New Approach for Efficient Diagnosis of Large and Complex Space Systems

Amir Fijany[1] and Farrokh Vatan[2]
[1]Jet Propulsion Laboratory, California Institute of Technology, Amir.Fijany@jpl.nasa.gov
4800 Oak Grove Drive, Pasadena, CA 91109, USA
[2]Jet Propulsion Laboratory, California Institute of Technology, Farrokh.Vatan@jpl.nasa.gov
4800 Oak Grove Drive, Pasadena, CA 91109, USA

***Abstract*- We propose a novel algorithmic approach and present a new algorithm for solving the diagnosis problem. We report the results of the performance of the new algorithm and compare them with the traditional and standard algorithms. These results show the strong performance of our new algorithm with several orders of magnitude improvement over the traditional approach.**

I. INTRODUCTION

Over the past decade, the number of Earth orbiters and deep space probes has grown dramatically and is expected to continue in the future as miniaturization technologies drive spacecraft to become more numerous and more complex. In addition, the new NASA vision for space exploration requires long-term and/or manned missions to Moon, Mars, and Jupiter, demanding extremely reliable spacecraft for crew safety and/or assuring long-term operation. This rate of growth, in terms of number, complexity, and duration, has brought a new focus on autonomous and self-preserving systems that depend on fault diagnosis. Although diagnosis is needed for any autonomous system, current approaches are almost uniformly "ad-hoc," inefficient, and incomplete. Systematic methods of general diagnosis exist in literature, but they all suffer from two major drawbacks that severely limit their practical applications. First, they tend to be large and complex and hence difficult to apply. Second and more importantly, in order to find the minimal diagnosis set, i.e., the minimal set of faulty components, they rely on algorithms with exponential computational cost and hence are highly impractical for application for many systems of interest.

In the current state of practice, the most disciplined approach to fault detection and diagnosis is the "model-based" approach, employing knowledge of device operation and connectivity in the form of models. This approach, which reasons from first principles, provides far better diagnostic coverage than traditional approaches based upon collection of symptom-to-suspect rules. However, there are two major drawbacks in current model-based diagnosis systems that severely limit their practicality. First, these systems tend to be large, complex, and difficult to apply. Second, in order to find the minimal diagnosis set (i.e., the minimal set of components that, if faulty, would fully explain the anomalous behavior detected), they rely on algorithms with an exponential computational cost.

The most widely used approach to model-based diagnosis consists of a two-step process: (1) generating conflict sets from symptoms; (2) calculating minimal diagnosis set from the conflicts. Here a *conflict set* is a set of assumptions on the modes of *some* components that is not consistent with the model of the system and observations, and a *minimal diagnosis* is a set of the consistent assumptions of the modes of *all* components with minimal number of abnormal components. For finding minimal diagnosis from the conflicts, the most common algorithm is based on Reiter's algorithm, which requires both exponential time and exponential space (memory) for implementation.

In this paper, we address the problem of generating the minimal diagnosis from the conflicts. This problem can be formulated as the well-known Hitting Set Problem. Our approach starts by mapping the Hitting Set problem onto the Integer Programming Problem that enables us, *for the first time*, a priori determination of the lower and upper bounds on the size of the solution. Based on these bounds, we introduce a *new concept of solution window* for the problem. We also propose a new branch-and-bound technique that not only is faster than the current techniques in terms of number of operations (by exploiting the structure of the problem) but also, using the concept of solution window, allows a massive reduction (pruning) in the number of branches. Furthermore, as the branch-and-bound proceeds, the solution window is dynamically updated and narrowed to enable further pruning. The concept of window also allows us to propose a new portfolio approach, i.e., combining several different algorithms, for the problem. In this sense, several other fast algorithms (e.g., Randomized Algorithm), which usually lead to sub-optimal solutions, can be run in parallel with the branch-and-bound algorithm. The sub-optimal solutions generated by these algorithms are then used for further dynamic updating of the solution window.

