we have $Y_j[i+1] = G(e) + G_{N'_j}[i]$, where $e = \langle N, N'_j \rangle$. One can observe that the i-information-optimal subtree at node N must be, for some $\texttt{index}_1, \cdots, \texttt{index}_q$, the union of $\texttt{index}_j$-information-optimal subtrees of $T_{N'_j}$, for all $1 \leq j \leq q$. Therefore, we have $G_N[i] = \sum_{1 \leq j \leq q} Y_j[\texttt{index}_j]$. The selection of $\texttt{index}_1, \cdots, \texttt{index}_q$ is left to the algorithm `MAX-SELECT`. The algorithm `ALG-opt` (T, T_Π, k) that calculates the k-information-optimal subtree of T_Π is given as follows, where $G_N[\cdot]$ and $OPT_N[\cdot]$ are global variables. If N is a leaf node, then we initialize $G_N[i] = 0$ and $OPT_N[i] = \emptyset$ for each $0 \leq i \leq k$.

```
ALG-opt ($T, T_\Pi, k$) :
//To find a $k$-information-optimal subtree of a given input testing tree $T_\Pi$ with
//pre-given probability assignment $p(e)$ for every edge $e$ in the
//trace-specification tree $T$.
//The return value has two parts: the entropy $H_{root}[k]$ and the set of
//edges $OPT_{root}[k]$ of the $k$-information-optimal subtree, where $root$ is the root of $T_\Pi$.
//$G_N[\cdot]$ and $OPT_N[\cdot]$ in below are global variables, initialized as in above.
1. If $k = 0$
2.     For each node $N$ in $T_\Pi$
3.         $G_N[k] := 0$ and $OPT_N[k] := \emptyset$;
4.     Return;
5. Run ALG-opt-tree($T, T_\Pi, k-1$);
6. For $level := 1$ to (height of $T_\Pi$)
   //a node of level (height of $T_\Pi$) is the root
7.     For each nonleaf node $N$ of level $level$
       //suppose that $N'_1, \cdots, N'_q$, for some $q \geq 1$, are all the child nodes of $N$
8.         For each $1 \leq j \leq q$
9.             $Y_j[0] := 0$;
10.            $G(e) := \sum_{t \simeq e} p(\omega_t)(-(\sum_{1 \leq i \leq l} p_i \log p_i + (1 - \sum_{1 \leq i \leq l} p_i) \log(1 - \sum_{1 \leq i \leq l} p_i)))$;
               //$e = \langle N, N'_j \rangle$; in the RHS of line 10, $e$ is treated as the
               //subtree of $T_\Pi$ that only has the edge $e$; $\omega_t = a_1 \cdots a_i$ is the
```

```
            //root path of $t$, $p(\omega_t) = p(a_1)\cdots p(a_i)$; $p(\omega_t) = 1$ if $\omega_t = \emptyset$; suppose
            //that $t$ exactly contains $l$ edges, say $e_1, \cdots, e_l$ that share the
            //same source, and each edge $e_l$ has probability assignment $p_i$
11.         For $0 \leq i \leq k-1$
12.             $Y_j[i+1] := G(e) + G_{N'_j}(i)$;
                //$Y_j[i+1]$ stores the entropy of the $(i+1)$-information-optimal
                //subtree of the component tree $T_{N'_j}$ mentioned earlier
13.     Run MAX-SELECT($\{Y_1, \cdots, Y_q\}, k$);
        //MAX-SELECT($\{Y_1, \cdots, Y_q\}, k$) returns a sequence $\mathtt{index}_1, \cdots, \mathtt{index}_q$
14.     $G_N[k] := \sum_{j:\mathtt{index}_j \neq 0} Y_j[\mathtt{index}_j]$;
15.     $OPT_N[k] := \bigcup_{j:\mathtt{index}_j \neq 0} OPT_{N'_j}[\mathtt{index}_j - 1] \cup \{\langle N, N'_j \rangle\}$;
16. Return $G_{root}[k]$ and $OPT_{root}[k]$.
```

In line 13, the (worst-case) time of running the algorithm MAX-SELECT($\{Y_1, \cdots, Y_q\}, k$) is $O(k^2 q)$, as pointed out in Section 2.4, MAX-SELECT is called every time when calculating an entry of $H_N[\cdot]$ and an entry $OPT_N[\cdot]$ of node N. Since there are $k+1$ entries for N, the time of finishing calculating all the entries of N is $O(k^3 q)$. Notice that q is the *branching factor* of N (see line 7), that is the number of child nodes of N, and the summation of all such branching factors for all nodes is m, which is the size of T. Hence the (worst-case) time complexity of ALG-opt (T, k) is $O(mk^3)$, given that in line 10, all the t's satisfying $t \simeq e$ are preprocessed and already stored in a data structure at e. The preprocessing can be done in time $O(n+m)$, where n is the size of T. Therefore, the (worst-case) time complexity of ALG-opt (T, T_Π, k) is $O(n + mk^3)$.

As mentioned earlier, a global information-optimal testing strategy might not exist for certain trace-specification trees. We can use the algorithm ALG-opt (T, T_Π, k) for computing k-information-optimal testing strategies to determine the existence of a global information-optimal testing strategy by setting k all the way from 1 to m. However, this is not practically efficient when m is large. It is interesting to see whether there are efficient algorithms in deciding the existence of a global information-optimal testing strategy by looking at the tree's structure.

We now consider the following *greedy information-optimal* testing strategy, which aims to reduce the entropy most at each step.

Definition 3. Let T be a trace-specification tree and T_Π be the corresponding input testing tree with m edges. $\mathcal{C}^*$ is a greedy information-optimal testing strategy if, for any testing strategy $\mathcal{C}$ with $k-1$, for some k, being the length of the longest common prefix between $\mathcal{C}^*$ and $\mathcal{C}$, $G(Pre_{\mathcal{C}^*}(k)) \geq G(Pre_{\mathcal{C}}(k))$.

From the definition, we can have the following observation. Consider two testing strategies $\mathcal{C}$ and $\mathcal{C}'$ of T_Π with the longest common prefix between them of length $k-1$. Suppose that $e_{\mathcal{C}(k)}$ is e and $e_{\mathcal{C}'(k)}$ is e'. We have

$$G(Pre_{\mathcal{C}}(k)) \geq G(Pre_{\mathcal{C}'}(k)), \text{ if and only if, } G(e) \geq G(e'). \tag{10}$$

