# Logic Extraction for Explainable AI

Susmit Jha

Computer Science Laboratory
SRI International
susmit.jha@sri.com

**Abstract.** In this paper, we investigate logic extraction as a means for building explainable AI. Blackbox AI system is used as an oracle that can label inputs with positive or negative label depending on its decision. We formulate generating explanations as the problem of learning Boolean formulae from examples obtained by actively querying such an oracle. This problem has exponential worst-case complexity in the general case as well as for many restrictions. In this paper, we focus on learning *sparse* Boolean formulae which depend on only a small (but unknown) subset of the overall vocabulary of atomic propositions. We propose two algorithms - first, based on binary search in the Hamming space, and the second, based on random walk on the Boolean hypercube, to learn these sparse Boolean formulae with a given confidence. This assumption of sparsity is motivated by the problem of mining explanations for decisions made by artificially intelligent (AI) algorithms, where the explanation of individual decisions may depend on a small but unknown subset of all the inputs to the algorithm. We demonstrate the use of these algorithms in automatically generating explanations of these decisions. We show that the number of examples needed for both proposed algorithms only grows logarithmically with the size of the vocabulary of atomic propositions. We illustrate the practical effectiveness of our approach on a diverse set of case studies. In this paper, we summarize the results presented in our recent work [7, 5].

## 1 Introduction

The rapid integration of intelligent and autonomous agents into our industrial and social infrastructure has created an immediate need for establishing trust between these agents and their human users. Decision-making and planning algorithms central to the operation of these systems currently lack the ability to explain the choices and decisions that they make. This is particularly problematic when the results returned by these algorithms are counter-intuitive. It is important that intelligent agents become capable of responding to inquiries from human users. For example, when riding in an autonomous taxi, we might expect to query the AI driver using questions similar to those we would ask a human driver, such as “why did we not take the Bay Bridge”, and receive a response such as “there is too much traffic on the bridge” or “there is an accident on the ramp leading to the bridge or in the middle lane of the bridge.” These

explanations are essentially formulae in propositional logic formed by combining the atomic propositions corresponding to the user-observable system and the environment states using Boolean connectives.

Even though the decisions of intelligent agents are the consequence of algorithmic processing of perceived system and environment states, the straightforward approach of reviewing this processing is not practical. There are three key reasons for this. First, AI algorithms use internal states and intermediate variables to make decisions which may not be observable or interpretable by a typical user. For example, reviewing decisions made by the A* planning algorithm [12] could reveal that a particular state was never considered in the priority queue. But this is not human-interpretable, because a user may not be familiar with the details of how A* works. Second, the efficiency and effectiveness of many AI algorithms relies on their ability to intelligently search for optimal decisions without deducing information not needed to accomplish the task, but some user inquiries may require information that was not inferred during the original execution of the algorithm. For example, a state may never be included in the queue of a heuristic search algorithm like A* because either it is unreachable or it has very high cost. Thus, the ability to explain why this state is not on the computed path will require additional effort. Third, artificial intelligence is often a composition of numerous machine learning and decision-making algorithms, and explicitly modeling each one of these algorithms is not practical. Instead, we need a technique which can treat these algorithms as black-box oracles, and obtain explanations by observing their output on selected inputs.

These observations motivate us to formulate the problem of generating explanations as an oracle-guided learning of Boolean formula where the AI algorithm is queried multiple times on carefully selected inputs to generate examples, which in turn are used to learn the explanation. Given the observable system and environment states, $S$ and $E$ respectively, typical explanations depend on only a small subset of elements in the overall vocabulary $V = S \cup E$, that is, if the set of state variables on which the explanation $\phi$ depends, is denoted by $support(\phi) \subseteq V$, then $|support(\phi)| << |V|$ (sparse). The number of examples needed to learn a Boolean formula is exponential in the size of the vocabulary in the general case [11, 10, 3]. Our approach builds on recent advances in formal synthesis [8, 4, 9].

We summarize the following recent contributions made towards logic extraction for explainable AI. We formulate the problem of finding explanations for decision-making AI algorithms as the problem of learning sparse Boolean formulae. We present two algorithms to learn sparse Boolean formula where the size of required examples grows logarithmically (in contrast to exponentially in the general case) with the size of the overall vocabulary. We theoretically and empirically compare these algorithms. The first algorithm is based on a binary search in the Hamming space first described in our earlier work [5]. The second algorithm is based on random walk in the Boolean hypercube reported in our earlier work [7]. We present case studies used to demonstrate the effectiveness of our approach.