We present the results of the performance of the new algorithm on a set of test cases. These results clearly show the

The research described in this paper was performed at the Jet Propulsion Laboratory (JPL), California Institute of Technology, under contract with the National Aeronautics and Space Administration (NASA).

advantage of our new algorithm over the traditional branch-and-bound algorithm; in fact the new algorithm has achieved several orders of magnitude speedup over the standard algorithms.

II. New Algorithmic Approach to Diagnosis Problem

To overcome the complexity of calculating minimal diagnosis set, we will utilize and expand our new discovery relating this calculation and the solution of the Hitting Set Problem to the solution of Integer Programming and Boolean Satisfiability Problems [1, 2]. Our primary interest in the Hitting Set Problem is due to its connection with the problem of diagnosis.

In order to describe the mapping of Hitting Set Problem onto Integer Programming, let us define a 0/1 (binary) matrix A as the incidence matrix of the collection of the conflict sets; i.e., the entry a_{ij}=1 if and only if the j^{th} element m_j belongs to the i^{th} set C_i:

$A =$

	m_1	m_2	...	m_n
C_1	1	0	...	0
C_2	0	1	...	1
			...	
C_m	1	1	...	0

Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ be a binary vector, wherein $x_j = 1$ if the member m_j belongs to the minimal hitting set and hence the minimal diagnosis set, otherwise $x_j = 0$. It can be then shown [1, 2] that we have the following formulation of the Hitting Set Problem as a 0/1 Integer Programming Problem:

$$\begin{aligned} &\textit{minimize} \quad x_1 + x_2 + \ldots + x_n, \\ &\textit{subject to} \quad A\boldsymbol{x} \geq \boldsymbol{b}, \quad x_j = 0 \text{ or } 1, \end{aligned} \tag{1}$$

where $\boldsymbol{b}^T = (1,\ldots,1)$ is a vector whose components are all equal to one. This new mapping allows us to utilize existing efficient integer programming algorithms, permitting solution of problems with a much larger size. In fact, we have shown [1] that, even using commercially available Integer Programming tools, we can achieve a more efficient calculation of minimal diagnosis compared with the existing algorithms.

III. Bounds on Diagnosis

This new mapping offers two additional advantages that can be exploited to develop yet more efficient algorithms. First, note that this mapping represents a special case of Integer Programming Problem due to the structure of matrix A (binary matrix) and vector $\boldsymbol{b}$. Second, by using this mapping, we can determine the minimum number of faulty components without solving the problem explicitly [1,2]. For this purpose, we consider the 1–norm and 2–norm of vectors defined as

$$\|\boldsymbol{x}\|_1 = \sum_{j=1}^{n} |x_j|, \quad \|\boldsymbol{x}\|_2 = \sqrt{\sum_{j=1}^{n} x_j^2}.$$

For the vector $\boldsymbol{b}$ in (1), we then have $\|\boldsymbol{b}\|_1 = m$ and $\|\boldsymbol{b}\|_2 = \sqrt{m}$. Since the elements of both vectors $A\boldsymbol{x}$ and $\boldsymbol{b}$ are positive, we can then derive the following two inequalities:

$$\left.\begin{aligned} \|A\|_1 \times \|\boldsymbol{x}\|_1 \geq m \quad &\Rightarrow \quad \|\boldsymbol{x}\|_1 \geq m / \|A\|_1 \\ \|A\|_2 \times \|\boldsymbol{x}\|_2 \geq \sqrt{m} \quad &\Rightarrow \quad \|\boldsymbol{x}\|_2 \geq \sqrt{m} / \|A\|_2 \end{aligned}\right\} \tag{2}$$

Since $\boldsymbol{x}$ is a binary vector, then both norms in (2) give the bound on the size of the solution, that is, the number of nonzero elements of vector $\boldsymbol{x}$ which, indeed, corresponds to the minimal diagnosis set. Note that, depending on the structure of the problem, i.e., the 1– and 2–norm of the matrix A and m, a sharper bound can be derived from either of (2). To our knowledge, this is the first time that such bounds on the solution of the problem have been derived without any need to explicitly solve the problem. Such a priori knowledge on the size of solution will be used for developing much more efficient algorithms for the problem.