To construct the greedy information-optimal testing strategy, we first introduce the concept of *available set*. Let E be a set of edges in T_Π. We define the *available set* of E, written AS(E), to be the set of edges e in T_Π such that e is either a child edge or a sibling edge of some edge in E. Intuitively, if edges in E form a subtree $t \prec T_\Pi$, when adding one or more edges in AS(E) to t, we can obtain a new subtree t', such that $t' \prec T_\Pi$ and $t \prec t'$. By definition, $e_{\mathcal{C}(k)}$ must be selected from AS($\{e_{\mathcal{C}(k)}, \cdots, e_{\mathcal{C}(k-1)}\}$). By (10), at each step we pick the edge (from the available set) that can achieve the maximal gain. The gain of an edge e, $G(e)$, is calculated in the same way as in ALG-opt (T, T_Π, k).

```
ALG-greedy-opt ($T, T_\Pi$):
//To find a greedy information-optimal testing strategy of the input testing tree $T_\Pi$,
//with pre-given probability assignment $p(e)$ for each edge $e$
//in the trace-specification tree $T$.
//The return value is a greedy information-optimal testing strategy $\mathcal{C}^*$ of $T_\Pi$.
1. AS[1] := $\{e : e$ is an edge originating from $T_\Pi$'s root$\}$.
2. For each $1 \leq i \leq m$
3.      For each $e \in$ AS[$i$]
4.          $G(e) := \sum_{t \simeq e} p(\omega_t)(-(\sum_{1 \leq i \leq l} p_i \log p_i + (1 - \sum_{1 \leq i \leq l} p_i) \log(1 - \sum_{1 \leq i \leq l} p_i)))$;
            //in the RHS of line 4, $e$ is treated as a subtree of $T_\Pi$ that only
            //has the edge $e$; $\omega_t = a_1 \cdots a_i$ is the root path of subtree $t$ in $T$,
            //$p(\omega_t) = p(a_1) \cdots p(a_i)$; $p(\omega_t) = 1$ if $\omega_t = \emptyset$; suppose that $t$ exactly
```

```
        //contains $l$ edges, say $e_1,\cdots,e_l$ that share the same source,
        //and each edge $e_l$ has probability assignment $p_i$ in $T$
5.      Suppose that $e^* \in \mathtt{AS}[i]$ achieves $\max_{e\in \mathtt{AS}[i]} G(e)$;
6.      $e_{\mathcal{C}(i)} := e^*$;
7.      $\mathtt{AS}[i+1] := (\mathtt{AS}[i] - \{e_{\mathcal{C}(i)}\}) \cup \mathtt{CHILDREN}(e_{\mathcal{C}(i)})$;
        //$\mathtt{AS}[i+1]$ now records $\mathtt{AS}(\{e_{\mathcal{C}(1)},\cdots,e_{\mathcal{C}(i)}\})$
8. Return $\mathcal{C}^* = e_{\mathcal{C}(1)},\cdots,e_{\mathcal{C}(i)},\cdots,e_{\mathcal{C}(m)}$.
```

The (worst-case) time complexity of `ALG-greedy-opt` (T, T_Π) is $O(n+m^2)$, with n being the size of T and m being the size of T_Π.

By definition, a greedy information-optimal testing strategy always selects the edge that can reduce the entropy most at each step. Of course, this greedy strategy does not necessarily lead to the information-optimal testing strategies as given in Definition 1 and 2. For example, for the trace-specification tree and the input testing tree in Fig. 6 with probability assignments given in Example 7., the prefix of length 3 of a 3-information-optimal testing strategy is e'_1, e'_2, e'_3, while in a greedy information-optimal testing strategy, the prefix of length 3 is e'_5, e'_6, e'_1. However, the greedy information-optimal testing strategy fits the situation that the testing procedure may be stopped at any time, and hence each time, we only aim to maximally reduce the entropy for each individual step.

3.4. Entropy and Information-Optimal Test Strategies of a Trace-specification Tree with Probability Assignments in the Worst Case

The aforementioned algorithms for information-optimal testing strategies rely on the probability assignments of edges, $p(\cdot)$, in the trace-specification tree T. When the assignment $p(\cdot)$ is explicitly given, we write $H(T, p(\cdot))$ to denote the entropy of T under $p(\cdot)$, which is calculated using algorithm `ALG-entropy-tree` (T). However, in practice, we usually do not know what the $p(\cdot)$ is. That is, we do not know the probability of whether an edge will be connected or disconnected with respect to the system under test. We now consider the worst case that we know the least amount of information on the system (i.e., the system under test is a truly blackbox). We will give the result in the worst case, where the uncertainty $H(T, p(\cdot))$ achieves the maximum for some $p(\cdot)$. Unless stated otherwise, from now on in this section, we use $H(T)$ to denote the maximal entropy of the testing tree T in the worst case; i,e., $H(T) = \sup_{p(\cdot)}\{H(T, p(\cdot))\}$. We use $p^*(\cdot)$ to denote the worst-case probability assignments on which the maximum is achieved.

Because of Proposition 1, it suffices for us to consider the case when T is a $(b,\cdot)$-component tree shown in Fig. 4. In this case,

$$H(T) = \sum_{1\le i\le l} p_i H(T_i) - \sum_{1\le i\le l} p_i \log p_i - (1 - \sum_{1\le i\le l} p_i)\log(1 - \sum_{1\le i\le l} p_i). \tag{11}$$

Suppose that, for the child-trees $T_1,\cdots,T_l$, we have already obtained the worst-case probability assignments and each $H(T_i)$ in (11) is already the worst-case entropy. We now calculate the $p_i = p_i^*, i = 1,\cdots,l$, in (11) that make the RHS maximal. Notice that $H(T)$ in (11) is a concave function over $p_1,\cdots,p_l$, since the second-order partial derivatives

$$\frac{\partial^2 H(T)}{\partial p_i \partial p_j} < 0, \text{ for all } 1 \le i,j \le l.$$

Therefore, let partial derivatives

$$\frac{\partial H(T)}{\partial p_i} = 0, \ i = 1, \cdots l.$$

We have,

$$H(T_i) - \log p_i^* + \log(1 - \sum_{1\le j\le l} p_j^*) = 0, \ i = 1,\cdots l.$$

Solving the above equations for the $p_1^*, \cdots, p_l^*$, we obtain the solutions

$$p_i^* = \frac{2^{H(T_i)}}{1 + \sum\limits_{1 \le j \le l} 2^{H(T_j)}},\ i = 1, \cdots l. \tag{12}$$

This already gives an algorithm to calculate the $p^*(\cdot)$ as follows.

```
ALG-$p^*$($T$):
//To calculate the probability assignments $p^*(\cdot)$ of edges
//in the trace-specification tree $T$ in the worst case.
//The return value is the probability assignments $p^*(\cdot)$.
1. For each leaf node $N$ in $T$
2.     $H(t_N) := 0$;
       //$t_N$ is the child-tree under the edge $e = \langle N', N \rangle$ for some $N'$
3. For $level := 1$ to (height of $T$)
   //a node of level (height of $T$) is the root
4.     For each nonleaf node $N$ of level $level$
       //suppose that the component trees of $N$ are
       //$(b_1, \cdot)$-component tree, $\cdots$, $(b_q, \cdot)$-component tree for some $q$
5.         For $1 \le h \le q$
               //let $t_h$ be the $(b_h, \cdot)$-component tree that consists of edges
               //$e_1, \cdots, e_l$ labeled with $(b_h, c^1), \cdots (b_h, c^l)$ respectively, together
               //with the child-trees $T_i$'s under the edges, and the entropy
               //$H(T_i)$ is already calculated in the previous level; suppose
               //the probability assignment of $e_i$ labeled with $(b_j, c^i)$ is $p_i^*$
6.             $p_i^* := \frac{2^{H(T_i)}}{1 + \sum_{1 \le j \le l} 2^{H(T_j)}}$, $i = 1, \cdots, l$;
7.             $H(t_h) := \sum_{1 \le i \le l} p_i^* H(T_i) - \sum_{1 \le i \le l} p_i^* \log p_i^* - (1 - \sum_{1 \le i \le l} p_i^*) \log(1 - \sum_{1 \le i \le l} p_i^*)$;
8.         $H(t_N) := \sum_{1 \le h \le q} H(t_h)$;
           //$t_N$ is the child-tree under the edge $e = \langle N', N \rangle$ for some $N'$
9. Return $p^*(\cdot)$.
```

The time complexity of ALG-p^*(T) is $O(n)$, where n is the size of T.