## 2 Motivating Example

We describe a motivating example to illustrate the problem of providing human-interpretable explanations for the results of an AI algorithm. We consider the A* planning algorithm [12], which enjoys widespread use in path and motion planning due to its optimality and efficiency. Given a description of the state space and transitions between states as a weighted graph where weights are used to encode costs such as distance and time, A* starts from a specific node in the graph and constructs a tree of paths starting from that node, expanding paths in a best-first fashion until one of them reaches the predetermined goal node. At each iteration, A* determines which of its partial paths is most promising and should be expanded. This decision is based on the estimate of the cost-to-go to the goal node. Specifically, A* selects an intermediate node $n$ that minimizes $\texttt{totalCost}(n) = \texttt{partialCost}(n) + \texttt{guessCost}(n)$, where `totalCost` is the estimated total cost of the path that includes node $n$, obtained as the sum of the cost ($\texttt{partialCost}(n)$) of reaching $n$ from the initial node, and a heuristic estimate of the cost ($\texttt{guessCost}(n)$) of reaching the goal from $n$ . The heuristic function `guessCost` is problem-specific: e.g., when searching for the shortest path on a Manhattan grid with obstacles, a good `guessCost` is the straight line distance from the node $n$ to the final destination. Typical implementations of A* use a priority queue to perform the repeated selection of intermediate nodes. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest `totalCost` value is removed from the queue, and "expanded", This means that the `partialCost` values of its neighbors are updated accordingly based on whether going through $n$ improves them, and these neighbors are added to the queue. The algorithm continues until some goal node has the minimum cost value, `totalCost`, in the queue, or until the queue is empty (in which case no plan exists). The `totalCost` value of the goal node is then the cost of the optimal path. We refer readers to [12] for a detailed description of A*. In rest of this section, we illustrate the need for providing explanations using a simple example map and application of A* on it to find the shortest path.

Figure 1 depicts the result of running A* on a $50 \times 50$ grid, where cells that form part of an obstacle are colored red. The input map (Figure 1 (a)) shows the obstacles and free space. A* is run to find a path from lower right corner to upper left corner. The output map is shown in Figure 1 (b).

Consider the three cells X,Y,Z marked in the output of A* in Figure 1 (b) and the following inquiries on the optimal path discovered by A*:

- *Why was the cell Y not selected for the optimal path?* Given the output and logged internal states of the A* algorithm, we know that Y was considered as a candidate cell but discarded due to non-optimal cost.
- *Why was the cell X not selected for the optimal path?* If we logged the internal states of the A* algorithm, we would find that X was not even considered as a candidate and it never entered the priority queue of the A* algorithm. But this is not a useful explanation because a non-expert user cannot be

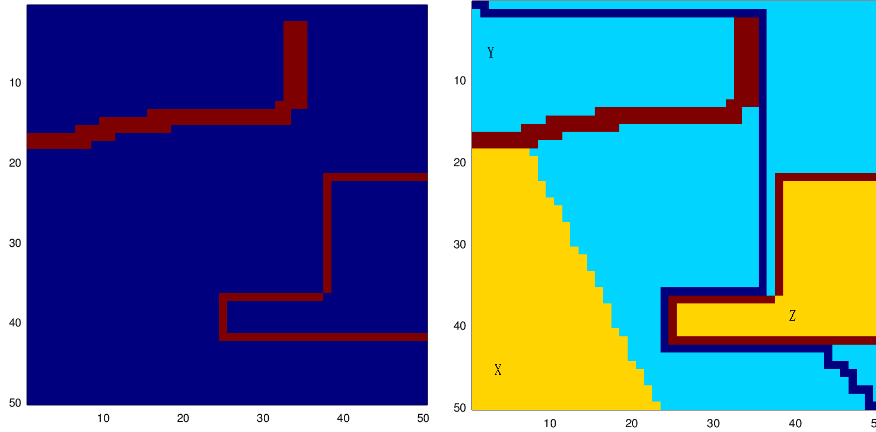


Fig. 1: (a) Input map to A* (b) Output showing final path and internal states of A*. Cells on the computed optimal path are colored dark blue. Cells which entered A*'s priority queue are colored light cyan, and those cells that never entered the queue are colored yellow.

expected to understand the concept of a priority queue, or the details of how A* works.