Furthermore, using *monotonicity* of the integer programming (1), we are able to efficiently find an *upper bound* for the solution size. Here by monotonicity we mean that if $\boldsymbol{x}$ is a solution of $A\boldsymbol{x} \geq \boldsymbol{b}$ and $\boldsymbol{y} \geq \boldsymbol{x}$ then $\boldsymbol{y}$ is also a solution of the same system. Note that finding a 0/1 solution $\boldsymbol{x}$ for the system $A\boldsymbol{x} \geq \boldsymbol{b}$ is equivalent to finding a subset of the columns of the matrix A such that their sum is a vector with components all equal to or greater than 1. Of course, any such solution provides an *upper bound* for the optimization problem (1), since for that problem we are looking for a *minimal* set of such columns.

Therefore, to find an upper bound, we first choose a column c_1 of A with largest weight (if there are several such columns, we choose one of them arbitrarily). Then we construct a submatrix of A by deleting the column c_1 and all rows of A that correspond to non-zero components of c_1. We apply the same process to the new matrix, until we end up with the empty matrix. The columns c_1, c_2, ..., c_t that we obtain determine a solution for $A\boldsymbol{x} \geq \boldsymbol{b}$ and the number t is an upper bound for the solution of the integer programming problem (1). Our initial test shows that the upper bound is actually sharp, particularly for small size solution (see Table I). Note that it is easy to modify this algorithm in a way that the it also provides a vector $\boldsymbol{a}$ such that the vector $A\boldsymbol{a}$ realizes the corresponding upper bound.

There are two simple rules that will help this algorithm in extreme cases. These rules also can be useful in other cases, as by the recursive nature of the algorithm, most likely the algorithm will ends up with submatrices that these rules can be applied. Here is the formulation of these rules:

(I) If the matrix A has an all-one column, then its upper bound is equal to 1;

(II) If some row of the matrix A has weight 1, then remove that row and the corresponding column to obtain the matrix A_1 and **Upper_Bound** $[A] = 1 +$ **Upper_Bound** $[A_1]$.

We could also improve the upper bound by a step-by-step method and in an iterative fashion wherein the cost of k^{th} step in the iteration is $O(n^k)$ so the first few steps are practically efficient. More specifically, for fixed k, instead of choosing the maximum weight column for the vector $\boldsymbol{a}_1$, we could choose the sum of k columns of A, and try all possible such vectors.

As another application of the *a priori* lower bound, before starting to solve the hard problem of finding the minimal hitting sets, we could separate the cases where the high number of faulty components requires another course of

action instead of usual identification of faulty components. Also a good lower and upper bound could determine whether the enhanced brute-force algorithm [1, 2] can provide a solution efficiently. Since, as it was stated before, this algorithm has a complexity of $O(n^t)$, where t is the number of faulty components.

IV. The New Branch-and-Bound Method

Our new branch-and-bound algorithm is based on our methods for computing lower and upper bounds for diagnosis. We also exploit the monotonicity property of this special case of integer programming problem. We list the rationale behind this new approach, and its possible advantages over the standard branch-and-bound, as follows:

- Since the optimal solution is one of the points on the discrete grid, our relaxation phase directly applies to the discrete grid, while the standard method starts with the much larger set consists of not only the discrete gird but also all real points inside the polygon.
- For each subproblem we find a lower bound in linear time, while the time of LP relaxation of standard method is $O(n^3)$.
- For each subproblem we are able to find an upper bound, while in the standard method the upper bound is found only in the case that the LP relaxation of the subproblem at hand ends up with an integer solution. This way our method provides more chance to eliminate subproblems with large lower bounds.

