Towards Partial (and Useful) Model Identification for Model-Based Diagnosis

Vladimir Sadov [1], **Eliahu Khalastchi** [1], **Meir Kalech** [2], **Gal A. Kaminka** [1]

[1] *Department of Computer Science, Bar-Ilan University, Israel*
iholaynen@gmail.com, eli.kh81@gmail.com, galk@cs.biu.ac.il
[2] *Department of Information Systems Engineering, Ben-Gurion University, Israel*
kalech@bgu.ac.il

ABSTRACT

A fundamental requirement for model-based diagnosis (MBD) is the existence of a model of the diagnosed system. Based on the model, MBD algorithms are able to diagnose the faulty components. Unfortunately, a model is not always available. While it is possible in principle to infer a partial model by repeated trials, performing such trials is time and resource costly for any practical system. Therefore minimizing the number of trials is important. In this paper, we propose three algorithms for learning the model: two algorithms are Depth-first search (DFS) based and one algorithm utilizes a binary search algorithm. We evaluate the algorithms theoretically and empirically through thousands of tests and show that one of the DFS-based algorithm scales well and the binary search algorithm is efficient for small systems. Finally, we successfully demonstrate the algorithms on a model of the NAO robot (20 components) to show its capability in real world domain.

1 INTRODUCTION

Many complex systems are capable of performing different actions. For instance, a humanoid robot can walk, manipulate physical objects, examine the world using sensors, adjust the parameters of these sensors, etc. Each of these actions requires activating different subsets of the robot's components. For instance, walking requires the activation of leg actuators (and sometimes arms), but does not involve the head. Some actions can be performed in more than one way: a robot can rotate its head to sense object laying out of direct range of its head camera but it can also rotate its entire body to achieve the same goal (Daigle *et al.*, 2006).

We refer to the description of the component subsets that are involved in the execution of an action as a *model* in terms of model-based diagnosis (MBD) (Steinbauer *et al.*, 2009). Given such a model, MBD algorithms can diagnose the faulty components utilizing reasoning methods (de Kleer and Williams, 1987), by comparing the observed system behavior and the expected behavior by the model.

Unfortunately, a model is not always available. For instance, the topology of the communication network is not always known (e.g., ad-hoc networks), and even if known, does not exactly match the actual routes used by the communicated data (e.g. IP packets). Indeed, even if a model is known, it may differ from the system as it executes, e.g., if the system uses learning to adapt itself. For instance, a robot can realize locomotion by learning which method to apply: crawling, walking, running, etc. Each of these approaches requires a different subset of active robot components to succeed, which are unknown before the learning occurs.

For each action available to the given system there is at least one minimal subset of fully functional system components that is sufficient for the operational of the action. In this paper we define a partial model of the system as a group of all such subsets. When no information about the model is available, it is possible to infer a partial model of the system, by repeated trials. Assuming that it is possible to control the activity of the systems' components, we can activate a subset of the components, try to perform an action and then analyze the results. We can repeat this iteratively for various actions to learn a partial model; It would describe which components are involved in the success of an action, but not the relation between the components.

Naturally, such an approach leads to massive failure injection into the studied system: the activation of a limited subset of required components will, of course, lead to intended failures in most of the cases. This limits the usage of the proposed approach. For instance, heavy industrial systems, where each failure can damage the system itself or cause environmental disaster. However, there are other systems where such an approach is applicable. For example, virtual systems or physical system which explicitly secured for evaluation like a humanoid robot on rope can securely learn locomotion. However, performing such trials is usually time and resource costly for any practical system. Therefore minimizing the number of trials is important.

In this paper, we propose three algorithms that infer a partial model by repeated trials, while minimizing the number of trials. Specifically, given an action, they find all the possible minimal sets of components that must cause an activation of that action. The first, DFS-based, algorithm is exponential both in the number of trials as well as in runtime. Under the assumption that the number of possible sets of components that activate an action is bounded by a constant, we propose an improvement to the first algorithm which results in a second algorithm, which is polynomial in the number of trials and runtime. The last algorithm is based on a binary search approach. This algorithm guarantees a polynomial number of trials independently of the number of sets of components that activate the action. However, computing these trials is exponential in the number of components.

Empirical evaluation shows that the improved version of the DFS algorithm finds the minimal sets of components in reasonable number of trials, even in large scale systems. The binary search algorithm solves the problem efficiently only for small systems. In addition, we ran several actions using a simulation of a NAO humanoid robot manufactured by Aldebaran Robotics, and showed that the improved DFS algorithm and the binary search succeeded in inferring the components that caused the actions in reasonable number of trials.

2 RELATED WORK

The question we present in this paper is similar to the problem of finding minimal conflict sets in MBD. A conflict set is a set of components such that, assuming they are normal, the system is not consistent with the observation. Similarly, in our problem we try to recognize a set of components that must be activated in order to perform a given action. In both problems, a set of all the components is both a conflict set and an activating set. On the other hand, the empty set is neither a conflict set nor an activating set. In addition, in MBD, any superset of a conflict is a conflict set too; the same is in our problem, any superset of an activating set will operate the action too. The main difference between these problems is that while in MBD we assume that the system description is known, and thus we can recognize the conflict sets (e.g., (de Kleer and Williams, 1987; Williams and Ragno, 2007). In our problem, the basic assumption is that the system description is not given, thus we propose to use a trial and error approach to address this challenge.