Example 8. Consider the trace-specification tree T shown in Fig. 3 (1). Now we calculate the probability assignments of edges in the worst case. For the node N_4, it has a $(b_3, \cdot)$-component tree and a $(b_4, \cdot)$-component tree. For the $(b_3, \cdot)$-component tree, the child-tree under the edge $\langle N_4, N_5 \rangle$ is empty and with entropy 0; hence $p^*(\langle N_4, N_5 \rangle) = \frac{2^0}{1+2^0} = \frac{1}{2}$. For the $(b_4, \cdot)$-component tree, the child-trees T_1 and T_2 under the edges $\langle N_4, N_6 \rangle$ and $\langle N_4, N_7 \rangle$, respectively, are also empty; hence $p^*(\langle N_4, N_6 \rangle) = p^*(\langle N_4, N_7 \rangle) = \frac{2^0}{1+(2^0+2^0)} = \frac{1}{3}$. Similarly, we have $p^*(\langle N_2, N_3 \rangle) = \frac{1}{2}$, $p^*(\langle N_0, N_1 \rangle) = \frac{1}{2}$, $p^*(\langle N_0, N_2 \rangle) = \frac{2}{9}$, and $p^*(\langle N_0, N_4 \rangle) = \frac{6}{9}$. Also, the entropy of T in the worst case is $H(T) = \log 18$ bits. □

Now, one can find the k-information-optimal testing strategy and the greedy information-optimal testing strategy of an input testing tree in the worst case using the algorithms ALG-opt(T, T_Π, k) and ALG-greedy-opt(T, T_Π), respectively, by running the algorithm ALG-p^*(T) first to give the probability assignments of edges in the worst case. Note that, as before, the global information-optimal testing strategy may or may not exist.

Example 9. Consider the trace-specification tree T in Fig. 3 (1) and its corresponding input testing tree T_Π in Fig. 3 (2). In the worst case that the entropy $H(T)$ reaches the maximum, let $k = 3$. Then a k-information-optimal testing strategy of T_Π is $\langle N_0', N_2' \rangle, \langle N_2', N_4' \rangle, \langle N_0', N_1' \rangle$. The greedy information-optimal strategy is $\langle N_0', N_2' \rangle, \langle N_2', N_4' \rangle, \langle N_0', N_1' \rangle, \langle N_2', N_3' \rangle$. Note that in this example, the global information-optimal testing strategy exists, which is (in this order) $\langle N_0', N_2' \rangle, \langle N_2', N_4' \rangle, \langle N_0', N_1' \rangle, \langle N_2', N_3' \rangle$. □

4. Information-Optimal Testing Strategies on Automata Used as Tree Representations

Throughout this section, the system Sys under test is assumed to be output-deterministic. Additionally, we assume that a trace-specification tree is *tight*; i.e., at each node, each input symbol has at most one output symbol. Formally,

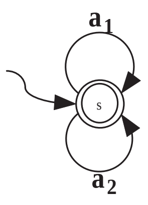

Fig. 7. A DFA A whose accepting language is $(a_1 + a_2)^*$.

T is tight if, for each node N in T and each input symbol $b \in \Pi$, there is at most one edge from node N with label (b, c), for some $c \in \Gamma$. In this case, a trace-specification tree and its input testing tree have exactly the same topological appearance (see Fig. 6 for an example). Since T is tight, for a test case $\omega = b_1 \cdots b_l$ (for some l) in the input testing tree T_Π, there is a unique path in T with labels $(b_1, c_1) \cdots (b_l, c_l)$, for some $c_1, \cdots, c_l \in \Gamma$. That is, the expected output $c_1 \cdots c_l$ is unique and already specified in T for the test case $\omega = b_1 \cdots b_l$. Therefore, we do not distinguish the two trees, and simply treat the trace-specification tree T as the *testing tree*. Because of this, in this section, we also call the input-output sequence $(b_1, c_1) \cdots (b_l, c_l)$ as a test case. Let $\Sigma = \Pi \times \Gamma$. Hence, the testing tree T is simply a tree with labels in Σ.

Recall that, the original trace-specification, $P_{original}$, could be an infinite set of words over the interface Σ; the trace-specification P we actually plan to test is the set of words where each word is a prefix (of length $\leq d$) of some word in $P_{original}$. In other words, P can be specified by $P_{original}$ together with d. One can often use a (deterministic finite) automaton A to represent $P_{original}$ (when it is regular), since it is practically a more succinct model compared with the tree representation of the trace-specification P. For instance, for the original trace-specification $P_{original} = (a_1 + a_2)^*$, the trace-specification we actually test is $P = (a_1 + a_2)^{\leq d}$. The testing tree T representing P is a complete binary tree, whose size is exponential in d. On the other hand, if we use an automaton A in Fig. 7 to accept the language $(a_1 + a_2)^*$ (i.e., $P_{original}$), the automaton only has one state and two transitions. Therefore, the trace-specification P we actually test can be specified by two parameters: an automaton A representing $P_{original}$, and the longest length d we could test for each sequence in $P_{original}$. Suppose that P is given as an automaton A and a length d. If we calculate the entropy of the testing tree T representing P by building T (using `ALG-entropy-tree(`T`)`), the time complexity would be exponential in the size of the automaton A and the length d. In the following, we discuss how to efficiently calculate the worst-case entropy of the testing tree T, and develop information-optimal testing strategies directly on the automaton A, without building T.

Let $A = \langle S, s_{init}, F, \Sigma, R \rangle$ be a DFA specified in Section 2.1. We further define $A(s) = \langle S, s, F, \Sigma, R \rangle$ as the finite automaton that keeps all the parameters in A except that it changes the initial state s_{init} to $s \in S$. Let $L(A, s, d)$ be the set of words such that each word is a prefix (of length $\leq d$) of some word in $L(A(s))$.

