

New High Performance Algorithmic Solution for Diagnosis Problem

Amir Fijany and Farrokh Vatan

Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109

ABSTRACT

We propose a novel algorithmic approach and present a new algorithm for solving the diagnosis problem. We report the results of the performance tests of the new algorithm and compare them with the traditional and standard algorithms. These results show the strong performance of our new algorithm with more than ten times improvement over the traditional approach.

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. However, there are major drawbacks in the current model-based diagnosis techniques in efficiently performing the above two steps that severely limit their practical application to many systems of interest. For conflict generating problem, these techniques are usually based on different versions of Truth Maintenance method, which lead to an exhaustive search in the space of possible modes of the components. For finding minimal diagnosis from the conflicts, the most common 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 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.

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 algorithm. For example, for the systems with 40 components, the new algorithm, in average, solves the problem more than 300 times faster than the traditional algorithm.

1. Introduction

The *Hitting Set Problem*, also known as the *Transversal Problem*, is one of key problems in the combinatorics of finite sets and the theory of diagnosis [4]. The problem

is simply described as follows. A collection $S = \{S_1, \dots, S_m\}$ of nonempty subsets of a set M is given. A hitting set of S is a subset H of M that intersects every set in the collection S . Of course, there are always trivial hitting sets, for example the background set M is always a hitting set. Actually we are interested in *minimal hitting sets with minimal cardinality*: a hitting set H is minimal if no proper subset of H is a hitting set.

Our primary interest to Hitting Set Problem is its connection with the problem of diagnosis. The diagnosis problem arises when some symptoms (anomalies) are observed, that is, when system's actual behavior contradicts the expected behavior. System diagnosis is then the task of identifying faulty components that are responsible for anomalous behavior. To solve the diagnosis problem, one must find the *minimal set* of faulty components that explain the observed symptoms. The most disciplined technique to diagnosis is termed "model-based" because it employs knowledge of devices' operation and their connectivity in the form of models. This approach [4] reasons from first principles and affords far better diagnostic coverage of a system than traditional rule-based diagnosis methods, which are based on a collection of specific symptom-to-suspect rules.

The diagnosis process starts with identifying symptoms that represent inconsistencies (discrepancies) between the system's model (description) and the system's actual behavior. Each symptom identifies a set of *conflicting* components as initial candidates. A *minimal diagnosis* is the smallest set of components that intersects all conflict sets. The underlying general approach in different model-based diagnosis approaches can be described as a two-step "divide-and-conquer" technique wherein finding the minimal diagnosis set is accomplished in two steps: a) Generating conflict sets from symptoms, and b) Calculating minimal diagnosis set from the conflict sets. In summary, the conflict generation corresponds to forming a collection of sets, and calculating minimal diagnosis corresponds to the solution of the hitting set problem for this collection (see Figure 1).

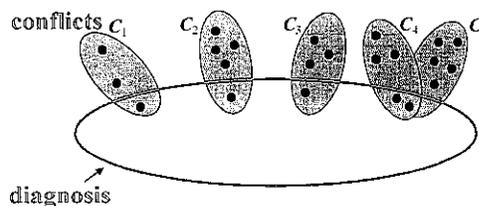


Figure 1. Diagnosis as the hitting set of conflict sets

However, there are major drawbacks in the current model-based diagnosis techniques in efficiently performing the above two steps that severely limit their practical application to many systems of interest. First, the existing conflict generating algorithms are all based on various versions of *constraint propagation method* and *truth maintenance systems*. The problem with these methods is that not only they need exponential time but usually they also require exponential memory to be implemented. Therefore, these methods cannot handle realistic systems with large number of components. Second, in order to find the minimal diagnosis set, current model-based diagnosis techniques rely on algorithms with exponential computational cost and hence are highly impractical for application to many systems of interest.

In this paper, we present a powerful yet simple representation of the calculation of minimal diagnosis set. This representation enables the mapping onto a 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 algorithm. For example, for the systems with 40 components, the new algorithm, in average, solves the problem more than 300 times faster than the traditional algorithm.