The problem is also related to computer network topology discovery (Farkas *et al.*, 2008). Several approaches have been studied: Address Forwarding Tables (AFT) collected from switches (O'Hallaron *et al.*, 2001), this method needs that all the network switches will support AFT forwarding and demands broadcasting of the routing data by every switch. Another approach is based on sending packets from the end-points and observing where they are delivered (Black *et al.*, 2004). This method requires that the network end-points will be equipped with monitoring software. Our algorithm, when applied to the networking, is less demanding in terms of required implemented protocols on network switches. While not proposing a protocol specific implementation for the topology discovery domain, we think that our algorithm in conjunction with domain-specific modifications, can potentially serve network topology discovery applications.

Another related problem is known as Frequent Itemset Mining (FIM) problem: given a set of items and database of *transactions*, the goal is to find all such subsets of items which appear more than t times in the database (Han and Kamber, 2001). FIM appears to be similar to our problem: the items in FIM are the components in our problem, the satisfaction of the threshold t is the activation of an action by the subset of components and the database queries can be replaced with the system trials introduced in this paper. The power set lattice of the items items and components is the same. However, a closer examination shows, that FIM DB queries and our system trials are very different actions. While FIM algorithms are performing bottom-up search, starting from the trivial subsets contained in transactions (so the database query is positive), our system trials will fail on every state set which does not contain all the components necessary to activate the action. Thus making bottom-up approach is infeasible. on the other hand, a top-down approach, when we start from the fully activated system does not allow to effectively decide on the next DB-query. A deterministic approach leads to full exploration of current subset $O(n)$ nodes making the search exponential. Less exhaustive search, leaves a probability to miss target subsets as discussed furthermore.

3 PARTIAL MODEL IDENTIFICATION

This section presents the basic objects used for model identification. A system is the most fundamental object. A *system* C is a set of n components $\{c_1, \ldots, c_n\}$.

Each component $c_i \in C$ can be enabled and disabled externally. When c_i is enabled it is fully operational and its output is defined to be correct. When c_i is disabled, it is not activated.

Definition 1 (enable)**.** *The predicate $\omega(c_i)$ is true if component c_i is* enabled*. We define a* state set $S \subseteq C$ *to represent a set of enabled components. In first order logic* $\bigwedge_{s_i \in S} \omega(s_i) \bigwedge_{s_j \in C \setminus S} \neg\omega(s_j)$.

The system is capable to perform several actions defined by the following set: $A = \{a_1, \ldots, a_m\}$. A certain action a_i is activated if a corresponding subset of the components is enabled.

Definition 2 (activating set)**.** *The predicate $\Phi(S, a_i)$ is true if a given state set S activates the action a_i.*

For instance, assume a humanoid robot with four servos $C = \{c_1, c_2, c_3, c_4\}$ such that two servos are controlling its hands (c_1 for the right hand and c_2 for the left), and the other two are controlling its legs (c_3 for the right leg and c_4 for the left). To operate a kick action (a_1), the robot could use his right leg: $\Phi(\{c_3\}, a_1)$.

The goal of this paper is to find the *minimal* state set that activates a given action. For this, we define here the minimality assumption: if a state set S does not activate a_i then all of its subsets will also fail. Formally:

Assumption 1. *Given sets $S_1 \in C$, $S_2 \in C$ such that $S_2 \subseteq S_1$, $\neg\Phi(S_1, a_i) \Rightarrow \neg\Phi(S_2, a_i)$.*

The contraposition of this assumption is a direct logical corollary of this statement.

Corollary 1. *Given sets $S_1 \in C$, $S_2 \in C$ such that $S_2 \subseteq S_1$, $\Phi(S_2, a_i) \Rightarrow \Phi(S_1, a_i)$.*

This assumption is reasonable in many physical systems, since a component that does not cause the activation of an action, is not expected to cause the action to fail. For instance, a humanoid robot will succeed kicking a ball with his right leg, even in case that all his other components are enabled.

The problem of model identification is defined as follow:

Definition 3 (Partial model identification problem)**.** *Given the system C and the action a_i, find a minimal state set $S \subseteq C$ such that $\Phi(S, a_i) = true$.*

Since it is possible that an action can be activated by more than one state set, we will extend the model identification problem to find *all* those state sets. For instance, the robot can kick either by his right leg or by his left leg: $\Phi(\{c_1\}, a_1) \vee \Phi(\{c_2\}, a_1)$.

4 MODEL IDENTIFICATION ALGORITHMS

To address the model identification problem we adopt a trial and error approach. This approach proceeds iteratively as follow:

1. Generate a state set S.
2. Check if $\phi(S, a_i) = True$ by trying to execute the required action a_i when only the components of state set S are enabled.
3. *if* $\Phi(S, a_i) = true$ then store the result and proceed to find the next result *else* goto 1.

A key question is how to choose the next state set S. A naive, brute-force approach will pass through all 2^n possible subsets of C to find those that activate the action. This naive approach is exponential both in terms of run-time and the number of trials. In the next subsections we will propose two approaches to reduce the complexity of traveling through the possible trials: (1) a depth first search with branch and bound (DFBnB) approach (Papadimitriou and Steiglitz, 1982), and (2) a binary search approach.