- *Why was the cell Z not selected for the optimal path?* The cell Z was also never inserted into the priority queue and hence, it was never a candidate to be selected on the optimal path similar to cell X. When responding to a user query about why X and Z were not selected in the optimal path, we cannot differentiate between the two even if all the internal decisions and states of the A* algorithm were logged. So, we cannot provide the intuitively expected explanation that Z is not reachable due to obstacles, while X is reachable but has higher cost than the cells that were considered.

This example scenario illustrates the need for new information to provide explanation in addition to the usual deduction by AI algorithm while solving the original decision making problem.

## 3 Problem Definition

A decision-making AI algorithm `Alg` can be modelled as a function that computes the values of output variables `out` given input variables `in`, that is,

$$\texttt{Alg} : \texttt{in} \rightarrow \texttt{out}$$

The outputs are the decision variables, while the inputs include the environment and system states as observed by the system through the perception pipeline. While the decision and state variables can be continuous and real valued, the inquiries and explanations are framed using predicates over these variables, such

as comparison of a variable to some threshold. These predicates can either be directly provided by the user or the developer of the AI system, or they can be automatically extracted from the implementation of the AI system by including predicates that appear in the control flow of the AI system. These must be predicates over the input and output variables, that is, `in` and `out`, which are understood by the users. Our approach exploits the sparsity of Boolean formula for learning the explanations and so, the vocabulary can include all possible predicates and variables that might be useful for explaining AI decisions. We propose methods to efficiently find relevant variables where these methods only depend logarithmically on the size of the vocabulary. This ensures that the definition of vocabulary can conveniently include all possible variables, and our approach can automatically find the relevant subset and synthesize the corresponding explanation.

We denote the vocabulary of atomic predicates used in the inquiry from the user and the provided explanation from the system by $\mathcal{V}$. We can separate the vocabulary $\mathcal{V}$ into two subsets: $\mathcal{V}_Q$ used to formulate the user inquiry and $\mathcal{V}_R$ used to provide explanations.

$$\mathcal{V}_Q = \{q_1, q_2, \dots q_m\}, \mathcal{V}_R = \{r_1, r_2, \dots r_n\} \text{ where } q_i, r_i : \texttt{in} \cup \texttt{out} \rightarrow \texttt{Bool}$$

Intuitively, $\mathcal{V}$ is the shared vocabulary that describes the interface of the AI algorithm and is understood by the human-user. For example, the inquiry vocabulary for a planning agent may include propositions denoting selection of a waypoint in the path, and the explanation vocabulary may include propositions denoting presence of obstacles on a map.

An *inquiry* $\phi_Q$ from the user is an observation about the output (decision) of the algorithm, and can be formulated as a Boolean combination of predicates in the vocabulary $\mathcal{V}_Q$. Hence, we can denote it as $\phi_Q(\mathcal{V}_Q)$ where the predicates in $\mathcal{V}_Q$ are over the set $\texttt{in} \cup \texttt{out}$, and the corresponding grammar is:

$$\phi_Q := \phi_Q \wedge \phi_Q \mid \phi_Q \vee \phi_Q \mid \neg \phi_Q \mid q_i \texttt{ where } q_i \in \mathcal{V}_Q$$

While conjunction and negation are sufficient to express any Boolean combination, we include disjunction and implication for succinctness of inquiries. Similarly, the *response* $\phi_R(\mathcal{V}_R)$ is a Boolean combination of the predicates in the vocabulary $\mathcal{V}_R$ where the predicates in $\mathcal{V}_R$ are over the set $\texttt{in} \cup \texttt{out}$, and the corresponding grammar is:

$$\phi_R := \phi_R \wedge \phi_R \mid \phi_R \vee \phi_R \mid \neg \phi_R \mid r_i \texttt{ where } r_i \in \mathcal{V}_R$$