Let T be the testing tree that represents P. Recall that P is truncated from $P_{original} = L(A)$ up to length d; i.e., $P = L(A, s_{init}, d)$. Now we develop algorithms to calculate the entropy of T in the worst case (i.e., the entropy $H(T)$ reaches the maximum). The algorithms are based on Proposition 1 and 2. Note that since T is tight, a $(b, \cdot)$-component tree T_b, with $b \in \Pi$, in Fig. 4 now exactly consists of one edge e originating from the root and a child-tree t' under e, as shown in Fig. 8; hence the formula in Proposition 2,

$$H(T_b) = \sum_{1 \leq i \leq l} p_i H(T_i) - \sum_{1 \leq i \leq l} p_i \log p_i - (1 - \sum_{1 \leq i \leq l} p_i) \log(1 - \sum_{1 \leq i \leq l} p_i),$$

now can be written as

$$H(T_b) = H_{\texttt{Binary}}(p(e)) + p(e) H(t'),$$

where $H_{\texttt{Binary}}(\cdot)$ is the binary entropy function, $H_{\texttt{Binary}}(p(e)) = -p(e) \log p(e) - (1 - p(e)) \log(1 - p(e))$. Therefore,

$$H(T) = \sum_i (H_{\texttt{Binary}}(p(e_i)) + p(e_i) H(T_i)), \tag{13}$$

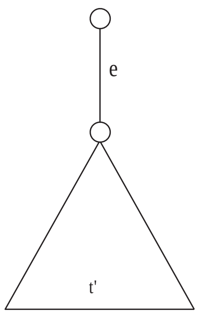

Fig. 8. The tree T_b consists of a child-tree t' and an edge e from the root of T to the root of t'.

where e_i is an edge that directly originates from the root of T, and T_i is the child-tree under e_i. In the worst case, from (12), we have $p^*(e_i) = \frac{2^{H(T_i)}}{1+2^{H(T_i)}}$, and therefore (13) can be written as

$$H(T) = \sum_i \log(1 + 2^{H(T_i)}). \tag{14}$$

Recall that the tree T is represented by a given automaton A. To use (14) to calculate the worst-case entropy $H(T)$ directly on the automaton A, we build an array $h_s[0 \cdots d]$ with $d+1$ entries for each state s in the automaton A, and the $level^{th}$ ($0 \le level \le d$) entry $h_s[level]$ equals $H(A, s, level)$, where $H(A, s, level)$ is the (worst case) entropy of the testing tree that represents $L(A, s, level)$. Clearly, for the testing tree that represents $L(A, s, level)$, each of its child-trees represents $L(A, s', level - 1)$, for each direct successor s' of s (i.e., $(s, a, s') \in R$ for some $a \in \Sigma$), respectively. From (14), we have

$$h_s[level] = \sum_{s' \in \mathtt{SUCC}(s)} \log(1 + 2^{h_{s'}[level-1]}),$$

where $\mathtt{SUCC}(s)$ is the set of direct successors of s. Note that $\mathtt{SUCC}(s)$ could be a *multiset*; i.e., if there are multiple transitions from s to s', then s' should be counted multiple times in $\mathtt{SUCC}(s)$. In particular, $h_s[0] = 0$ for each state s, which means that, the testing tree representing $L(A, s, 0)$ is an empty tree, therefore, $H(A, s, 0) = 0$. The algorithm `ALG-entropy-DFA-worst` (A, d) is given as follows.

```
ALG-entropy-DFA-worst $(A,d)$:
//To calculate the entropy of the testing tree representing $P = L(A, s_{init}, d)$
//in the worst case. The return value is the entropy $h_s[level]$ of the testing
//tree representing $L(A,s,level)$, for each state $s$ in $A$, and $0 \le level \le d$.
1. For each $s \in S$ and for $0 < level \le d$
2.     $h_s[level] :=$ null;
3.     $h_s[0] := 0$;
4. $level := 1$;
5. Repeat
6.     For each state $s$
7.          $h_s[level] := \sum_{s' \in \mathtt{SUCC}(s)} \log(1 + 2^{h_{s'}[level-1]})$;
8.     $level := level + 1$;
9. Until $level \ge d + 1$;
10.Return $h_s[0 \cdots d]$ for each state $s$.
```

The time complexity of the algorithm `ALG-entropy-DFA-worst` (A, d) is $O(dn)$, where n is the number of transitions in A (also called the size of A). When we finish running the above algorithm on A, we will get an array for each state s, and the value $h_s[level]$ equals $H(A, s, level)$. Therefore, $h_{s_{init}}[d]$ is the entropy of the testing tree T representing $L(A, s_{init}, d)$, which is the trace-specification P that we actually test, as mentioned earlier.

In the following, we show an example of running the algorithm `ALG-entropy-DFA-worst` (A, d) on an automaton A with $d = 3$.

Example 10. An automaton A is given in Fig. 9, with s_1 being the initial state, and s_5 being the accepting state.

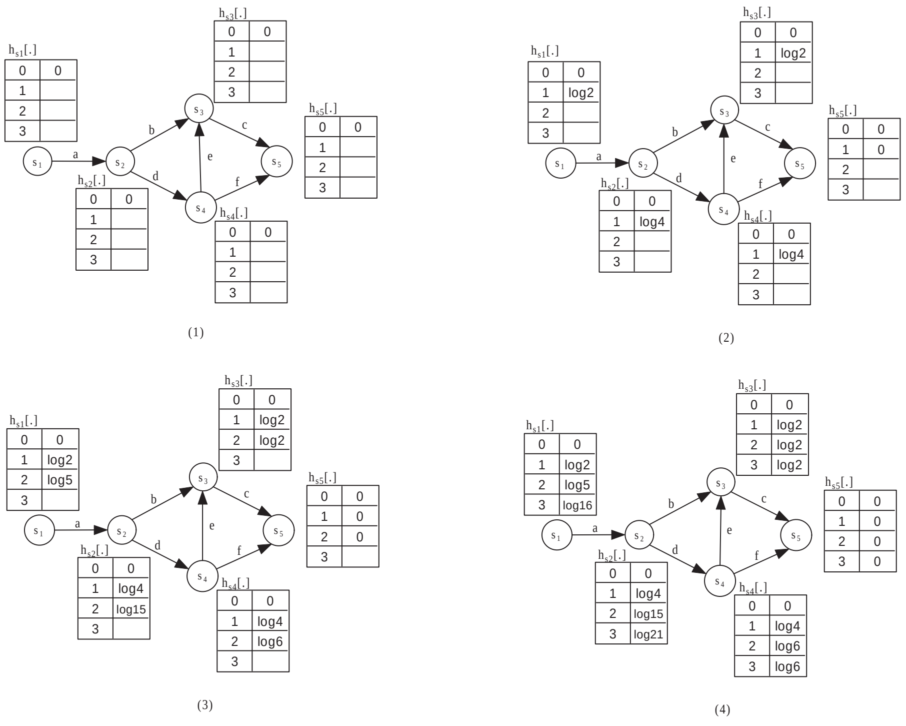

Fig. 9. Run `ALG-entropy-DFA-worst` (A, d) on a DFA A with $d = 3$.

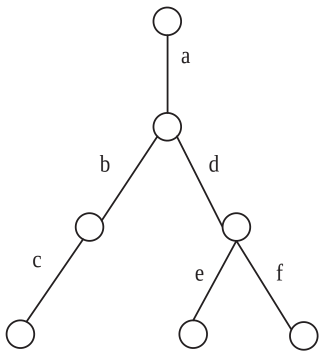

Fig. 10. The testing tree representing $P = L(A, s_1, 3)$.