4.1 A DFBnB Approach

In this section, we present a DFBnB approach for the trials generation order. Usually, in DFBnB, the search proceeds in a DFBnB manner while bounded by a cost function (Papadimitriou and Steiglitz, 1982). In our algorithm the bound is determined by the set of states that have been proven to not activate the action. The search starts with a root trial which includes all the components (C) since, based on Assumption 1, it must activate a_i, yet it is not necessarily the minimal set. Then, in a DFBnB manner, we go over all the states except those that were bounded since they are subsets of states that have been proven to not activate the action.

To implement the DFBnB algorithm we manage two lists, a close list ($CloseList$) which contains the roots of all the explored state sets sub-trees and a results list ($Results$) that contains only those minimal state sets that do activate the action. To guarantee maximality (in case of CloseList) and minimality (in case of Results list) of state sets, when inserting a new state to one of these lists we use the next routines:

Subroutine 1. *InsertSuperset(CloseList,S) - Removes all the subsets of S from the CloseList; If no superset of S is in the CloseList, inserts S to the CloseList.*

Subroutine 2. *InsertSubset(Results,S) - Removes all the supersets of S from Results; If no subset of S is in Results, inserts S to Results.*

Finally, since a state can be reached through different descending states, we define a boolean property $visit$ for a state set S, representing whether it has already been visited or not.

Algorithm 1 DFBnB
(**input**: system – C
input: action – a_i
output: list of results – $Results$)

```
1: CloseList ← ∅
2: Result ← ∅
3: Stack.push(C)
4: while Stack is not empty do
5:     S ← Stack.pop()
6:     if not S.visit then
7:         S.visit = true
8:         if Φ(S, a_i) = true then
9:             InsertSubset(Results, S)
10:            for all S' such that child(S', S) do
11:                if S' is not a subset of a state in the CloseList then
12:                    Stack.push(S')
13:                end if
14:            end for
15:        else
16:            InsertSuperset(CloseList, S)
17:        end if
18:    end if
19: end while
20: return Results
```

Algorithm 1 presents the DFBnB-based algorithm. When generating a new state set S, we try to activate a_i by S: $\Phi(S, a_i)$ (line 8). If it returns false, then we insert S to the $CloseList$ (15–16), otherwise, we insert it to $Result$ (line 9) and push all its children ($child(S', S) \equiv S' \subseteq S \wedge |S'| = |S| - 1$) that are not subsets of a state in the $CloseList$, into the $Stack$ (lines 10–14).

Figure 1 demonstrates the DFBnB traveling where $C = \{c_1, c_2, c_3, c_4\}$ and $\Phi(\{c_4\}, a_i) = true$. The order of the trial states generation is marked. The following run will be generated (each list entry represents a distinct DFBnB descending chain). $\times$ represents an insertion to the $CloseList$ and $\surd$ represents an insertion to $Results$:

1. $\{c_1, c_2, c_3, c_4\} \rightarrow \{c_1 c_2, c_3\} \rightarrow \times$
2. (backtracking to $\{c_1, c_2, c_3, c_4\}$)$\rightarrow$ $\{c_1, c_2, c_4\} \rightarrow \{c_1, c_4\} \rightarrow \{c_4\} \rightarrow \surd$
3. (backtracking to $\{c_1, c_2, c_4\}$)$\rightarrow$ $\{c_2, c_4\} \rightarrow \times$
4. (backtracking to $\{c_1, c_2, c_3, c_4\}$)$\rightarrow$ $\{c_2, c_3, c_4\} \rightarrow \{c_3, c_4\} \rightarrow \times$

Although pruning by the close list, the worst-case complexity of the DFBnB-based algorithm is still exponential in the number of components (2^n). This will happen, for instance, in case that $\Phi(\{c_1\}, a_i) = true$.

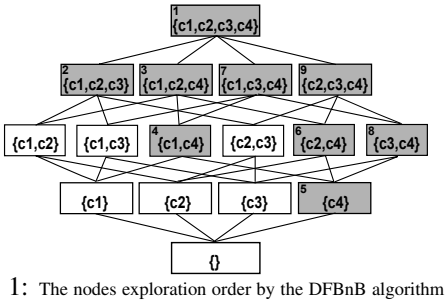

Figure 1: The nodes exploration order by the DFBnB algorithm for C : $\{c_1, c_2, c_3, c_4\}$ action a_i is activated by $\Phi(\{c_4\}, a_i)$

4.2 Polynomial DFBnB

In the DFBnB algorithm, when generating a new state, we prune all its children that are subsets of a state in the $CloseList$. However, if a child is a superset of a state in the $Results$ list, we could not prune it since it may contain another state, that has not been generated yet, and does activate a_i. For instance, generating trial state 7 in Figure 1 is necessary, since although it contains c_4 and thus $\Phi(\{c_1, c_3, c_4\}, a_i) = true$, it could also be true due to other subsets as $\{c_1, c_3\}$ that have not been generated yet.

DFBnB can be significantly improved by pruning also subsets of the $Results$ list. For this aim, before pushing a trial state to the stack, we check first whether it potentially contains states that should be in $Results$ but have not been generated yet, as shown in Algorithm 2. The improved DFBnB search proceeds as presented in Algorithm 1, but instead of checking the new state children only against the $CloseList$ (as in line 12), we check them also considering the $Results$, as shown in Algorithm 2.