**Definition 1.** *Given an AI algorithm* `Alg` *and an inquiry* $\phi_Q(\mathcal{V}_Q)$*,* $\phi_R(\mathcal{V}_R)$ *is a* necessary and sufficient explanation *when* $\phi_R(\mathcal{V}_R) \iff \phi_Q(\mathcal{V}_Q)$ *where* $\mathcal{V}_R, \mathcal{V}_Q$ *are predicates over* $\texttt{in} \cup \texttt{out}$ *as explained earlier, and* $\texttt{out} = \texttt{Alg}(\texttt{in})$*.* $\phi_R(\mathcal{V}_R)$ *is a* sufficient explanation *when* $\phi_R(\mathcal{V}_R) \Rightarrow \phi_Q(\mathcal{V}_Q)$*.*

If the algorithm $\texttt{out} = \texttt{Alg}(\texttt{in})$ could be modeled explicitly in appropriate logic, then the above definition could be used to generate explanations for a given inquiry using techniques such as satisfiability solving. However, such an explicit

modeling of these algorithms is currently outside the scope of existing logical deduction frameworks, and is impractical for large and complicated AI systems even from the standpoint of the associated modeling effort. The AI algorithm `Alg` is available as an executable function; hence, it can be used as an oracle that can provide an outputs for any given input. This motivates oracle-guided learning of the explanation from examples using the notion of confidence associated with it.

**Definition 2.** *Given an AI algorithm* `Alg` *and an inquiry* $\phi_Q(\mathcal{V}_Q)$, $\phi_R(\mathcal{V}_R)$ *is a* necessary and sufficient explanation *with probabilistic confidence* $\kappa$ *when* $Pr(\phi_R(\mathcal{V}_R) \iff \phi_Q(\mathcal{V}_Q)) \geq \kappa$, *where* $\mathcal{V}_R, \mathcal{V}_Q$ *are predicates over* $\texttt{in} \cup \texttt{out}$ *as explained earlier,* $\texttt{out} = \texttt{Alg}(\texttt{in})$ *and* $0 \leq \kappa \leq 1$. *The probability of satisfaction of* $\phi_R(\mathcal{V}_R) \iff \phi_Q(\mathcal{V}_Q)$ *is computed using uniform distribution over the variables in* $\mathcal{V}$. *This uniform distribution is not an assumption over the context in which an AI algorithm* `Alg` *is used. This uniform distribution is only used to estimate the probability of finding the correct explanation. Similarly,* $\phi_R(\texttt{in})$ *is a* sufficient explanation *with confidence* $\kappa$ *when* $Pr(\phi_R(\mathcal{V}_R) \Rightarrow \phi_Q(\mathcal{V}_Q)) \geq \kappa$.

The oracle used to learn the explanation uses the AI algorithm. It runs the AI algorithm on a given input $in_i$ to generate the decision output $out_i$, and then marks the input as a positive example if $\phi_Q(out_i)$ is true, that is, the inquiry property holds on the output. It marks the input as a negative example if $\phi_Q(out_i)$ is not true. We call this an *introspection oracle* which marks each input as either positive or negative.

**Definition 3.** *An* introspection oracle $\mathcal{O}_{\phi_Q,\texttt{Alg}}$ *for a given algorithm* `Alg` *and inquiry* $\phi_Q$ *takes an input* $in_i$ *and maps it to a positive or negative label, that is,* $\mathcal{O}_{\phi_Q,\texttt{Alg}} : \texttt{in} \rightarrow \{\oplus, \ominus\}$.

$$\mathcal{O}_{\phi_Q,\texttt{Alg}}(in_i) = \oplus \textit{ if } \phi_Q(\mathcal{V}_Q(out_i)) \textit{ and } \mathcal{O}_{\phi_Q,\texttt{Alg}}(in_i) = \ominus \textit{ if } \neg\phi_Q(\mathcal{V}_Q(out_i)), \textit{ where } out_i = \texttt{Alg}(in_i), \textit{ and } \mathcal{V}_Q(out_i) \textit{ is the evaluation of the predicates in } \mathcal{V}_Q \textit{ on } out_i$$

We now formally define the problem of learning Boolean formula with specified confidence $\kappa$ given an oracle that labels the examples.

**Definition 4.** *The problem of oracle-guided learning of Boolean formula from examples is to identify (with confidence* $\kappa$*) the target Boolean function* $\phi$ *over a set of atomic propositions* $\mathcal{V}$ *by querying an oracle* $\mathcal{O}$ *that labels each input* $in_i$ *(which is an assignment to all variables in* $\mathcal{V}$*) as positive or negative* $\{\oplus, \ominus\}$ *depending on whether* $\phi(in_i)$ *holds or not, respectively.*

We make the following observations which relates the problem of finding explanations for decisions made by AI algorithms to the problem of learning Boolean formula.