Recall that a, b, c, d, e, f in Fig. 9 are in $\Sigma = \Pi \times \Gamma$. Assume that we can only test up to $d = 3$ steps for each string in $L(A)$; i.e., the trace-specification P we actually test is $L(A, s_1, 3)$. Fig. 9 (1) $\sim$ (4) shows how the arrays of states evolve while `ALG-entropy-DFA-worst` (A, d) is running on A (blank entries denotes for `null`). At the beginning, lines $1 \sim 3$ initialize the entries as in Fig. 9 (1), which corresponds to $H(A, s, 0) = 0$ for every $s \in S$, since the set of strings originating from s of length 0 is empty. For each state $s \in S$, we gradually update its entries according to lines $6 \sim 8$ as shown in Fig. 9 (2) $\sim$ (4). Finally, $h_{s_1}[3] = \log 16$ implies that $H(A, s_1, 3) = \log 16 = 4$ bits. That is, the entropy of the testing tree T representing $P = L(A, s_1, d)$ is 4 bits. We can check the result with the tree T in Fig. 10. In the worst case, $H(T) = 4$ bits, which coincides with $h_{s_1}[3]$. $\square$

Note that, the number d is usually pre-given. However, if A is a directed acyclic graph (DAG), we could set d to the length of the longest string in $L(A)$ (which could be found using depth-first-search) such that the trace-specification we actually test is $L(A)$; i.e., $P = P_{original}$.

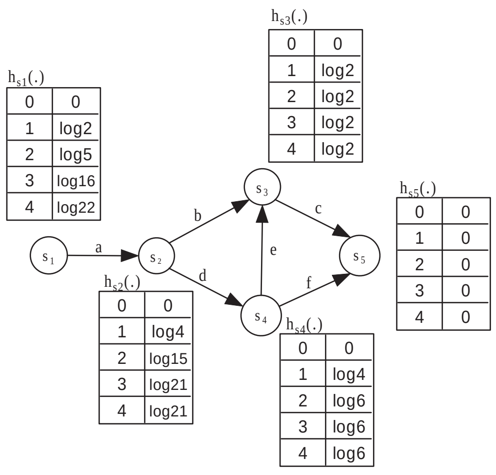

Fig. 11. Run `ALG-entropy-DFA-worst` (A, d) on the finite automaton A in Fig. 9, with $d = 4$.

Example 11. In the DFA A in Fig. 9, the longest string in $L(A)$ is of length 4. Therefore, we set d to 4, and run `ALG-entropy-DFA-worst` (A, d) on A. In this case, the trace-specification P is $L(A)$. The result is shown in Fig. 11. Since $h_{s_1}[4] = \log 22$, the entropy of the testing tree T that represents $P = L(A)$ is $\log 22$. □

Next, we study information-optimal testing strategies of the trace-specification P in the worst case directly on the finite automaton A. Recall that in the testing tree T that represents P, an edge e is labeled with some symbol $a \in \Sigma = \Pi \times \Gamma$. $p(e)$ denotes the probability that e is connected; i.e., the symbol a *succeeds* (in fact, $p(e)$ is the probability that a succeeds, given that the symbols labeling the parent of e succeeds. In the sequel, we simply say that $p(e)$ is the probability that a succeeds, when the context is clear). Correspondingly, we also have the probability that symbol a succeeds in the automata representation. Let $r = (s, a, s')$ be a transition in the finite automaton A. We define $p(r(level))$ as the probability that, in the testing tree representing $L(A, s, level)$, the symbol a (that is labeled on an edge originating from the root of the testing tree) succeeds, for each $1 \le level \le d$. Recall that in the worst case, the probability of an edge e is $p^*(e) = \frac{2^{H(t')}}{1+2^{H(t')}}$, where t' is the child-tree under e. In the DFA A, after running `ALG-entropy-DFA-worst` (A, d) on A, $h_s[level]$ is the entropy of the testing tree t that represents $L(A, s, level)$, and $h_{s'}[level - 1]$ is the entropy of the testing tree t' that represents $L(A, s', level - 1)$. Clearly, t' is a child-tree of t. Therefore, in the worst case, $p(r(level)) = p^*(r(level)) = \frac{2^{h_{s'}[level-1]}}{1+2^{h_{s'}[level-1]}}$. The algorithm that calculates probability assignments in the worst case for transitions in the DFA A, $p^*(\cdot)$, is given as follows.

```
ALG-$p^*$-DFA-worst$(A, d)$ :
//To calculate the probability assignments $p^*(\cdot)$ in the worst case in a DFA $A$
//when only testing sequences of length up to $d$. The return value is
//the probability $p^*(r(level))$ for each transition $r$ in $A$, $1 \le level \le d$.
1. Run ALG-entropy-DFA-worst$(A, d)$ on $A$;
   //This will return $h_s[0 \cdots d]$ for each state $s$.
2. For each transition $r = (s, a, s')$ in $A$
3.     For each $1 \le level \le d$
4.         $p^*(r(level)) := \frac{2^{h_{s'}[level-1]}}{1+2^{h_{s'}[level-1]}}$;
5. Return $p^*(\cdot)$.
```

The time complexity of `ALG-`p^*`-DFA-worst`(A, d) is actually the time complexity of `ALG-entropy-DFA-worst`(A, d), which is $O(dn)$. (From now on, when the context is clear, we simply use a to denote a transition (s, a, s')).

Example 12. Now we calculate the probability assignments $p^*(\cdot)$ in the DFA A shown in Fig. 11. After running `ALG-`p^*`-DFA-worst`(A, d), $p^*(a(4)) = 21/22$, $p^*(b(4)) = 2/3$, $p^*(d(4)) = 6/7$, $p^*(e(4)) = 2/3$, $p^*(c(4)) =$

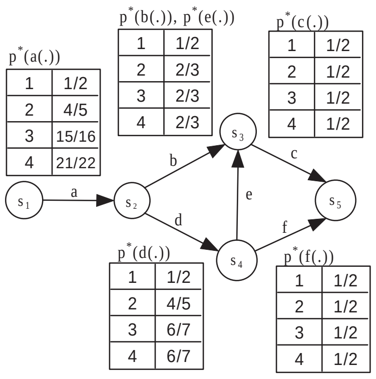

Fig. 12. Run `ALG-`p^*`-DFA-worst` (A, d) on the finite automaton A in Fig. 11, with $d = 4$.