Algorithm 2 will be invoked, substituting line 11 in Algorithm 1. It obtains four parameters: S, a candidate child state that we check whether to add as a trial to the stack or not; L, a list of states in $Results$ that are subsets of S ($L = \{R | R \in Results \wedge R \subseteq S\}$); and $Stack$ and $CloseList$, the same as in the DFBnB algorithm. Algorithm 2 proceeds recursively. At the beginning, the algorithm checks whether L is empty, if it is empty then S is not a superset of any member of $Results$. In such cases, if it is not in the $CloseList$, it is added to $Stack$ as a new trial state (lines 2–4). In lines 7–9 we go through the children of S ($child(S', S) \equiv S' \subseteq S \wedge |S'| = |S| - 1$) that are not superset of R (and thus they could be supersets of other results), and recursively call Algorithm 2 with each child and the remaining of L.

Let us demonstrate Algorithm 2. Assume $S = \{c_1, c_3, c_4\}$ (state 7 in Figure 1). This state is a superset of c_4 and thus we call Algorithm 2 with $S = \{c_1, c_3, c_4\}$ and $L = \{\{c_4\}\}$. L is not empty and thus in line 8 we call again to the algorithm with the child state of S $S' = \{c_1, c_3\}$, and the remaining of L. In the next call, L is an empty set, and since $\{c_1, c_3\}$ is a subset of a state in the $CloseList$ $\{c_1, c_2, c_3\}$, we do not generate it.

Figure 2 presents the order of the trial states of Algorithm 2. We can see that the pruning by the $Results$ list, reduces the number of trials versus Figure 1.

To prove completeness of Algorithm 2 we first prove the following lemma:

Algorithm 2 Prune_DFBnB
(**input**: state – S
input: sublist of $Results$ – L
input: stack – $Stack$
input: close list – $CloseList$)

```
1: if L = ∅ and S is not a subset of a state in the CloseList then
2:     Stack.push(S)
3: else
4:     R ← L.first
5:     if R ⊆ S then
6:         for all S' such that child(S', S) ∧ R ⊈ S' do
7:             Prune_DFBnB(S', L \ {R}, Stack, CloseList)
8:         end for
9:     else
10:        Prune_DFBnB(S, L \ {R}, Stack, CloseList)
11:    end if
12: end if
```

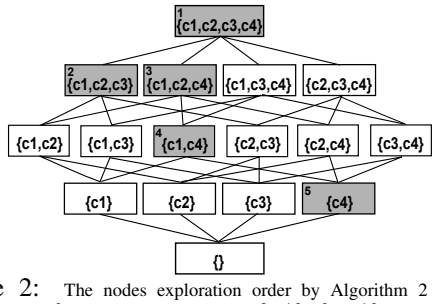

Figure 2: The nodes exploration order by Algorithm 2 for C : $\{c_1, c_2, c_3, c_4\}$ action a_i is activated by $\{\Phi(\{c_4\}, a_i)\}$

Lemma 1. *Given S and L and let $R^\star \subseteq S$ such that $\Phi(R^\star, a_i) = true$ and $R^\star \notin L$. Then, after invoking $Prune\_DFBnB(S, L, Stack, CloseList)$ $\exists S^\star \in Stack$ such that $R^\star \subseteq S^\star$*

Proof. If $L = \emptyset$ then, since $R^\star \subseteq S$ and $\Phi(R^\star, a_i) = true$, $S \notin CloseList$ and then S is pushed to $Stack$ (lines 1–3). If, however, $L \neq \emptyset$, let $R_1 \in L$ be the first member of L, if $R_1 \subseteq S$ then $Prune\_DFBnB(S \setminus \{c_i\}, L \setminus \{R_1\}, Stack, CloseList)$ is executed for each $c_i \in R_1$. Necessarily $\exists c_i$ such that $S \setminus \{c_i\} \subseteq R^\star$. Because, $R^\star \notin L$ and $R_1 \in L$ and $R^\star \subseteq S$, then $\exists c_i \in R_1 \wedge c_i \notin R^\star$. So, the next recursion level of $Prune\_DFBnB$ is executed on a superset of $R^\star$ (lines 9–11). If $R_1 \not\subseteq S$ then $Prune\_DFBnB(S, L \setminus \{R_1\}, Stack, CloseList)$ is executed (lines 6-8). Note, that $R^\star \subseteq S$. Again, the next recursion level of $Prune\_DFBnB$ is executed on a superset of $R^\star$. Recursively, $Stack$ generated by $Prune\_DFBnB$ will include $S^\star \supseteq R^\star$, by meeting the stop condition $L = \emptyset$. □

Now we can prove the algorithm's completeness:
Theorem 1. *$\forall S$ such that $\Phi(S, a_i) = true$ and there is no $S' \subset S$ such that $\Phi(S', a_i) = true$ then $S \in Results$.*

Proof. Given $R^\star \subset S$ such that $\Phi(R^\star, a_i) = true$ and $R^\star \notin L$, once S is popped out of $Stack$, according to Lemma 1, $Prune\_DFBnB$ will push $S^\star \subset S$ to $Stack$ such that $R^\star \subseteq S^\star$. The recursion will repeat itself in every decreasing size of $S^\star$ such that $R^\star \subseteq S^\star$ until $S^\star = R^\star$ and therefore $R^\star$ will finally be pushed to $|Results|$. Based on this consequence, since that in the first iteration of Algorithm 1: (1) $S = C$, (2) $\forall R$

such that $\Phi(R^\star, a_i) = true$: $R \subseteq C$ and (3) $L = \emptyset$, then $\forall R$: R is pushed to $|Results|$. □

The soundness of the algorithm is presented in the next proof:

Theorem 2. *$\forall S \in Results$, $\Phi(S, a_i) = true$ and there is no $S' \subset S$ such that $\Phi(S', a_i) = true$.*

Proof. Based on lines 8–9 in Algorithm 1, a state that has been added to the $Results$ list affirms $\Phi(S, a_i)$=true. Subroutine 2 (Section 4.1) guarantees minimality. □

Finally, we analyze the complexity of the algorithm:

Theorem 3. *The worst case complexity of the number of trial states and the runtime is bounded by $O(|Results|n^{|Results|+1})$.*

Proof. Given a state $S \in 2^C$ where $\Phi(S, a_i) = true$ and given $R \subseteq S$ where R is a minimal set such as $\Phi(R, a_i) = true$ but $R \notin Result$ yet. We will prove that $n^{|Results|+1}$ trial states are required to add R to $Result$. Since $\Phi(S, a_i) = true$, Algorithm 2 is invoked for each child S' of S. In line 6, Algorithm 2 goes over through the children of S' that are not supersets of R. There are at most $|R| - 1$ such children, where R is bounded by the number of components (n). For each one of these children we recursively call Algorithm 2 with the next R in L (and again we go through $|R| - 1$ children). This recursion will proceed as long as the list L is not empty, while L is bounded by $|Results|$. Thus, in the worst case Algorithm 2 will be invoked $n^{|Results|}$ times recursively. In each time it is invoked, it will add at most one trial state to the stack. Since Algorithm 2 is invoked by Algorithm 1 for each one of the children of S, the worst case complexity provides $O(n^{|Results|+1})$ trial states until R will be added to the stack. In the first iteration, only C is in the stack and $|Result|$ is empty, thus any R that affirms $\Phi(R, a_i) = true$ is $R \subseteq S$, and therefore it requires at most $|Results|n^{|Results|+1}$ trial states to be added to $Results$. The worst case complexity of the required number of trials to explore *all* the $Results$ is $O(|Results|n^{|Results|+1})$. The number of iterations of the $while$ loop of DFBnB algorithm is bounded by the total number of trials; thus the runtime complexity is $O(|Results|n^{|Results|+1})$. □

Since the complexity of the algorithm is exponential in the size of $Results$, we can relax the complexity by proposing the next assumption:

Assumption 2. *The number of minimal state sets that activate the action a_i is constant, that means $|Result| = O(1)$.*

This is a reasonable assumption in many physical systems. For instance, a humanoid robot can operate a pushing action either by its right hand or by its left hand. A kick action can be achieved by one of its legs. Thus, in many cases, the total size of the $Results$ list will be very small.

Based on this assumption, the complexity of the number of trial states as well as the runtime complexity will be polynomial.

5 A BINARY SEARCH APPROACH

The complexity of the number of trials can be reduced by using a binary search approach. In this approach in each iteration we generate a new state S, where its size is the average of (1) a currently known state S' that activates a_i ($\Phi(S', a_i) = true$) and (2) a currently known state S" that does not activate a_i ($\Phi(S", a_i) = false$). This binary search prunes the state space efficiently, and thus generates only a polynomial number of trial states. However, the computational time for choosing the next trial is exponential.

The binary search is presented in Algorithm 3. The algorithm manages two stacks: $StackUP$ and $StackLOW$. In the first iteration, the algorithm initializes StackUP (line 3)with C (this set must affirm $\Phi(C, a_i)$) and StackLOW (line 4) with the empty set (this set must affirm $\neg\Phi(\emptyset, a_i)$). The algorithm iteratively continues as long as the StackUP is not empty. This will happen only when there are no possible trial states anymore. In each iteration it tries to find a trial state between the $root$ and the $leaf$ states. This state must satisfy the conditions that appear in lines 9–12. Then we perform the founded trial state S and based on the response we decide to continue the search with S either as a new $root$ (line 15) or as a new $leaf$ (line 18). Correspondingly, we also decide to add S either to $Results$ or to the $CloseList$. If no trial state has been found, we pop out the $leaf$ from $StackLOW$ – in case that the $leaf$ exists in the $CloseList$ (lines 22–23), otherwise we are guaranteed that we cover the power set of the $root$ and then we continue with the next $root$ in $StackUP$ (line 24).

Algorithm 3 Binary_Search
(**input**: system – C
input: action – a_i
output: list of results – $Results$)

```
1: CloseList ← ∅
2: Result ← ∅
3: StackUP.push(C)
4: StackLOW.push(∅)
5: while StackUP is not empty do
6:     root ← StackUP.top()
7:     leaf ← StackLOW.top()
8:     choose S such that:
9:     1. S ⊆ root and S ⊇ leaf AND
10:    2. S is not a subset of any state in the CloseList AND
11:    3. S is not a superset of any state in Results AND
12:    4. |S| is close as possible to (|root|+|leaf|)/2
13:    if Φ(S, a_i) then
14:        InsertSubset(Results, S)
15:        StackUP.push(S)
16:    else
17:        InsertSuperset(CloseList, S)
18:        StackLOW.push(S)
19:    end if
20:    if no S has been found then
21:        if leaf ∈ CloseList then
22:            StackLOW.pop()
23:        else
24:            StackUP.pop()
25:        end if
26:    end if
27: end while
28: return Results
```