Before describing our method, we introduce a set of useful functions and notation. We start with the $m \times n$ binary matrix A. We label the columns of A by the numbers 1, 2, …, n, and we denote any subset of these columns by simply as a subset of $\{1, 2, \ldots, n\}$. Similar to the traditional branch-and-bound method, the new algorithm is also based on search on the nodes of a tree. Each node of the search tree has a label of the form (M, T_{in}, T_{out}) where M is a submatrix of the original matrix A, and T_{in} and T_{out} are *disjoint* subsets of the columns of the original matrix A. The meaning of this partition is that T_{in} is the set of the columns (of the original matrix A) considered as part of the solution, T_{out} is the set of the columns (of the original matrix A) considered not as part of the solution. *Note that a solution of the optimization problem* (1) *can be considered as a subset of the columns of the matrix A whose addition is a vector with all non-zero components.*

Here is the list of the auxiliary functions and subroutines.

- **Function Place_Finding**

Consider a node with the label (M, T_{in}, T_{out}). Since M is a submatrix of the original matrix A obtained by removing the columns in the set $T_{in} \cup T_{out}$, every column of M corresponds with a unique column of A, for example the column 1 of M corresponds with the column 3 of A and the column 2 of M corresponds with the column 7 of A, and so on. Therefore, we can refer to the label of a *column of M in A* without any ambiguity. In fact, given the sets T_{in} and T_{out}, for every column j of M, it is possible to find the corresponding column in the original matrix A. We denote this relation by the function **Place_Finding**, also to keep the notation simple, instead of writing **Place_Finding**$[T_{in}, T_{out}, j]$ we simply write $\tilde{j}$ if the sets T_{in} and T_{out} are understood from the context.

- **Function Place_Finding**

Consider a node with the label (M, T_{in}, T_{out}). Since M is a submatrix of the original matrix A obtained by removing the columns in the set $T_{in} \cup T_{out}$, every column of M corresponds with a unique column of A, for example the column 1 of M corresponds with the column 3 of A and the column 2 of M corresponds with the column 7 of A, and so on. Therefore, we can refer to the label of a *column of M in A* without any ambiguity. In fact, given the sets T_{in} and T_{out}, for every column j of M, it is possible to find the corresponding column in the original matrix A. We denote this relation by the function **Place_Finding**, also to keep the notation simple, instead of writing **Place_Finding**$[T_{in}, T_{out}, j]$ we simply write $\tilde{j}$ if the sets T_{in} and T_{out} are understood from the context.

- **Functions Remove_0 and Remove_1**
  - **Remove_1** $[M, j]$: the result is the submatrix of M obtained by deleting the j^{th} column of M and deleting all rows of M that correspond with non-zero components of that column;
  - **Remove_0** $[M, j]$: the result is the submatrix of M obtained by deleting the j^{th} column of M.

For example, consider the matrix

$$M = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}.$$

Then to compute **Remove_1** $[M, 2]$, we first delete the 2nd column of M, and since the 2nd and 4th components of this column are 1, then we delete the 2nd and 4th rows. The result is

$$\textbf{Remove\_1}\ [M, 2] = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

While for **Remove_0** $[M, 2]$, we just delete the 2nd column of M. The result is

$$\textbf{Remove\_0}\ [M, 2] = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

- **Function Rule***

This function is based on some rules that simplify the process of finding the optimal solution for the system (1):

Rule 1 If the matrix A has an all-one column C_j, then the set consisting of the j^{th} element x_j alone is the hitting set; or the vector $\boldsymbol{x}$ with 1 at its j^{th} component and zero at other components, is an optimal solution for the system (1).

Rule 2 If the matrix A has a row of weight one, with 1 as its j^{th} component, then the j^{th} element x_j is contained in the minimal hitting set. In this case we can simplify the system (1) by removing this row and the j^{th} column, and we add the j^{th} element x_j to the solution of the new system.