**Observation 1** *The problem of generating explanation* $\phi_R$ *for the AI algorithm* `Alg` *and an inquiry* $\phi_Q$ *is equivalent to the problem of oracle-guided learning of Boolean formula* $\phi_R$ *using oracle* $\mathcal{O}_{\phi_Q,\texttt{Alg}}$ *as described in Definition 4.*

$\phi[r_i]$ denotes the restriction of the Boolean formula $\phi$ by setting $r_i$ to `true` in $\phi$ and $\phi[\overline{r_i}]$ denotes the restriction of $\phi$ by setting $r_i$ to `false`. A predicate $r_i$ is in the support of the Boolean formula $\phi$, that is, $r_i \in \texttt{support}(\phi)$ if and only if $\phi[r_i] \neq \phi[\overline{r_i}]$.

**Observation 2** *The explanation $\phi_R$ over a vocabulary of atoms $\mathcal{V}_R$ for the AI algorithm* `Alg` *and a user inquiry $\phi_Q$ is a sparse Boolean formula, that is,* $|\texttt{support}(\phi_R)| << |\mathcal{V}_R|$.

These observations motivate the following problem definition for learning sparse Boolean formula.

**Definition 5.** *Boolean function $\phi$ is called k-sparse if $|\texttt{support}(\phi_R)| \leq k$. The problem of oracle-guided learning of k-sparse Boolean formula from examples is to identify (with confidence $\kappa$) the target k-sparse Boolean function $\phi$ over a set of atomic propositions $\mathcal{V}$ by querying an oracle $\mathcal{O}$ that labels each input $in_i$ (which is an assignment to all variables in $\mathcal{V}$) as positive or negative $\{\oplus, \ominus\}$ depending on whether $\phi(in_i)$ holds or not, respectively.*

The explanation of decisions made by an AI algorithm can be generated by solving the problem of oracle-guided learning of k-sparse Boolean formula.

## 4 Learning Sparse Boolean Formula

Recall that the vocabulary of explanation is $\mathcal{V}_R = \{r_1, r_2, \ldots, r_n\}$. Given any two inputs $in_1$ and $in_2$, we define the *difference* between them as follows: $\texttt{diff}(in_1, in_2) = \{i \mid r_i(in_1) \neq r_i(in_2)\}$. Next, we define a *distance* metric $d$ on inputs as the size of the difference set, that is, $d(in_1, in_2) = |\texttt{diff}(in_1, in_2)|$. $d(in_1, in_2)$ is the Hamming distance between the $n$-length vectors that record the evaluation of the atomic predicates $r_i$ in $\mathcal{V}_R$. We say that two inputs $in_1, in_2$ are *neighbours* if and only if $d(in_1, in_2) = 1$. We also define a partial order $\preceq$ on inputs as follows: $in_1 \preceq in_2$ *iff* $r_i(in_1) \Rightarrow r_i(in_2)$ *for all* $1 \leq i \leq n$.

Given an input $in$ and a set $J \subseteq \{1, 2, \ldots, n\}$, a *random $J$-preserving mutation* of $in$, denoted $\texttt{mutset}(in, J)$, is defined as: $\texttt{mutset}(in, J) = \{in' | in' \in$ `in` *and* $r_j(in') = r_j(in)$ *for all* $j \in J\}$.

A random walk `walk` over the Boolean hypercube starts with a random initial input $in^0$. The input at iteration $t+1$ is $in^{t+1} = \texttt{walk}(in^t)$ and $in^{t+1}$ is obtained by randomly sampling an index $j$ from $[1, n]$ with uniform probability and then flipping the variable at index $j$ of $in^t$ with probability $1/2$.

$$p(\texttt{walk}(in^t) = in \mid in^t) = \frac{1}{2} \text{ if } in = in^t = \frac{1}{2n} \text{ if } d(in, in^t) = 1$$

**Learning Based on Binary Search in Hamming Space:** We begin with two random inputs $in_1, in_2$ on which the oracle $\mathcal{O}_{\phi_Q,\texttt{Alg}}$ returns different labels, for example, it returns positive on $in_1$ and negative on $in_2$ without loss of generality. Finding such $in_1, in_2$ can be done by sampling the inputs and

querying the oracle until two inputs disagree on the outputs. The more samples we find without getting a pair that disagree on the label, the more likely it is that the Boolean formula being used by the oracle to label inputs is a constant (either `true` or `false`).