2. A 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 (see Figure 2) 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 . Let $x = (x_1, x_2, \dots, x_n)$ 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:

$$\begin{aligned} & \text{minimize} && x_1 + x_2 + \dots + x_n, \\ & \text{subject to} && Ax \geq \mathbf{b}, \quad x_j = 0, 1, \end{aligned} \tag{1}$$

where $\mathbf{b}^T = (1, \dots, 1)$ 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.

$$A = \begin{array}{c|cccc} & m_1 & m_2 & \dots & m_n \\ \hline C_1 & 1 & 0 & \dots & 0 \\ C_2 & 0 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_m & 1 & 1 & \dots & 0 \end{array}$$

Figure 2. Mapping hitting set problem onto integer programming

3. 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 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

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

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

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

Since 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 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 x is a solution of $Ax \geq b$ and $y \geq x$ then y is also a solution of the same system. Note that finding a 0/1 solution x for the system $Ax \geq b$ 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. 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_1, \dots, C_t that we obtain determine a solution for $Ax \geq b$ 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 a such that the vector Aa realizes the corresponding upper bound.

```

Upper_Bound [A]
/* returns an upper bound for the solution of  $Ax \geq b$  */

 $U_1 \leftarrow 1$ 
 $a_1 \leftarrow$  a maximum-weight column of  $A$ 
 $A_1 \leftarrow$  submatrix of  $A$  obtained by deleting the column  $a_1$  and all rows
of  $A$  that correspond to non-zero components of  $a_1$ 
if  $A_1$  is the empty matrix return  $U_1$ 
else return  $U_1 + \text{Upper\_Bound} [A_1]$ 
end if

```

Figure 3. A recursive procedure for computing the 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 end 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 the 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 $\text{Upper_Bound} [A] = 1 + \text{Upper_Bound} [A_1]$.

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 fix k , instead of choosing the maximum weight column for the vector a_1 , we could choose the sum of k columns of A , and try all possible such vectors.

Table I. Upper and Lower Bounds

Size of the Matrix	Size of the Hitting Set	Upper Bound	Lower Bound
43x21	3	3	2
17x21	6	7	3
17x33	5	5	4
159x25	4	4	2
39x25	7	8	2
21x38	5	5	3
23x38	6	6	4
27x40	6	8	2
21x60	7	8	6
78x63	13	18	7
94x76	16	22	8

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.

4. The Traditional Branch-and-Bound Method

The branch-and-bound method is one of the most common methods for solving NP-complete problems. For the case of Integer Programming (IP) problem, this method traditionally begins by solving the *Linear Programming (LP) relaxation* of the IP; i.e., by removing the condition that the variables x_j are integers. Figure 4 shows typical solution sets of the LP relaxation and IP problems, where the polygon represents the solution set of the LP relaxation and the grid points inside this polygon represent the solution set of the IP problem. If the optimal solution of the LP relaxation consists only of integer values, then the optimal solution of the LP relaxation will be the optimal solution for the IP problem. Otherwise, if the IP problem is defined by a system like (1), the optimal solution of the LP relaxation provides a *lower bound* for the IP problem. In this case, we choose one of the non-integer values of the optimal solution of the LP relaxation, say $x_j = a$, and define two new subproblems by adding the conditions $x_j \leq [a]$ and $x_j \geq [a]+1$ to the system, where $[a]$ denotes the integer part of a . Then we have to solve the two new subproblems. By continuing this procedure, we define subproblems of the original IP problem. Once we find an *integer* optimal solution for the LP relaxation of a subproblem, that solution gives us an *upper bound* for the IP problem. After finding such integer solution, we eliminate any subproblem whose (LP relaxation) lower bound is larger than upper bound provided by some other subproblem. We continue this procedure until all subproblems are eliminated or we find an *integer* optimal solution for the corresponding LP relaxation problem. At the end, the optimal solution for the IP problem is the best of the optimal integer solutions of the subproblems.