Rule 3 If the matrix A has an all-one row, delete that row.

Rule 4 If the matrix A has an all-zero column, delete that column.

Rule 5 If the matrix A has two equal rows, delete one of them.

Let see how these Rules affect the labels of the nodes in the search tree. Suppose that $(M, \{T_{in}, T_{out}\})$ is the label of a node. First we describe the action of Rule 1. If the matrix M does not have an all-one column, then there no action is performed and the label is unchanged. Otherwise, assume that j^{th} column is all-one. Then Rule 1 changes the label to $(\varnothing, T_{in} \cup \{\tilde{j}\}, T_{out})$. In a more formal language, we define a function **Rule_1** on the set of labels as in Fig. 1.

To define the action of Rule 2, first we introduce a useful notation. Let e_j be the unit binary vector (of weight one) with its only 1 component at j^{th} position. Now the action of Rule 2 can be described by the formal function in Fig. 1.

The action of Rule 3 is described by the function in Fig. 1, where M' is obtained from M by deleting all all-one rows.

Finally, the actions of Rules 4 and 5 are described by the function in Fig. 1, where the matrix M' is obtained from M by deleting one of the equal rows.

Note that once one of these rules is applied on a label $\lambda_1 = (M, T_{in}, T_{out})$, and the result is the label λ_2 then it may be possible to apply one of these rules on λ_2, and so on. For these reason we define the function **Rule*** on the set of the labels as repeated applications of **Rule_1-5** until none of them can be applied anymore. It is easy to show that **Rule*** is well-defined. i.e., the result of **Rule***(λ) does not depend on the order of the functions **Rule_1** and **Rules_2** are applied.

As an example, consider the label $\lambda_1 = (M_1, \varnothing, \varnothing)$, where

$$M_1 = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

The 2nd row of M is equal to e_2 and its 3rd row is equal to e_4. Therefore, for applying **Rule_2** we have two possible choices. First we choose to remove 2nd row, the result is the label $\lambda_2 = (M_2, \{2\}, \varnothing)$, where

$$M_2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}.$$

The 2nd row of M_2 is equal to e_3, thus we can apply **Rule_2**. The result is the label $\lambda_3 = (M_3, \{2, 4\}, \varnothing)$, where

$$M_3 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Note that 3rd column of M_3 is the 4th column of the original matrix M_1. Finally, we apply **Rule_1**, and the result is the label $\lambda_4 = (\varnothing, \{2, 3, 4\}, \{1, 5\})$ (again note that the 2nd column of the matrix M_3 corresponds with the 3rd column of the original matrix M_1). Therefore,

Rule* $[\lambda_1] = (\varnothing, \{2, 3, 4\}, \{1, 5\})$.

- **Function Split**

We define the (partial) function **Split** on the set of the labels, where the value **Split** $[\lambda]$ is a *pair* (λ_0, λ_1) of labels. Suppose that $\lambda = (M, T_1, T_2)$. If $T_1 \cap T_2 = \{1, \ldots n\}$, i.e., if $T_1 \cap T_2$ is the set of all columns of the matrix A, then the function **Split** is not defined. Otherwise, let $j \notin T_1 \cap T_2$ be a column of the original matrix A which is corresponded with a maximum-weight column of the submatrix M (if there are several such columns then we choose the first one). Then we define two new labels based on the assumption that the j^{th} column is part of the solution set or not; more specifically, we define two new labels as follows:

$$\lambda_0 = (\textbf{Remove\_0}\ [M, j],\ T_1,\ T_2 \cup \{\tilde{j}\}),$$

$$\lambda_1 = (\textbf{Remove\_1}\ [M, j],\ T_1 \cup \{\tilde{j}\},\ T_2),$$

$$\textbf{Rule\_1}\ [(M, T_{in}, T_{out})] = \begin{cases} (M, T_{in}, T_{out}) & \text{if } M \text{ has no all-one column,} \\ (\varnothing, T_{in} \cup \{\tilde{j}\}, T_{out}) & \text{if } j\text{th column is an all-one column.} \end{cases}$$