We can define $\texttt{sample}(\mathcal{O}_{\phi_Q,\texttt{Alg}}, in, J, \kappa)$ that samples $m = 2^k \ln(1/(1-\kappa))$ inputs from the set $\texttt{mutset}(in, J)$ and generates two inputs on which the oracle $\mathcal{O}_{\phi_Q,\texttt{Alg}}$ disagrees and produces different outputs. If it cannot find such a pair of inputs, it returns $\perp$. Lemma 1 justifies why the size $m$ of the samples is sufficient to achieve the probabilistic confidence $\kappa$.

**Lemma 1.** *If $m$ random samples $in_1, in_2, \ldots, in_m$ from $\texttt{mutset}(in, J)$ produce the same output as input 'in' for the oracle $\mathcal{O}_{\phi_Q,\texttt{Alg}}$ where $\phi_R$ is k-sparse, then the probability that all mutations $in' \in \texttt{mutset}(in, J)$ produce the same output (that is, the oracle is a constant function over $\texttt{mutset}(in, J)$) is at least $\kappa$, where $m = 2^k \ln(1/(1-\kappa))$.*

If $\texttt{sample}(\mathcal{O}_{\phi_Q,\texttt{Alg}}, in, J, \kappa)$ returns $\perp$, we have found the constant function. Otherwise, it returns two inputs $in_1, in_2$ on which the oracle disagrees. We find $J = \texttt{diff}(in_1, in_2) = \{i_1, i_2, \ldots, i_l\}$ on which the inputs differ with respect to the vocabulary $\mathcal{V}_R = \{r_1, r_2, \ldots, r_n\}$. We partition $J$ into two subsets $J_1 = \{i_1, i_2, \ldots, i_{\lfloor l/2 \rfloor}\}$ and $J_2 = \{i_{\lfloor l/2 \rfloor+1}, i_{\lfloor l/2 \rfloor+2}, \ldots, i_l\}$. The two sets $J_1$ and $J_2$ differ in size by at most 1. The set of inputs that are halfway between the two inputs w.r.t the Hamming distance metric $d$ defined earlier is given by the set $\texttt{bisect}(in_1, in_2)$ defined as:

$$\texttt{bisect}(in_1, in_2) = \{in' |\ \forall j \in J_1\ r_j(in')\ \texttt{iff}\ r_j(in_1), \forall j \in J_2\ r_j(in')\ \texttt{iff}\ r_j(in_2)\}$$

Satisfiability solvers can be used to generate an input $in'$ from $\texttt{bisect}(in_1, in_2)$. The oracle $\mathcal{O}_{\phi_Q,\texttt{Alg}}$ is run on $in'$ to produce the corresponding label. This label will match either the label for the input $in_1$ or that of the input $in_2$. We discard the input whose label matches $in'$ to produce the next pair of inputs, that is,

$$\texttt{introspect}(in_1, in_2) = \begin{cases} (in_1, in') & \textit{if } \mathcal{O}_{\phi_Q,\texttt{Alg}}(in') \neq \mathcal{O}_{\phi_Q,\texttt{Alg}}(in_2) \\ (in', in_2) & \textit{if } \mathcal{O}_{\phi_Q,\texttt{Alg}}(in') \neq \mathcal{O}_{\phi_Q,\texttt{Alg}}(in_1) \end{cases}$$
$$\textit{where } in' \in \texttt{bisect}(in_1, in_2)$$

Starting from an initial pair of inputs on which $\mathcal{O}_{\phi_Q,\texttt{Alg}}$ produces different labels, we repeat the above process, considering a new pair of inputs at each iteration until we have two inputs $in_1, in_2$ that are neighbours, with $\texttt{diff}(in_1, in_2, S) = \{j\}$. At this point, we can conclude that $r_j \in \mathcal{V}_R$ is in the support of the explanation $\phi_R$ because changing the assignment of $r_j$ changes the response of the oracle.