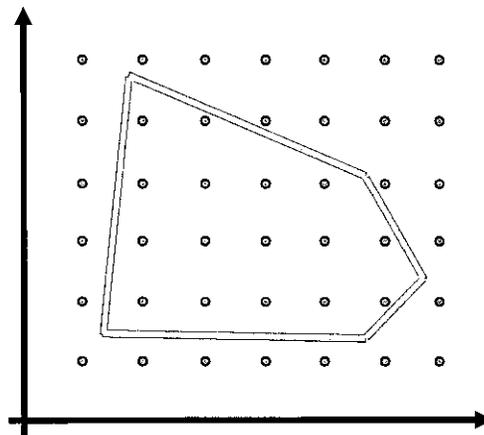


Figure 4. Solution sets of an Integer Programming problem and a corresponding Linear Programming relaxation problem

We can simply generalize the above procedure as follows. To begin the branch-and-bound procedure, we need to have the capability to perform the following tasks:

- (a) Partitioning a problem P into a collection P_1, P_2, \dots, P_k of mutually disjoint subproblems;
- (b) Finding a lower bound for each subproblem P_j ;
- (c) Finding an integer solution for a subclass of subproblems (in this case we obtain an upper bound), we could also determine whether the integer solution is *optimal* for the corresponding subproblem.

Note that in the case of LP relaxation, whenever we find an integer solution for a subproblem, it is guaranteed that it is an optimal solution for that subproblem. We now start with the original problem P , defined by a system like (1), and in the m^{th} step of the procedure we have a partition $P_{m,1}, P_{m,2}, \dots, P_{m,j}$ of mutually disjoint subproblems. For each subproblem $P_{m,j}$ we find a lower bound $L_{m,j}$ and, if possible, an integer solution and corresponding upper bound $U_{m,j}$. Then we decide which subproblems $P_{m,j}$ should be eliminated at this step. There are two criteria for this decision:

- (i) The lower bound $L_{m,j}$ is larger than some upper bound $U_{m,k}$ of some other subproblem $P_{m,k}$;
- (ii) An optimal solution for the subproblem $P_{m,j}$ is found.

In the case (ii) we keep the record of the best optimal solution of the subproblems. Then we apply the partitioning method (a) on the remaining subproblems and find lower and (if possible) upper bounds for the new subproblems. We continue this procedure until no subproblem remains. Then the best of the optimal solutions of subproblems is the optimal solution for the original problem.

5. 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 rational 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 of Figure 4, 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 grid 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 $1, 2, \dots, n$, and we denote any subset of these columns by simply as a subset of $\{1, 2, \dots, n\}$. 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

$$(M, T_{\text{in}}, T_{\text{out}})$$

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 , 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

$$\mathbf{Remove_1} [M, 2] = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

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

$$\mathbf{Remove_0} [M, 2] = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

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 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 $(M, \{T_{\text{in}}, T_{\text{out}}\})$ 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

$$(\emptyset, T_{\text{in}} \cup \{\tilde{j}\}, T_{\text{out}}).$$

In a more formal language, we define a function **Rule_1** on the set of labels as follows:

$$\mathbf{Rule_1} [(M, T_{\text{in}}, T_{\text{out}})] = \begin{cases} (M, T_{\text{in}}, T_{\text{out}}) & \text{if } M \text{ has no all - one column,} \\ (\emptyset, T_{\text{in}} \cup \{\tilde{j}\}, T_{\text{out}}) & \text{if } j^{\text{th}} \text{ column is an all - one column.} \end{cases}$$

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 following function:

$$\mathbf{Rule_2} [(M, T_{\text{in}}, T_{\text{out}})] = \begin{cases} (\mathbf{Remove_1} [M, j], T_{\text{in}} \cup \{\tilde{j}\}, T_{\text{out}}) & \text{if } M \text{ has a row equal to } e_j, \\ (M, T_{\text{in}}, T_{\text{out}}) & \text{otherwise.} \end{cases}$$

The action of Rule 3 is described by the following function

$$\mathbf{Rule_3} [(M, T_{\text{in}}, T_{\text{out}})] = \begin{cases} (M', T_{\text{in}}, T_{\text{out}}) & \text{if } M \text{ has all - one rows,} \\ (M, T_{\text{in}}, T_{\text{out}}) & \text{otherwise,} \end{cases}$$

where M' is obtained from M by deleting all all-one rows. The action of Rule 4 is described by the following function

$$\mathbf{Rule_4}[(M, T_{in}, T_{out})] = \begin{cases} (\mathbf{Remove_0}[M, j], T_{in}, T_{out} \cup \{j\}) & \text{if } j^{\text{th}} \text{ column of } M \text{ is all-zero,} \\ (M, T_{in}, T_{out}) & \text{otherwise.} \end{cases}$$