$1/2$, and $p^*(f(4)) = 1/2$. The complete probability assignments $p^*(r(level))$ for each transition r and $1 \leq level \leq 4$ are illustrated in Fig. 12. □

Remember that a finite automaton is just another form of a testing tree. A testing strategy $\mathcal{C}$ of a finite automaton A, which resembles a testing strategy of a testing tree, specifies an ordering in which we test sequences in the trace-specification $P = L(A, s_{init}, d)$. Now that we have ways to calculate probability assignments on DFA A, we can develop algorithms to calculate testing strategies of A. Before giving the formal definition of a testing strategy on a DFA, we first introduce some notation. Let α be a string and L be a language. We say $\alpha \prec L$, if α is a prefix of some word in L. We use $\alpha = \beta \circ \kappa$ to denote the string α that is a concatenation of β and κ, where each of β and κ can either be a symbol or a string. Sometimes, we simply write $\alpha = \beta\kappa$. Suppose that $\alpha = \omega a \prec L(A, s_{init}, d)$. When A runs on α, we use $s(\alpha)$ to denote the state of A right after the last symbol a in $\alpha = \omega a$ is read. In particular, for an empty string ϵ, we define $s(\epsilon) = s_{init}$. A testing strategy $\mathcal{C}$ of a DFA A with respect to a given length d is in the form of

$$\mathcal{C} = \alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i)}, \alpha_{\mathcal{C}(i+1)}, \cdots, \alpha_{\mathcal{C}(g)},$$

for some $g > 0$, where each string $\alpha_{\mathcal{C}(i)} = \omega a \prec L(A, s_{init}, d)$ for some ω and a. Sometimes, we just call $\mathcal{C}$ a testing strategy of $L(A, s_{init}, d)$. By testing $\alpha_{\mathcal{C}(i)} = \omega a$ in the testing strategy $\mathcal{C}$, we mean to test the rightmost symbol a in $\alpha_{\mathcal{C}(i)}$, and symbols in ω have already been tested. In this way, the testing strategy $\mathcal{C}$ actually specifies a testing strategy in the corresponding testing tree T that represents $L(A, s_{init}, d)$. Therefore, a prefix of $\mathcal{C}$ corresponds to a subtree $t \prec T$ that represents the strings in the prefix. We simply call the entropy of that subtree t as the gain of that prefix of the testing strategy $\mathcal{C}$. The number g is pre-given. In fact, in order to make the strategy $\mathcal{C}$ be an exhaustive testing strategy of $L(A, s_{init}, d)$, the g, in worst case, can be exponential in n (the number of transitions in A). In practice, such a long strategy may not be exhaustively tested anyway. Therefore, one can expect that the given length g of the strategy is not unreasonably large.

For the k-information-optimal testing strategy (with worst-case entropy) of a DFA A, the idea to calculate it is similar to `ALG-opt` (T, k). We first run `ALG-`p^*`-DFA-worst` (A, d) to obtain the worst-case probability assignments $p^*(\cdot)$. We associate each state s in A with two arrays $H_s[\cdot, \cdot]$ and $OPT_s[\cdot, \cdot]$. The meanings of the two arrays are explained as follows. Let T be the testing tree representing $P = L(A, s_{init}, d)$. For a state s in A and a number $0 \leq level \leq d$, consider the subtree $T(s, level)$ of T that represents $L(A, s, level)$, which keeps the probability assignments of edges in T. $H_s[level, i]$ and $OPT_s[level, i]$ record the entropy and the set of root paths (i.e., the k-information-optimal testing strategy) of the i-information-optimal subtree of $T(s, level)$, for each $0 \leq level \leq d$ and $0 \leq i \leq k$, respectively. We initialize $H_s[level, i] = 0$ and $OPT_s[level, i] = \emptyset$ for each $0 \leq level \leq d$ and $0 \leq i \leq k$. For each state s in A, suppose that $r_1, \cdots, r_q$ are all the transitions from s, and $r_j = (s, a_j, s'_j)$ for some a_j and s'_j, for all $1 \leq j \leq q$. Note that the $s'_1, \cdots, s'_q$ are not necessarily distinct. From the i-information-optimal subtree of $T(s'_j, level)$ (whose entropy is stored in $H_{s'_j}[level, i]$), we can calculate the $(i+1)$-information-optimal subtree of the component tree $T_{s'_j}$ which consists of an edge labeled with a_j together with $T(s'_j, level)$ under the edge. Similar to the algorithm ALG-opt (T, k), we define another array $Y_j[\cdot]$ to record the entropy of that $(i+1)$-information-optimal subtree of $T_{s'_j}$ in $Y_j[i+1]$. In this way, we obtain arrays $Y_1[\cdot], \cdots, Y_q[\cdot]$. The problem of calculating the entropy of the i-information-optimal subtree of $T(s, level+1)$ (i.e., $H_s(level+1, i)$) now becomes selecting indices $\texttt{index}_1, \cdots, \texttt{index}_q$, satisfying $\sum_{1\le j\le q} \texttt{index}_j = i$, for $Y_1, \cdots, Y_q$, such that $\sum_{1\le j\le q} Y_j[\texttt{index}_j]$ achieves the maximum, which, again, can be solved by the algorithm MAX-SELECT. The algorithm ALG-opt-DFA-worst (A, d, k) that calculates the k-information-optimal subtree of the testing tree representing $L(A, s_{init}, d)$ is given as follows, where $H_s[\cdot,\cdot]$ and $OPT_s[\cdot,\cdot]$ are global variables, which are already initialized in above.

```
ALG-opt-DFA-worst($A, d, k$):
//To calculate the $k$-information-optimal subtree of the tree representing
//$L(A, s_{init}, d)$. Assume that probability assignments in the worst case $p^*(\cdot)$ are
//pre-calculated using ALG-$p^*$-DFA-worst($A, d$). The return values has two
//parts: the entropy $H_{s_{init}}[d,k]$ and the set of root paths (i.e., the
//$k$-information-optimal testing strategy) $OPT_{s_{init}}[d,k]$.
//$H_s[\cdot,\cdot]$ and $OPT_s[\cdot,\cdot]$ are global variables, which are initialized as in above.
1. If $k = 0$
2.     For each state $s$ and each $0 \le level \le d$
3.         $H_s[level, k] := 0$ and $OPT_s[level, k] := \emptyset$;
4.     Return;
5. Run ALG-opt-DFA-worst($A, d, k-1$);
6. For each state $s$ that has at least one successor
7.     For $level := 1$ to $d$
           //suppose that $r_1, \cdots, r_q$, for some $q \ge 1$, are all the transitions
           //from state $s$. We use $s'_1, \cdots, s'_q$ to denote the target-states in the
           //transitions (note that the $s'_1, \cdots, s'_q$ are not necessarily distinct).
8.         For each $1 \le j \le q$
9.             $Y_j[0] := 0$;
10.            For $0 \le i \le k-1$
11.                $Y_j[i+1] := p^*(r_j(level)) \cdot H_{s'_j}[level-1, i] + H(p^*(r_j(level)))$;
12.        Run MAX-SELECT($\{Y_1, \cdots, Y_q\}, k$);
13.        $H_s[level, k] := \sum_{j:\texttt{index}_j \ne 0} Y_j[\texttt{index}_j]$;
14.        $OPT_s[level, k] := \bigcup_{j:\texttt{index}_j \ne 0} \{a_j\} \cup \{a_j \circ \omega : \omega \in OPT_{s'_j}[\texttt{index}_j - 1]\}$;
           //$a_j$ is the symbol on transition $r_j$
15. Return $H_{s_{init}}[d,k]$ and $OPT_{s_{init}}[d,k]$.
```

Following the analysis of algorithm ALG-opt (T, k), we can obtain that the (worst-case) time complexity of ALG-opt-DFA-worst (A, d, k) is $O(ndk^3)$, where n is the number of transitions in A.