$$\textbf{Rule\_2}\ [(M, T_{in}, T_{out})] = \begin{cases} (\textbf{Remove\_1}\ [M, j], T_{in} \cup \{\tilde{j}\}, T_{out}) & \text{if } M \text{ has a row equal to } e_j, \\ (M, T_{in}, T_{out}) & \text{otherwise.} \end{cases}$$

$$\textbf{Rule\_3}\ [(M, T_{in}, T_{out})] = \begin{cases} (M', T_{in}, T_{out}) & \text{if } M \text{ has all-one rows,} \\ (M, T_{in}, T_{out}) & \text{otherwise,} \end{cases}$$

$$\textbf{Rule\_4}\ [(M, T_{in}, T_{out})] = \begin{cases} (\textbf{Remove\_0}\ [M, j], T_{in}, T_{out} \cup \{\tilde{j}\}) & \text{if } j^{th} \text{ column of } M \text{ is all-zero,} \\ (M, T_{in}, T_{out}) & \text{otherwise.} \end{cases}$$

$$\textbf{Rule\_5}\ [(M, T_{in}, T_{out})] = \begin{cases} (M', T_{in}, T_{out}) & \text{if } M \text{ has two equal rows,} \\ (M, T_{in}, T_{out}) & \text{otherwise,} \end{cases}$$

Figure 1. The definitions of Rule functions

Then the **Split** function is defined as

$$\textbf{Split}\ [\lambda] = (\textbf{Rule*}\ [\lambda_0], \textbf{Rule*}\ [\lambda_1]).$$

- **Function Upper_Bound**

The **Upper_Bound** function is defined in Section 3 to find the number **Upper_Bound** $[A]$ as an upper bound on the solution of the optimization problem defined by the system (1). We extend this function to the set of labels as follows. For the label $\lambda = (M, T_1, T_2)$, where M is a submatrix of the original matrix A, **Upper_Bound** $[\lambda]$ provides an upper bound for the system defined by (1) augmented by the following conditions:

$$x_j = 1, \quad x_j \in T_1,$$
$$x_j = 0, \quad x_j \in T_2.$$

Then it is easy to see that

$$\textbf{Upper\_Bound}\ [\lambda] = |T_1| + \textbf{Upper\_Bound}\ [M].$$

In the special case that $M = \varnothing$, we have **Upper_Bound** $[\lambda] = |T_1|$. Note that we apply the function **Upper_Bound** both on matrices and labels.

- **Function Upper_Bound_Set**

For the label $\lambda = (M, T_1, T_2)$, the function **Upper_Bound_Set**$[\lambda]$ returns the set which realizes the bound **Upper_Bound** $[\lambda]$, i.e., the union of T_1 and the set of columns of M which provide the bound **Upper_Bound** $[M]$.

- **Function Lower_Bound**

Like the previous function, we extend the lower bound defined by (2) to the set of labels. More specifically, for the label $\lambda = (M, T_1, T_2)$, where M is a $k \times j$ submatrix of the original matrix A, we have

$$\textbf{Lower\_Bound}\ [\lambda] = |T_1| + k / \|M\|_1 .$$

- **Function Test_Solution**

This function is defined on the set of the labels and its value is either *True* or *False*. The value of **Test_Solution** $[(M, T_1, T_2)]$ is *True* if the columns in the set T_1 form a solution for the system (1). Otherwise, the value of the function is *False*.