Finally, the following function describes Rule 5

$$\mathbf{Rule_5}[(M, T_{in}, T_{out})] = \begin{cases} (M', T_{in}, T_{out}) & \text{if } M \text{ has two equal rows,} \\ (M, T_{in}, T_{out}) & \text{otherwise,} \end{cases}$$

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 \mathbf{Rule}^* on the set of the labels as repeated applications of $\mathbf{Rule_1-5}$ until none of them can be applied anymore. It is easy to show that \mathbf{Rule}^* is well-defined. i.e., the result of $\mathbf{Rule}^*(\lambda)$ does not depend on the order of the functions $\mathbf{Rule_1}$ and $\mathbf{Rules_2}$ are applied. As an example, consider the label

$$\lambda_1 = (M_1, \emptyset, \emptyset),$$

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 $\mathbf{Rule_2}$ we have two possible choices. First we choose to remove 2nd row, the result is the label $\lambda_2 = (M_2, \{2\}, \emptyset)$, 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 $\mathbf{Rule_2}$. The result is the label $\lambda_3 = (M_3, \{2, 4\}, \emptyset)$, 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 $\lambda_4 = (\emptyset, \{2, 3, 4\}, \{1, 5\})$ (again note that the 2nd column of the matrix M_3 corresponds with the 3rd column of the original matrix M_1). Therefore,

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

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 $T_1 \cup T_2 = \{1, \dots, n\}$, i.e., if $T_1 \cup T_2$ is the set of all columns of the matrix A , then the function **Split** is not defined. Otherwise, let $j \notin T_1 \cup T_2$ 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:

$$\lambda_0 = (\mathbf{Remove_0} [M, j], T_1, T_2 \cup \{\tilde{j}\}),$$

$$\lambda_1 = (\mathbf{Remove_1} [M, j], T_1 \cup \{\tilde{j}\}, T_2),$$

Then the **Split** function is defined as

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

Function Upper_Bound

The **Upper_Bound** function is defined in Section 4 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

$$\mathbf{Upper_Bound} [\lambda] = |T_1| + \mathbf{Upper_Bound} [M].$$

In the special case that $M = \emptyset$, we have $\text{Upper_Bound} [\lambda] = |T_1|$. 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 $\text{Upper_Bound_Set}[\lambda]$ returns the set which realizes the bound $\text{Upper_Bound} [\lambda]$, i.e., the union of T_1 and the set of columns of M which provide the bound $\text{Upper_Bound} [M]$.

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

$$\text{Lower_Bound} [\lambda] = |T_1| + k/\|M\|_1.$$

Function Test_Solution

This function is defined on the set of the labels and its value is either *True* or *False*. The value of $\text{Test_Solution} [(M, T_1, T_2)]$ 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

$$\text{Test_Leaf} [\lambda, U] = \begin{cases} \text{True} & \text{if } T_1 \cup T_2 = \{1, 2, \dots, n\}, \text{ or} \\ \text{True} & \text{if } \text{Lower_Bound} [\lambda] \geq U, \text{ or} \\ \text{True} & \text{if } \text{Test_Solution} [\lambda] = \text{True}, \text{ or} \\ \text{True} & \text{if } M \text{ contains an all-zero row,} \\ \text{False} & \text{otherwise.} \end{cases}$$