Figure 3 demonstrates a run of the algorithm. In the first iteration all the states have not been generated

yet, so we randomly choose one state from the middle stage (state 1). This trial state fails to activate a_i ($\Phi(\{c_1, c_3\}, a_i) = false$) and so it is added to the $CloseList$ and continues the search with this state as a $leaf$. The next chosen state, between $\{c_1, c_3\}$ and the root C, is $\{c_1, c_3, c_4\}$ (there are only two options – the parents of $\{c_1, c_3\}$). This trial state succeeds to activate a_i and so we add it to $Results$ and continue the search between this state, as the $root$, and $\{c_1, c_3\}$ as the $leaf$. Since there are no more states between these states, and since $\{c_1, c_3\}$ is in the $CloseList$, we pop out $\{c_1, c_3\}$ from the $StackLOW$, and continue the search with the empty set as the $leaf$. Assume $\{c_4\}$ has been chosen ($\{c_1\}$ and $\{c_3\}$ are subsets of a state in the $CloseList$). This trial state activates a_i and so it is inserted to $Results$. The next iteration does not find any state between $\{c_4\}$ and the empty set, and so $\{c_4\}$ is popped out from the $StackUP$. The next iteration returns to the former search to search for more trial states between $\{c_1, c_3, c_4\}$ and the empty set. However, no fitting S can be found since either the states are supersets of $\{c_4\}$ or subsets of $\{c_1, c_3\}$. $\{c_1, c_3, c_4\}$ is popped out and we return to the first bounds, C and the empty set. Assume $\{c_1, c_2\}$ has been chosen (there is only one more possible option: $\{c_2, c_3\}$). This state is added to the $CloseList$ and we continue the search with this state as the new $leaf$. The last chosen trial state is $\{c_1, c_2, c_3\}$, which is added to the $CloseList$ too and then all the state space is either subsets of the $CloseList$ or supersets of $Results$.

To prove completeness of Algorithm 3 we first prove the following lemma:

Lemma 2. *At the end of the execution of Algorithm 3, $\forall S \in 2^C$ either $S \subseteq S' | S' \in CloseList$ or $S \supseteq S"| S" \in Results$.*

Proof. By initialization, state C is in the bottom of $StackUP$ (line 6). $StackLOW$ and the $CloseList$ contain only states that do not affirm a_i (lines 17–18), therefore C will stay in $StackUP$ as long as $StackLOW$'s top is not the empty set (lines 21–22). Thus in the last iteration of the algorithm $root = C$ and leaf=$\emptyset$. In this case, the algorithm tries to find a state in the full state space that satisfies the condition $S \nsubseteq S' | S' \in CloseList$ and $S \not\supseteq S"| S' \in Results$. Since it does not find such state the algorithm stops. Thus, $\forall S \in 2^C$ either $S \subseteq S' | S' \in CloseList$ or $S \supseteq S"| S' \in Results$. □

Now we can prove the algorithm's completeness:

Theorem 4. *$\forall S \subseteq C$ such that $\Phi(S, a_i) = true$ and there is no $S' \subset S$ such that $\Phi(S', a_i) = true$ then $S \in Results$.*

Proof. Based on Lemma 2, given $S \in 2^C$ either $S \subseteq S' | S' \in CloseList$ or $S \supseteq S"| S' \in Results$. If $\Phi(S, a_i) = true$ then, by line 14, there is no $S \nsubseteq S'$ such that $S' \in CloseList$. On the other hand, it must be added to $Results$. If not exist $S' \subset S$ such that $\Phi(S', a_i) = true$, S must be in $Results$ based on Subroutine 2 (Section 4.1). □

Algorithm 3 is sound:

Theorem 5. *$\forall S \in Results$, $\Phi(S, a_i) = true$ and there is no $S' \subset S$ such that $\Phi(S', a_i) = true$.*

Proof. Only states that affirm the condition $\Phi(S, a_i) = true$ are added to $Results$ (lines13–14), thus $\forall S \in Results$, $\Phi(S, a_i) = true$. Subroutine 2 (Section 4.1) guarantees minimality. □

Finally, we analyze the complexity of Algorithm 2:

Theorem 6. *The number of trial states is bounded by $O(|Results| n^2 \log n)$.*

Proof. Let us analyze the complexity of finding only one state that affirms $\Phi(S, a_i) = true$. In the binary search, we divide the state space in each iteration by two and continue only with one half. Thus, to find a trial state that affirms $\Phi(S, a_i) = true$ is $O(\log n)$, similarly, to find a trial state that affirms $\Phi(S, a_i) = false$ is $O(\log n)$. The worst case complexity of finding a child S' of C ($S' \subset C$, where $|S'| = |C| - 1$) that affirms $\Phi(S', a_i) = true$ is $O(n \log n)$. The reason is that it is possible that there is only one such a child; then we should verify, for each state S' of the other n children, that it does not activate a_i: $\Phi(S, a_i) = false$. To guarantee minimality, we continue this search iteratively with S' as the $root$, and then with the child of S' that activates a_i as the new $root$ and so on, until the the size of the $root$ is one, then one of these states must be minimal. The number of such iterations is bounded by n, and the overall worst case complexity to gauarantee one minimal trial state is $O(n^2 \log n)$. To find all the $|Results|$ trial states, we can iterate through the same process for each one of them and thus the worst case complexity will be $O(|Results| n^2 \log n)$. □

Unfortunately, the runtime complexity of finding the next trial state is exponential, since the number of possible states between the $root$ and the $leaf$ is the power set of the $root$ $O(2^n)$.