- **Function Test_Leaf**

This function is defined on the set of the labels and its value is either *True* or *False*. As it is suggested by the name of the function, here we determine whether a node in the search tree is a leaf or not; i.e., whether that node has any children or not. The arguments of this function are a label $\lambda = (M, T_1, T_2)$ and a value U for the *upper bound* on the solution of the problem. Then

$$\textbf{Test\_Leaf}\ [\lambda, U] = \begin{cases} True & \text{if } T_1 \cup T_2 = \{1, 2, \mathrm{K}\ , n\}, \text{ or} \\ True & \text{if } \textbf{Lower\_Bound}\ [\lambda] \geq U, \text{ or} \\ True & \text{if } \textbf{Test\_Solution}\ [\lambda] = True, \text{ or} \\ True & \text{if } M \text{ contains an all - zero row,} \\ False & \text{otherwise.} \end{cases}$$

Now we are ready to present Our new branch-and-bound algorithm. This algorithm is described in Fig. 2. It is easy to show that this algorithm is correct. The complete proof we will be presented in the subsequent paper.

Table I shows the results of performance of the new algorithm and its comparison with the traditional Branch-and-Bound method. These results show the average time and the number of iterations (i.e., the number of nodes in the search tree) used by these algorithms on 100 random binary matrices.

```
branch-and-bound (A)
/* solves the hitting set problem defined by the system (1) */
Labels = {Rule* [(A, ∅, ∅)]}
U = infinity        /* upper bound */
Solution = ∅
while Labels ≠ ∅
   chose λ = (M, T1, T2) ∈ Labels
   Labels = Labels − {λ}
   If Test_Solution[λ] = True & Upper_Bound[λ] < U then
      Solution = T1
      U = Upper_Bound[λ]
   end if
   If Test_Solution[λ] = True & Upper_Bound[λ] = U & Solution =∅then Solution = T1
   If Upper_Bound[λ] < U then
      U = Upper_Bound[λ]
      Solution = Upper_Bound_Set[λ]
   end if
   If Test_Leaf[λ, U] = False then
      (λ0, λ1) = Split[λ]
      Labels = Labels ∪ {λ0, λ1}
   end if
   If Test_Leaf[λ, U] = True & Upper_Bound[λ] = U & Solution =∅
      then Solution = Upper_Bound_Set[λ]
end while
return Solution
```

Figure 2. The new Branch-and-bound algorithm

TABLE I
COMPARING THE AVERAGE PERFORMANCE OF ALGORITHMS ON 100 RANDOM MATRICES

Size	Traditional Branch-and-Bound		New Branch-and-Bound	
	Time (seconds)	Size of the tree	Time (seconds)	Size of the tree
30×20	0.0045	63.49	0.00016	32.1
40×30	0.018	186.35	0.0017	130.48
35×40	0.027	283.65	0.0028	125.46
50×40	0.059	514.8	0.0077	311.72
40×50	0.07	634.1	0.005	207.22
60×45	0.164	1200.38	0.0163	609.94
70×50	0.406	2754.06	0.048	1617.6
80×55	0.917	5676.74	0.123	3544.24
60×70	0.455	2638.9	0.0343	911.7
100×80	7.01	33929.5	0.659	14902.5
100×90	13.469	55836.7	1.82	20841.4
120×90	18.55	65984.1	1.246	22269.9
130×100	40.85	123596	1.87	29093.6
150×120	177.475	427467	3.658	46474.9

For above benchmarking, we used the GLPK (GNU Linear Programming Kit), version 4.7, to solve the problems with the traditional branch-and-bound method. This is a set of routines in the ANSI C programming language. The integer programming routine of GLPK (actually it is much more powerful routine and is capable of solving mixed integer programming problems) applies a variant of branch-and-bound method for the problem.