Now we are ready to present Our new branch-and-bound algorithm. This algorithm is described in Figure 5.

```

branch-and-bound (A)
/* solves the hitting set problem defined by the system (1) */

1. Labels ← {Rule* [(A, ∅, ∅)]}
2. U ← ∞ /* upper bound */
3. Solution ← ∅
4. while Labels ≠ ∅
5.   chose λ = (M, T1, T2) ∈ Labels
6.   Labels ← Labels - {λ}
7.   If Test_Solution [λ] = True & Upper_Bound [λ] < U then
8.     Solution ← T1
9.     U ← Upper_Bound [λ]
10.  end if
11.  If Test_Solution [λ] = True & Upper_Bound [λ] = U & Solution = ∅ then Solution ← T1
12.  If Upper_Bound [λ] < U then
13.    U ← Upper_Bound [λ]
14.    Solution ← Upper_Bound_Set[λ]
15.  end if
16.  If Test_Leaf [λ, U] = False then
17.    (λ0, λ1) ← Split [λ]
18.    Labels ← Labels ∪ {λ0, λ1}
19.  end if
20.  If Test_Leaf [λ, U] = True & Upper_Bound [λ] = U & Solution = ∅
21.    then Solution ← Upper_Bound_Set[λ]
22.  end while
23.  return Solution

```

Figure 5. The new Branch-and-bound algorithm

Table II 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.

Table II. Comparing the average performance of algorithms on 100 random matrices

Size of the matrix	Traditional Branch-and-Bound		New Branch-and-Bound	
	Time (seconds)	Number of iterations	Time (seconds)	Number of iterations
20×25	0.98	26.08	0.18	12.13
23×26	3.68	55.52	0.19	13.13
25×30	5.64	53.28	0.34	18.28
28×31	20.59	126.48	0.45	26.2
30×35	31.92	130.34	0.77	36.25
33×36	83.95	245.42	0.84	41.46
35×40	127.32	252.54	1.42	58.84
38×41	273.79	404.12	2.58	106.32
40×45	633.22	573.4	2.93	110.07
45×50	2392.88	933.88	6.00	196.5

Example

In this example we consider the following 10×12 matrix:

$$A_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

The nodes of the search tree and their labels are as follows.

Node 0. The label of the root is $\lambda_0 = (A_0, \emptyset, \emptyset)$. In this case, we have $\text{Rule}^*[\lambda_0] = \lambda_0$. The initial lower and upper bounds are $\text{Lower_Bound} = 2$ and $U = \text{Upper_Bound} = 5$, the upper bound is realized with the set $\{3, 4, 5, 6, 8\}$. Moreover, $\text{Test_Leaf}[\lambda_0, U] = \text{False}$, thus this node has two children: Node 1 and Node 2. To find them, notice that column 8 has maximum weight, then

$$\text{Remove_0}[A_0, 8] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} = A_1,$$

and

$$\text{Remove_1}[A_0, 8] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Now note that none of the Rules 1-5 can be applied on the matrix $\text{Remove_0}[A_0, 8]$, and only Rule 4 can be applied to $\text{Remove_1}[A_0, 8]$, because 1st column of this matrix is an all-zero vector. After applying this rule, we get the following matrix

$$A_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Therefore, the labels of the children of Node 0 are

$$\begin{aligned} \lambda_1 &= (A_1, \emptyset, \{8\}), \\ \lambda_2 &= (A_2, \{8\}, \{1\}). \end{aligned}$$

Node 1. Its label is λ_1 . The updated lower and upper bounds are $\text{Lower_Bound} = 2$ and $U = \text{Upper_Bound} = 4$, the upper bound is realized with the set $\{3, 5, 6, 9\}$, and $\text{Test_Leaf}[\lambda_1, U] = \text{False}$, thus this node has two children: Node 3 and Node 4. The column 8 of the matrix A_1 , which is the 9th column of the original matrix A_0 , has the maximum weight. Then

$$\text{Remove_0}[A_1, 8] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} = A_3,$$

and

$$\text{Remove_1}[A_1, 8] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

None of the Rules 1-5 can be applied on the matrix $A_3 = \text{Remove_0}[A_1, 8]$, and only Rule 4 can be applied to $\text{Remove_1}[A_1, 8]$, because 4th column of this matrix, which is also the 4th column of the original matrix A_0 , is an all-zero vector. After applying this rule, we get the following matrix

$$A_4 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

Therefore, the labels of the children of Node 1 are

$$\lambda_3 = (A_3, \emptyset, \{8, 9\}),$$

$$\lambda_4 = (A_4, \{9\}, \{4, 8\}).$$

Node 2. Its label is λ_2 . The updated lower and upper bounds are $\text{Lower_Bound} = 4$ and $U = \text{Upper_Bound} = 4$. Since $\text{Lower_Bound} = U$, $\text{Test_Leaf}[\lambda_2, U] = \text{True}$, thus this node is a leaf of the search tree.