We add $r_j$ to the set of variables $\mathcal{V}_{\phi_R}$. We repeat the above process twice to find the next relevant variable to add to the support set by setting $r_j$ to first true and then false. This introspective search for variables in the support set $\mathcal{V}_{\phi_R}$ is repeated till we cannot find a pair of inputs $in_1, in_2$ on which the oracle produces different outputs. This continues till the $\texttt{sample}(\mathcal{O}_{\phi_Q,\texttt{Alg}}, in, J, \kappa)$ returns $\perp$ and

we have found all the relevant variables with probabilistic confidence $\kappa$. This overall algorithm for finding the support of the explanations $\phi_R$ with probability $\kappa$ is presented in Algorithm 1 using the oracle $\mathcal{O}_{\phi_Q,\mathtt{Alg}}$.

**Algorithm 1** Computation of $\mathcal{V}_{\phi_R}$ using binary search in Hamming space: $\mathtt{getSupport}(\mathcal{O}_{\phi_Q,\mathtt{Alg}}, in, J, \kappa)$

**if** $\mathtt{sample}(\mathcal{O}_{\phi_Q,\mathtt{Alg}}, in, J, \kappa) = \perp$ **then**
  // The $J$-restricted Boolean formula is constant function with probability $\kappa$.
  **return** $\{\}$
**else**
  $(in_1, in_2) \Leftarrow \mathtt{sample}(\mathcal{O}_{\phi_Q,\mathtt{Alg}}, in, J, \kappa)$
  **while** $|\mathtt{diff}(in_1, in_2)| \neq 1$ **do**
    $in_1, in_2 \Leftarrow \mathtt{introspect}(in_1, in_2)$
  $r_i$ is the singleton element in $\mathtt{diff}(in_1, in_2)$
  $J \Leftarrow J \cup \{i\}$
  **return** $\{r_i\} \cup \mathtt{getSupport}(\mathcal{O}_{\phi_Q,\mathtt{Alg}}, in_1, J, \kappa) \cup \mathtt{getSupport}(\mathcal{O}_{\phi_Q,\mathtt{Alg}}, in_2, J, \kappa)$

**Learning Based on Random Walk in Boolean Hypercube:** The algorithm for finding relevant variables using random walk in Boolean hypercube starts with a random initial input $in^0$. This input is a vertex in the n-dimensional Boolean hypercube of the $\mathcal{V}_R$ variables. The algorithm performs a random walk from this vertex, proceeding in iteration $t$ from the vertex $in^t$ to the vertex $\mathtt{walk}(in^t) = in^{t+1}$. As defined earlier, $\mathtt{walk}$ probabilistically either chooses to stay at the same vertex, or to move to a vertex which is 1 Hamming distance away where the differing variable in $\mathcal{V}_R$ is uniformly selected from the $n$ variables. This random walk is performed for at most $L(k', \kappa) = 2^{k'}(2k'^2)(1 + \log k') \log(k'/\kappa)$ steps when searching for $k'$ relevant variables. Using results from mixing time, we will later show that random walk of this length must either find two neighboring vertices on which the oracle produces different labels or the oracle computes a constant function with probabilistic confidence $\kappa$. When we find neighboring vertices with different labels, we can find the single variable on which these two inputs differ and clearly, this variable is a relevant variable since changing its assignment changes the output of the oracle. This is repeated to find all the relevant variables. The recursive Algorithm 2 is initially called as $\mathtt{getSupportRW}(\mathcal{O}_{\phi_Q,\mathtt{Alg}}, in_0, \{\}, \kappa, k')$ with random initial input $in^0$ and an empty set as the so far found variables, $J = \{\}$. For learning sparse Boolean formula, we may not know the size of the support set and so, Algorithm 2 is repeated with $k' = 1, 2, \ldots, k$ till we can't find more relevant variables. We analyze the complexity of this algorithm in Section 4, and show that the number of examples required to find all the relevant variables is logarithmic in the size of the vocabulary $\mathcal{V}_R$. Lemma 2 establish that $L(k', \kappa) = 2^{k'}(2k'^2)(1 + \log k') \log(n/(1 - \kappa))$ is a sufficiently long random walk that the function must be constant if all the vertices have the same label.