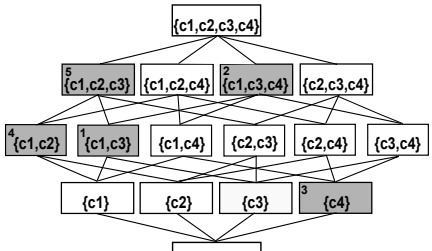

Figure 3: The nodes exploration order by the binary algorithm for $C : \{c_1, c_2, c_3, c_4\}$ action a_i is defined by $\{\Phi(\{c_4\}, a_i)\}$

6 EXPERIMENTAL RESULTS

In this section we present empirical evaluation for the algorithms. We run thousands of experiments by varying (1) the number of components (n) (2) the number of activating sets (v) and (3) the number of enabled components in each activating set (e). In particular, for a specific combination we generated a system with n components, and randomly selected v sets where each one of them contains e enabled components among the system. Since we chose the enabled components randomly, we run each combination 30 times. Then we run each algorithm trying to find the activating sets and measured the runtime and the number of trials the algorithm used.

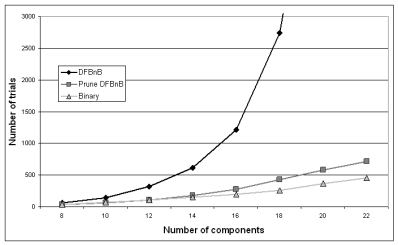

Figure 4: The impact of changing the number of components on the number of trials.

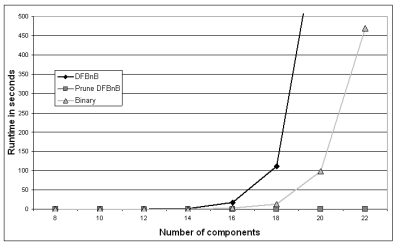

Figure 5: The impact of changing the number of components on the runtime.

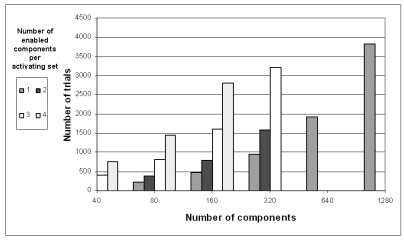

Figure 6: Prune DFBnB: the impact of #components in large scale systems. #activating sets = 3.

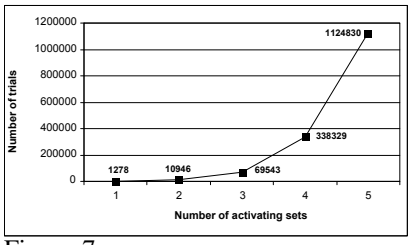

Figure 7: The impact of the the number of activating sets in the Prune DFBnB algorithm.

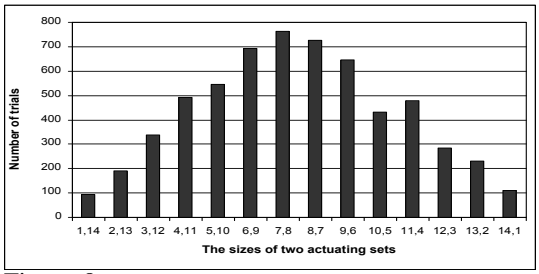

Figure 8: Prune DFBnB: the impact of the distribution of the enabled components among the activating sets.

In the first set of experiments, we examine the influence of the size of the system on the number of required trials (Figure 4) and on the runtime (Figure 5), for the three algorithms. In this experiment, the number of activating sets is fixed to two where each one of them contains $n/2$ enabled components selected randomly. The size of the system (n) was varied from 8 to 22 in a skip of two (represented by the x-axis). Every data point is an average over 30 executions randomizing the enabled components.

In Figure 4 the y-axis represents the number of required trials to recognize all the activating sets. As expected, by the complexity analysis, this number grows exponentially in the DFBnB algorithm, and polynomially in the Prune DFBnB and the binary search algorithms. There is no significant difference between the two algorithms since there are only two activating sets. Unfortunately, we could not run the binary search algorithm for large number of activating sets due to the exponential runtime of this algorithm. In Figure 5 we can see the runtime of the algorithm (the y-axis represents the runtime). As expected, the DFBnB and the binary search grow exponentially while the Prune DFBnB algorithm remains polynomial.

In the next set of tests we focus on the prune DFBnB algorithm to examine the number of required trials in large scale systems. The reason that we focus on this algorithm is that it is polynomial in runtime. The other two algorithms are not feasible to more than 30 components. Figure 6 shows the results for three activating sets. The x-axis represents the number of components and the y-axis represents the number of trials. The bars represent different sizes of activating sets (enabled components 1–4). We bounded the runtime of all the tests to one minute, thus we omitted bars that took more than one minute. Every data point is an average over 10 executions randomizing the enabled components taken from the first 16 components of the system.

It seems, from the figures, that Prune DFBnB algorithm scales well in terms of the number of trials. Note that the algorithm performs much better than the theoretical worst case. For example, the average number of trials where $n = 1280$ is $3800 \ll 3 * 1280^{(3+2)}$. In addition, we can see that as the number of enabled components per activating set grows, the number of queries grows. The reason for this fact will be investigated bellow (Figure 8). Finally, Prune DFBnB's complexity depends on the number of activating sets ($|Results|n^{|Results|+1}$), therefore we can see the high difference between the two figures. This is clearly shown in Figure 7, where the x-axis represents the number of activating sets. The system's size, in this experiment, is 40, and each one of the activating sets contains 7 enabled components. The curve grows exponentially as expected by the complexity analysis ($|Results|n^{|Results|+1}$).