For the greedy information-optimal testing strategy, similarly as in the case for the testing tree, we also have the *available set* of $\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\}$, denoted as $\texttt{AS}(\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\})$, to represent the set of strings that are qualified to be $\alpha_{\mathcal{C}(i)}$. Formally, for a set of strings L, we define $\texttt{AS}(L) = \{\omega a : \omega \in L \text{ and } s(\omega, a, s') \in R \text{ for some state } s'\}$. That is, the $\texttt{AS}(L)$ is the set of string ωa that are extended from a string ω in L, with an additional symbol a such that the automaton A will not crash after reading through ωa. Notice that, when strings in $\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\}$ form a subtree $t \prec T$ (in here, each string in t is a path), we can show that, strings in $\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i)}\}$ also form a subtree $t' \prec t'$ with $t \prec t' \prec T$, whenever $\alpha_{\mathcal{C}(i)} \in \texttt{AS}(\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\})$. Now consider $\alpha_{\mathcal{C}(i)} = \omega a$, for some ω and a. Suppose that, when running A on the word ωa, the last transition fired is $r = (s, a, s')$ for some s, s'. When one tests the last symbol a in the ωa (so, in this case, all symbols in ω are already tested), the probability that a succeeds in the worst case, as show in ALG-p^*-DFA-worst (A, d), is $p^*(r(d-|\omega|)) = p^*(r(d - |\alpha_{\mathcal{C}(i)}| + 1))$. Modifying the algorithm ALG-greedy-opt (T) that calculates the greedy information-optimal testing strategy of a testing tree, we have the following algorithm, ALG-greedy-DFA-worst (A, d), that calculates the greedy information-optimal testing strategy of a DFA A in the worst case.

```
ALG-greedy-DFA-worst($A, d$):
//To calculate the greedy information-optimal testing strategy of a DFA $A$,
//when only testing sequences of length up to $d$.
//The return value is the greedy information-optimal testing strategy $\mathcal{C}^*$ of $A$.
```

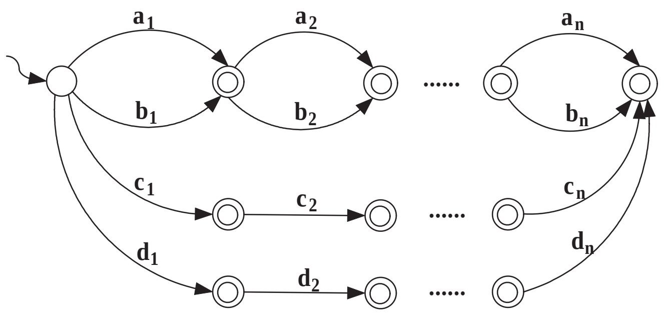

Fig. 13. An example finite automaton A.

```
1. run ALG-$p^*$-DFA-worst$(A,d)$ on $A$;
   //This gives probability assignment $p^*(r(level))$ for all transitions
   //$r \in R$ in $A$ and $1 \le level \le d$.
2. $\mathtt{AS}[1] := \{a : r = (s_{init}, a, s') \in R$ for some $s'\}$;
3. For each $1 \le i \le g$
4.     If $\mathtt{AS}[i] = \emptyset$
5.         Return $\mathcal{C}^* = \alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}$;
6.     Else
7.         For each $\alpha \in \mathtt{AS}[i]$
8.             If $|\alpha| = 1$
9.                 $\Delta(\alpha) := H(p^*(r(d)))$;
                   //$\alpha = a$, and $r = (s_{init}, a, s')$ for some $s'$.
10.            Else
11.                $\Delta(\alpha) := (\prod_{1 \le j \le |\alpha|-1} p(r_j(d-j+1)))H(p^*(r_{|\alpha|}(d-|\alpha|+1)))$;
                   //$\alpha = a_1 \cdots a_j \cdots a_{|\alpha|-1} a_\alpha$, and $r_j = (s, a_j, s')$ for some $s$ and $s'$.
12.        Suppose that $\alpha^* \in \mathtt{AS}[i]$ achieves $\max_{\alpha \in \mathtt{AS}(\alpha_{\mathcal{C}(i)})} \Delta(\alpha)$;
13.        $\alpha_{\mathcal{C}(i)} := \alpha^*$;
14.        $\mathtt{AS}[i+1] := (\mathtt{AS}[i] - \{\alpha_{\mathcal{C}(i)}\}) \cup \{\alpha_{\mathcal{C}(i)} \circ a : (s(\alpha_{\mathcal{C}(i)}), a, s') \in R$ for some $s'\}$;
           //Now, $\mathtt{AS}[i+1] = \mathtt{AS}(\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i)}\})$
15.Return $\mathcal{C}^* = \alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(g)}$.
```

Let b_s be the number of transitions starting from a state s, and in worst case, $b_s = O(n)$, where n is the number of transitions in A. We assume that $g > d$. The time complexity of ALG-greedy-DFA-worst(A,d) is $O(dn) + O(gb_s \log b_s) = O(gn \log n)$, where $O(dn)$ is time of running ALG-p^*-DFA-worst(A,d), and $O(gn \log n)$ is time of running lines $7 \sim 14$ for maximally g times.

Example 13. Following ALG-greedy-DFA-worst(A,d), we calculate the greedy information-optimal testing strategy $\mathcal{C}^*$ for the DFA A in Fig. 11, where $d = 4$. We set g to a relatively large number such that the testing strategy obtained is the exhaustive testing of $P = L(A, s_{init}, d)$. At the beginning, from line 2 in ALG-greedy-DFA-worst(A,d), $\mathtt{AS}[1] = \{a\}$; therefore, $\alpha_{\mathcal{C}(1)} = a$, and $\mathtt{AS}[2] = \{ab, ad\}$ according to line 9. Since the probability that the symbol b in ab succeeds is $p^*(b(4 - |ab| + 1)) = p(b(3)) = 2/3$, and the probability that the symbol d in ad succeeds is $p^*(d(3)) = 6/7$. We have $\Delta(ab) = p^*(a(4))H(p^*(b(3))) = 0.88$, and $\Delta(ad) = p^*(a(4))H(p^*(d(3))) = 0.56$; from lines 12 and 13, we have $\alpha_{\mathcal{C}(2)} = ab$ and $\mathtt{AS}[3] = \{ad, abc\}$. Keep doing so, until $\mathtt{AS}[i] = \emptyset$ for some i, and finally we obtain the greedy information-optimal testing strategy $\mathcal{C}^* = a, ab, abc, ad, adf, ade, adec$ (in this order). □

Finally, we show that a test set that achieve 100% branch coverage could still reveal very little information of a software system. Consider the finite automaton A (which can be interpreted as a software design) shown in Fig. 13. Let a test set t consist of $a_1 \cdots a_n$, $b_1 \cdots b_n$, $c_1 \cdots c_n$, $d_1 \cdots d_n$ and together with all their prefixes. Clearly, t achieves 100% branch coverage in A. However, one can show that, under worst-case probability assignments of edges, the ratio of t's information gain to the entropy of the automaton approaches 0 as $n \to \infty$. On the other hand, two test sets sharing the same branch coverage may gain dramatically different amount of information. For instance, consider the two test sets

- t_1 consisting of $a_1 \cdots a_n$, $b_1 \cdots b_n$, and together with all their prefixes,
- t_2 consisting of $c_1 \cdots c_n$, $d_1 \cdots d_n$, and together with all their prefixes.