**Algorithm 2** Computation of $\mathcal{V}_{\phi_R}$ using random walk over Boolean hypercube: $\texttt{getSupportRW}(\mathcal{O}_{\phi_Q,\texttt{Alg}}, in, J, \kappa, k')$

```
t = 0
while t ≤ L(k', κ) do
   in^{t+1} = walk(in^t)
   if O_{φQ,Alg}(in^{t+1}) ≠ O_{φQ,Alg}(in^t) then
      r_i is the singleton element in diff(in^{t+1}, in^t)
      J ⇐ J ∪ {i}
      return getSupportRW(O_{φQ,Alg}, in, J, κ, k')
   t = t + 1
return J
```

**Lemma 2.** *The expected mixing time of the uniform random walk in Algorithm 2 is smaller than $k(1 + \log k)$ where $k = |\mathcal{V}_R|$ is the number of relevant variables, that is, the expected number of steps starting from any vertex in the hypercube, after which sampling is identical to uniform sampling from all the $2^n$ possible assignments is less than $k(1 + \log k)$. This is a formulation of the well known "coupon collector's problem" [1].*

**Learning Boolean formula $\phi_R$ with given support** We adopt a technique based on the use of distinguishing inputs proposed by us in [4], and described in [6]. The theoretical analysis and empirical evaluation of the presented approach is described in [6] and [7]. For brevity, we state the main theoretical result here:

**Theorem 1.** *The overall algorithm to generate k-sparse explanation $\phi_R$ for a given query $\phi_Q$ takes $O(2^{2k} \ \ln(n/(1-\kappa)))$ queries to the oracle, that is, the number of examples needed to learn the Boolean formula grows logarithmically with the size of the vocabulary $n$.*

## 5 Conclusion and Future Work

We present an algorithm to first find the support of any sparse Boolean formula using two alternative methods, followed by a formal synthesis approach to learn the target formula from examples. We demonstrate how this method can be used to learn Boolean formulae corresponding to the explanation of decisions made by an AI algorithm. We identify two dimensions along which our work can be extended. First, the approach needs to be extended to non deterministic AI systems by considering learning Boolean formula from a noisy oracle. Further, we are working on combining this model-agnostic method for generating explanations by interrogating the model with white-box methods for analyzing neural networks [2]. This work is a first step towards using formal methods, particularly, formal synthesis to aid in interpretability of artificial intelligence by automatically generating explanations of decisions made by AI algorithms.

## Acknowledgement

The author acknowledges support from the US ARL Cooperative Agreement W911NF-17-2-0196 on Internet of Battlefield Things (IoBT) and National Science Foundation(NSF) #1750009 and #1740079.

## References

1. A. Boneh and M. Hofri. The coupon-collector problem revisiteda survey of engineering problems and computational methods. *Stochastic Models*, 13(1):39–66, 1997.
2. S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari. Output range analysis for deep neural networks. *arXiv preprint arXiv:1709.09130*, 2017.
3. A. Ehrenfeucht, D. Haussler, M. Kearns, and L. Valiant. A general lower bound on the number of examples needed for learning. *Information and Computation*, 82(3):247 – 261, 1989.
4. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224. IEEE, 2010.
5. S. Jha, V. Raman, A. Pinto, T. Sahai, and M. Francis. On learning sparse boolean formulae for explaining AI decisions. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 99–114, 2017.
6. S. Jha, V. Raman, A. Pinto, T. Sahai, and M. Francis. On learning sparse boolean formulae for explaining ai decisions. In *NASA Formal Methods Symposium*, pages 99–114. Springer, 2017.
7. S. Jha, T. Sahai, V. Raman, A. Pinto, and M. Francis. Explaining ai decisions using efficient methods for learning sparse boolean formulae. *Journal of Automated Reasoning*, pages 1–21, 2018.
8. S. Jha and S. A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica, Special Issue on Synthesis*, 2016.
9. S. Jha, S. A. Seshia, and A. Tiwari. Synthesis of optimal switching logic for hybrid systems. In *EMSOFT*, pages 107–116. ACM, 2011.
10. M. Kearns, M. Li, and L. Valiant. Learning boolean formulas. *J. ACM*, 41(6):1298–1328, Nov. 1994.
11. M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM (JACM)*, 41(1):67–95, 1994.
12. S. M. LaValle. *Planning algorithms.* Cambridge university press, 2006.