In the previous experiments we distributed the enabled components uniformly among the activating sets. In the next experiment we examine the influence of other distributions. For that we fixed the number of components to 20, the number of activating sets to two. We run all the options of distributing 15 enabled components between the two activating sets $(1/14, 2/13, ...)$. For each one of this options we run

action	DFBnB		Prune-DFBnB		Binary	
	#trials	runtime	#trials	runtime	#trials	runtime
1	76	13934	21	6	153	60347
2	127216	1863477	2582	249	125	10940
3	134673	2163477	636	101	201	124406
4	1048575	5732982	39	56	34	6404

Table 1: Number of trials and runtime for Nao experiments.

30 tests randomizing the enabled components. Figure 8 shows the results of the experiment. The Y-axis represents the average number of trials, and the x-axis represents the number of enabled components in the activating sets.

As shown in Figure 8, the number of required trials grows as the distribution between the activating sets is close to uniform. The reason is that the space of optional trial states is $\binom{x}{n}$, where x is the number of enabled components in the set. This space is the smallest when the number of enabled components is the minimum or the maximum, and is is the largest when they are equal. We ran additional experiments for $n = 30$ 40 and 50 and for various number of enabled components, and the same pattern was observed.

Lastly, we examined the different algorithms in a simulation of real world tasks by recognizing the sets of components activating different actions performed by a NAO humanoid robot. It has 20 servos controlling its movements. These servos are the components in our problem, thus the size of NAO system is $n = 20$. We defined a set of actions which were explored by the different algorithms without making any assumptions on robot real architecture:

1. Walking - the task requires all the hands and legs to be active jointly (total of 18 joints).
2. Hitting button by hand - this task is performed by either right or left hand, so the correct output is two activating sets with four components each.
3. Tracking a moving ball with video sensor - this task is performed by moving only the head, or by moving the entire body to compensate for more extreme ball motion.
4. Trigger a security motion sensor in the room - The task requires a motion of any single servo.

Table 1 summarizes, for each one of the above actions, the results by running the different algorithms. The algorithms are presented in the columns, where the left sub-column represents the number of trials and the right sub-column represents the runtime. It seems that the runtime of Prune DFBnB algorithm is much less than the others, but the Binary search algorithm succeeded to achieve the least number of trials by inferring the activating servos of actions 2–4.

7 SUMMARY AND FUTURE WORK

In this paper we addressed the challenge of identifying a model, in particular, identifying the minimal sets of components that must be activated in order to operate a given action. We presented three algorithms: (i) a DFBnB-based algorithm that is exponential both in terms of number of the trials as well as in runtime ($O(2^n)$); (ii) an improved DFBnB-based algorithm with pruning with polynomial complexity ($O(|Results|n^{|Results|+1})$, under the reasonable assumption that the number of result sets is constant); (iii) a binary search based algorithm finds the minimal sets of components in polynomial number of trials ($O(|Results|n^2 \log n)$), yet it is exponential in terms of runtime ($O(2^n)$).

We empirically evaluated our algorithm and found that the Prune DFBnB algorithm is efficient both in terms of number of trials as well as in runtime. The binary search algorithm presented polynomial growth in number of trials but exponential in runtime. We examined also the Prune DFBnB algorithm in large scale systems and showed that it scale well both in the number trials and in runtime.

We also ran the Prune DFBnB algorithm on a model of the NAO robot for several test cases, and showed that it succeeded in finding the activating sets. Obviously, the Prune DFBnB and the binary algorithms solved the problem in reasonable number of trials.

In the future we plan to explore the similarity of our problem to the problem of the conflict sets in MBD. We plan to examine the use of the algorithms for finding conflict sets.

REFERENCES

(Black *et al.*, 2004) Richard Black, Austin Donnelly, and Cedric Fournet. Ethernet topology discovery without network assistance. In *In ICNP*, 2004.

(Daigle *et al.*, 2006) Matthew Daigle, Xenofon Koutsoukos, and Gautam Biswas. Multiple fault diagnosis in complex physical systems. In DX-06, pages 69–76, 2006.

(de Kleer and Williams, 1987) J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

(Farkas *et al.*, 2008) Janos Farkas, Vinicius G. de Oliveira, Marcos R. Salvador, and Giovanni C. dos Santos. Automatic discovery of physical topology in ethernet networks. In *22nd International Conference on Advanced Information Networking and Applications*. IEEE, 2008.

(Han and Kamber, 2001) Jiawei Han and Micheline Kamber. *Data mining : Concepts and techniques*. Morgan Kaufmann, 2001.

(O'Hallaron *et al.*, 2001) David R. O'Hallaron, Bruce Lowekamp, and Thomas R. Gross. Topology discovery for large ethernet networks. In *ACM Sigcomm 2001*, 2001.

(Papadimitriou and Steiglitz, 1982) Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

(Steinbauer *et al.*, 2009) Gerald Steinbauer, Alexander Kleiner, and Franz Wotawa. Using qualitative and model-based reasoning for sensor validation of autonomous mobile robots. In DX-09, pages 219–226, 2009.

(Williams and Ragno, 2007) Brian C. Williams and Robert J. Ragno. Conflict-directed a* and its role in model-based embedded systems. *Discrete Appl. Math.*, 155(12):1562–1595, 2007.