Node 3. Its label is λ_3 . The updated lower and upper bounds are $\text{Lower_Bound} = 2$ and $U = \text{Upper_Bound} = 3$, the upper bound is realized with the set $\{6, 11, 12\}$, and $\text{Test_Leaf}[\lambda_3, U] = \text{False}$, thus this node has two children: Node 5 and Node 6. The column 9 of the matrix A_3 , which is the 11th column of the original matrix A_0 , has the maximum weight. Then

$$\text{Remove_0}[A_3, 9] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} = A_5,$$

and

$$\text{Remove_1}[A_3, 9] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} = A_6.$$

None of the Rules 1-5 can be applied on the matrix **Remove_0** $[A_3, 9]$ and **Remove_1** $[A_3, 9]$. Therefore, the labels of the children of Node 3 are

$$\lambda_5 = (A_5, \emptyset, \{8, 9, 11\}),$$

$$\lambda_6 = (A_6, \{11\}, \{8, 9\}).$$

Node 4. Its label is λ_4 . The updated lower and upper bounds are $\text{Lower_Bound} = 3$ and $U = \text{Upper_Bound} = 3$. Since $\text{Lower_Bound} = U$, $\text{Test_Leaf} [\lambda_4, U] = \text{True}$, thus this node is a *leaf* of the search tree.

Node 5. Its label is λ_5 . The updated lower and upper bounds are $\text{Lower_Bound} = 3$ and $U = \text{Upper_Bound} = 3$. Since $\text{Lower_Bound} = U$, $\text{Test_Leaf} [\lambda_5, U] = \text{True}$, thus this node is a *leaf* of the search tree.

Node 6. Its label is λ_6 . The updated lower and upper bounds are $\text{Lower_Bound} = 3$ and $U = \text{Upper_Bound} = 3$. Since $\text{Lower_Bound} = U$, $\text{Test_Leaf} [\lambda_6, U] = \text{True}$, thus this node is a *leaf* of the search tree.

Therefore, the minimal hitting set is $\{6, 11, 12\}$ which produced by Node3. Figure 6 shows the search tree of this example.

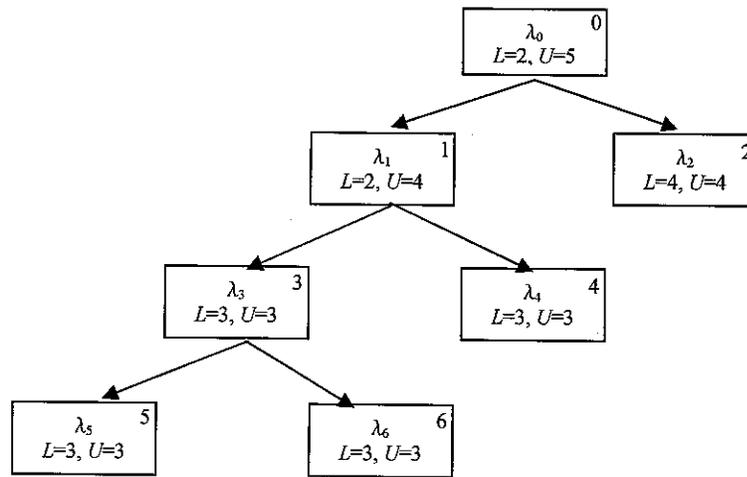


Figure 6. The search tree

7. 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 more than 10 times speedup over the standard algorithms.

Acknowledgement

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

References

- [1] 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.
- [2] A. Fijany, F. Vatan, A. Barrett, M. James, and R. Mackey, An advanced model-based diagnosis engine, *Proc. 7th Int. Symp. On Artificial Intelligence, Robotics and Automation in Space*, May 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*, 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* 79, 45–51, 2001.
- [10] J. de Kleer and B. Williams, Diagnosing Multiple Faults, *Readings in Model-Based Diagnosis*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.