V. A NOVEL CONFLICT GENERATION ALGORITHM

We introduce a novel approach for generating conflict sets based on mapping this problem onto the well-studied problem of finding paths in a graph [5]. The main idea of this approach is based on the fact that only the value of observed parameters can produce the conflicts; i.e., if the description of the system and the value of the inputs could imply a value different from the observed value. We should also consider the values that could be inferred from the observed values by the "back-propagation" method; i.e., the values that could be inferred at some node from the values observed at the other nodes. All subsystems that are involved in the process of finding these inferred values can be described as paths on the graph of the system. Therefore, to find all conflict sets, it is enough to consider only paths that start at inputs or nodes of observed values and end at one of these nodes. This approach can significantly accelerate the conflict generation step by bounding the search space. The details of this method will be explained in the subsequent paper.

VI. SUMMARY AND CONCLUSIONS

We proposed a new approach to overcome one of the major limitations of the current model-based diagnosis techniques, that is, the exponential complexity of calculation of minimal diagnosis set. To overcome this challenging limitation, we have proposed a novel algorithmic approach for calculation of minimal diagnosis set. Starting with the relationship between the calculation of minimal diagnosis set and the celebrated Hitting Set problem, we have proposed a new method for solving the Hitting Set Problem, and consequently the diagnosis problem. This method is based on a powerful and yet simple representation of the problem that enables its mapping onto another well-known problem, that is, the 0/1 Integer Programming problem.

The mapping onto 0/1 Integer Programming problem enables the use of variety of algorithms that can efficiently solve the problem for up to several thousand components. Therefore, these new algorithms significantly improve over the existing ones, enabling efficient diagnosis of large complex systems. In addition, this mapping enables *a priori* and fast determination of the lower and upper bounds on the solution, i.e., the minimum number of faulty components, before solving the problem. We exploit this powerful insight to develop yet more powerful algorithm for the problem. This new algorithm is a new version of the well-known branch-and-bound method. We present the results of the performance of the new algorithm on a set of test cases. These results clearly show the advantage of our new algorithm over the traditional branch-and-bound algorithm; more specifically the new algorithm has achieved several orders of magnitude speedup over the standard algorithms.

REFERENCES

[1] A. Fijany, F. Vatan, A. Barrett, and R. Mackey, "New approaches for solving the diagnosis problem," *The JPL Interplanetary Network Progress (IPN) Report*, May 2002; available at http://ipnpr.jpl.nasa.gov/tmo/progress_report/42-149/149K.pdf.

[2] A. Fijany, F. Vatan, A. Barrett, M. James, C. Williams, and R. Mackey, "A novel model-based diagnosis engine: Theory and Applications," *Proc. 2003 IEEE Aerospace Conference*, March 2003.

[3] F. Vatan, "The complexity of diagnosis problem," *NASA Tech Briefs*, vol. 26, p. 20, 2002.

[4] J. de Kleer, A. K. Mackworth, and R. Reiter, "Characterizing diagnoses and systems," *Artificial Intelligence*, vol. 56, 197–222, 1992.

[5] G. Rote, Path problems in graphs, *Computing*, vol. 7, pp. 155–189, 1990.

[6] T. Hogg and C. Williams, "Solving the really hard problems with cooperative search," *Proc. of AAAI-93*, pp. 231–236, 1993.

[7] B.C. Williams and P. Nayak, "A model-based approach to reactive self-configuring systems," *Proc. 13th Nat. Conf. Artif. Intell. (AAAI-96)*, pp. 971–978, 1996.

[8] S. Chung, J.V. Eepoel, and B.C. Williams, "Improving model-based mode estimation through offline compilation," *Int. Symp. Artif. Intell., Robotics, Automation Space (ISAIRAS-01)*, 2001.

[9] F. Wotawa, "A variant of Reiter's hitting-set algorithm," *Information Processing Letters*, vol. 79, 45–51, 2001.

[10] J. de Kleer and B. Williams, "Diagnosing Multiple Faults" in *Readings in Model-Based Diagnosis*, Morgan Kaufmann Publishers, San Mateo, CA, 1992, pp. 100–117.

[11] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons, New York, 1986.