Clearly, t_1 and t_2 both achieve 50% branch coverage. One can show that under worst-case probability assignments of edges, the ratio of the information gain of t_1 to the information gain of t_2 approaches to ∞ as $n \to \infty$. Similarly, our conclusions remain for path coverage. This example implies that the path $a_1 \cdots a_n$ is "more important" (i.e., contains much more information) than the path $c_1 \cdots c_n$ as n becomes large. Notice that, under path coverage, these two paths are not distinguishable since they have the same path coverage. As the information gain criterion is syntax-independent, it provides a way to compare test sets that are not differentiable under other traditional testing criteria and helps us select a set that achieves maximal information gain using algorithms presented earlier.

5. Conclusion

In our understanding, software testing is a cooling-down process, during which the entropy (or uncertainty) of the system under test decreases. In this paper, we have studied information-optimal software testing where test cases are selected to gain the most information (i.e., cools down the system under test fastest). More specifically, we represent a trace-specification (a finite set of input-output sequences) as a tree, that the black-box transition system under test is intended to conform with. When the tree is associated with pre-given probability assignments on the edges, we have developed polynomial-time algorithms calculating the entropy of the trace-specification and computing k-information-optimal and greedy information-optimal testing strategies. When the tree is not pre-given with probability assignments on the edges, we have studied efficient algorithms calculating the assignments that make the trace-specification be with maximal entropy (i.e., the worst case that we know the least amount of information about the system under test). In this latter case, we have also provided polynomial-time algorithms computing k-information-optimal and greedy information-optimal testing strategies. Finally, we have generalized our algorithms to find information-optimal testing strategies to the case when the trace-specification is tight and when the trace-specification is, often succinctly, represented as a finite automaton. We shall emphasize that the information-optimal testing strategies are calculated *before* any testing is performed.

We now briefly discuss testing an output-nondeterministic system Sys. We assume that there is an oracle $\mathcal{O}$ (i.e., a test engine) to consume a test case (which is a sequence of input-output pairs) and provide a sequence of Boolean values to tell whether the prefixes of a test case are the observable behaviors of Sys. Logically, we can treat the Sys as an output-deterministic system Sys' where an input-output pair in Sys is an input symbol in Sys' and the output symbols in Sys' are simply Booleans. However, this does not confirm the following statement: testing output-nondeterministic systems is special case of testing output-deterministic ones. A precise conclusion should be, once the oracle $\mathcal{O}$ for testing output-nondeterministic system is built, the statement is true. However, as we have mentioned earlier, the oracle is difficult to build in practice, which makes testing nondeterministic systems hard.

In the future, we plan to compare our cooling-down testing approaches with existing structural testing techniques and see if our approaches can be combined with the existing techniques to improve their effectiveness. We will also study an information-theoretic information-optimal approach in testing concurrent systems. However, the entropy of a concurrent system is difficult (even in theory) to calculate because of the communications between components. We have studied a simple model where in a concurrent system, two identical components communicate with each other only through synchronization. For this simple model, we are able to effectively calculate the entropy of the concurrent system and its information-optimal testing strategies (the strategy is decompositional; i.e., testing each component alone without integrating testing. See also [XiD05, Xie05] for similar ideas.). We plan to study similar algorithms for more general concurrent system models and determine how the communications between components affect the information-optimal testing over the entire system.

References

[AmO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[AOH03] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 99–107. IEEE Computer Society, 2003.

[ALR09] C. Andrés, L. Llana, and I. Rodríguez. Formally transforming user-model testing problems into implementer-model testing problems and viceversa. *Journal of Logic and Algebraic Programming*, 78(6):425–453, 2009.

[Bei95] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, 1995.

[BJK05] M. Broy, B. Jonsson, J-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Lecture Notes in Computer Science, Vol. 3472, Springer, 2005.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[CoT06] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, second edition, 2006.

[DuN84] J. W. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, 1984.

[Gau95] M. C. Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, 1995. Lecture Notes in Computer Science, Vol. 915, pages 82–96, Springer.

[GoG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510. ACM, 1975.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HaP89] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1989.

[Hei02] C. Heitmeyer. Software cost reduction. *Encyclopedia of Software Engineering, second edition*, 2002.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HMU07] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2007.

[Lad07] J. Ladyman. Physics and computation: The status of landauer's principle. In *CiE '07: Proceedings of the 3rd conference on Computability in Europe*, 2007. Lecture Notes in Computer Science, Vol. 4497, pages 446–454, Springer.

[Lan61] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[LFK03] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *KDD '03: Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 388–396, 2003.

[LyT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[MyS04] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[NVS04] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29(4):55–64, 2004.

[PVY01] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, 2001.

[PYH03] A. Petrenko, N. Yevtushenko, and J. Huo. Testing transition systems with input and output testers. In *TestCom '03*, 2003. Lecture Notes in Computer Science, Vol. 2644, pages 129–145, Springer.

[Pnu77] A. Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[TSL04] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IRI'04: proceedings of IEEE Internation Conference on Information Reuse and Integration*, pages 483–498. IEEE Computer Society, 2004.

[Tre08] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, 2008. Lecture Notes in Computer Science, Vol. 4949, pages 1–38, Springer.

[TrB03] J. Tretmans and E. Brinksma. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

[VaP05] M. van der Bijl and F. Peureux. I/O-automata based testing. *Chapter 7 in Model-Based Testing of Reactive Systems. Lecture Notes in Computer Science, Vol. 3472, Springer*, pages 173–200, 2005.

[WeJ91] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17(7):703–711, 1991.

[Xie05] G. Xie. *Fundamental Studies on Automatic Verification of Component-based Systems – A Decompositional and Hybrid Approach*. PhD thesis, Washington State University, 2005.

[XiD05] G. Xie and Z. Dang. Testing systems of concurrent black-boxes - an automata-theoretic and decompositional approach. In *FATES'05: Proceedings of the 5th International Workshop on Formal Approaches to Software Testing*, 2006. Lecture Notes in Computer Science, Vol. 3997, pages 170–186, Springer.

[YDF09] L. Yang, Z. Dang and T. R. Fischer. Optimal Software Testing - A Cooling Down Process. In *FCS'09: Proceedings of the International Conference on Foundations of Computer Science*, pages 162–168, 2009.

[ZhH92] H. Zhu and P. A. V. Hall. Test data adequacy measurement. *Softw. Eng. J.*, 8(1):21–29, 1992.

[ZHM97] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.