https://ntrs.nasa.gov/search.jsp?R=19880006224 2020-03-20T07:38:56+00:00Z

55-61
N88-15606 116707 15 p.

1987

NASA/ASEE SUMMER FACULTY RESEARCH FELLOWSHIP PROGRAM

MARSHALL SPACE FLIGHT CENTER
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

INVESTIGATION OF CANDIDATE DATA STRUCTURES AND SEARCH ALGORITHMS TO SUPPORT A KNOWLEDGE BASED FAULT DIAGNOSIS SYSTEM

Prepared by:	Edward L. Bosworth, Jr.
Academic Rank:	Assistant Professor
University and Department:	The University of Alabama in Huntsville Computer Science Department
NASA/MSFC:	
Laboratory:	Systems Analysis and Integration
Division:	Space Systems
Branch:	Spacelab Payload Integration
NASA Colleague:	Michael S. Freeman
Date:	September 11, 1987
Contract No:	The University of Alabama in Huntsville NGT-01-008-021

ABSTRACT

The focus of this research was the investigation of data structures and associated search algorithms for automated fault diagnosis of complex systems such as the Hubble Space Telescope (HST). Such data structures and algorithms will form the basis of a more sophisticated Knowledge Based Fault Diagnosis System. As a part of the research, several prototypes were written in VAXLISP and implemented on one of the VAX-11/780's at the Marshall Space Flight Center. This report describes and gives the rationale for both the data structures and algorithms selected. A brief discussion of a user interface is also included.

1. Introduction

This paper will discuss a candidate data structure and associated search algorithm to be used as the basis for a Knowledge Based Fault Diagnosis System. Such a system might also be called an Expert System. The first part of the paper will define the structure and indicate the ways in which it matches the requirements of the fault diagnosis problem. In the second part of the paper, the shortcomings of the algorithm are indicated and indications given of modifications needed in order for the fault diagnosis task to be approached efficiently.

2. Objectives

The objective of this research is to support the development of an Automated Fault Diagnosis System to be used on the Hubble Space Telescope while on orbit. The primary function would be to interpret the downlink health telemetry on the HST and assist the operators in hypothesizing possible system faults and in deciding the proper corrective actions to be taken. Ideally, such a system would monitor performance trends and predict the time at which component operating capability would be so far degraded as to require corrective action.

For the purpose of fault diagnosis, the orbital mission of the Hubble Space Telescope may be divided into two time periods. The first period, lasting about six to nine months, is called the Orbital Verification Mission. This is a time during which all of the systems in the HST will be checked out and during which particular attention will be paid to the possibility of system faults. This period will be characterized by a high volume of system diagnostic information which could well be fed into a Knowledge Based Fault Diagnosis System such as is the subject of this research.

The second and much longer time period in the operation of the HST is the operational phase during which the telescope is expected to be collecting a large amount of scientific data. During this time the normal mode for collection and reporting of system health data will be the low data rate telemetry; i.e., there will be a low volume of data from which to infer the system status. A Knowledge Based Fault Diagnosis System could well be used during this phase, especially if it had the ability to detect potential failures and to devise measurements to be made during a brief period of high data rate health telemetry in order to confirm or deny the hypothesis.

3. Design of the Data Structure and Search Algorithm

The efficiency of a computational solution to any interesting problem depends, in general, on the choice of two items: a data structure appropriate to represent the structure of the problem and an algorithm which can operate efficiently on the selected data structure. One should note that these two are often quite interdependent: it is usually the case that the more appropriate the data structure, the less complex and more efficient the algorithm associated. This section will describe the selection of a data structure which fits the fault diagnosis problem quite well.

3.1 Choice of the Data Structure

As stated above, it is desirable to choose a data structure for at least two reasons: 1) natural description of the problem to be solved, and 2) the expected efficiency of the associated algorithm. For these criteria to be applied, one must first ask the fundamental question of what part of the problem is the more important to be represented efficiently.

In the fault diagnosis problem, the key feature to be represented is the interrelationship of the components in the structure being modeled. The usual method for representing this structure is a design schematic. There are two common data structures which will correspond naturally to a design schematic: a Tree and a Directed Acyclic Graph (DAG). Both of these are specializations of the general graph data structure.

The graph data structure comprises a collection of nodes, usually denoted as V, and a set of edges, usually denoted as E, connecting the nodes. The nodes are also called vertices, hence the symbol V for the set of nodes. One may name the edges by the nodes connected. Thus if

V = (V1, V2, ..., Vn) - a set of n vertices
E = (E1, E2, ..., Em) - a set of m edges

We might name an edge as Ei = (Vj, Vk); thus the edge (V1, V2) connects vertex 1 to vertex 2. The degree of a node is the number of edges connected to it.

A graph may be either directed or undirected; i.e., it may have edges which are directed or undirected. The edges in an undirected graph are similar to two way streets connecting intersections; thus if the edge (Vi, Vj) exists, so does the edge (Vj, Vi). Edges in a directed graph are similar to one way streets in that the existence of an edge (Vi, Vj) does not automatically imply the existence of (Vj, Vi). If both edges do exist in the graph, they must be called out explicitly in the edge list. Note that an node in a directed graph usually has a number of edges for which it is the first node in the pair and a different number of edges for which it is the second node in the pair describing the edge. The former number is called the "out-degree" of the node; the latter the "in-degree".

The directionality of a graph provides a natural mechanism for the representation of hierarchies normally found in design schematics. Thus the edges in a directed graph might represent the relationship "subcomponent". Note that if V1 represents a component in the design schematic and V2, V3, V4 represent its subcomponents, this fact may be naturally expressed by the edge list (V1, V2), (V1, V3), (V1, V4). Here the directionality represents the hierarchy; in particular V1 is not a subcomponent of V2. Note that in this representation, the out-degree of a node is the number of immediate subcomponents.

A path in a directed graph, denoted here as P (i, j) is a list of edges in the edge set E which connects Vi to Vj. In a design schematic the existence of a path from Vi to Vj would imply that the component represented by Vj is a subcomponent of Vi. Thus if the edge list for the graph included (V1, V2), (V2, V5), and (V5, V9), one could construct the paths P (1, 5), P (1, 9), and P (2, 9). Note that edges in the edge list are not usually called paths although they are such in the strict sense.

A cycle in a directed graph is a path of more than one edge which starts and ends on the same node. A graph is said to be acyclic if there are no cycles in it. A directed graph which is acyclic is called, naturally enough, a Directed Acyclic Graph or DAG. Note that the DAG structure is a natural representation of a design schematic because the absence of cycles implements the requirement that no component be a subcomponent of itself.

There are two specializations of a Directed Acyclic Graph which should be considered for representing the design schematic: the tree and a "tree-like DAG". A tree is a DAG with two constraints, one of which is that there be a distinguished node, often called the "root node", which has an in-degree of zero. While it can be shown that a Directed Acyclic Graph must have at least one such distinguished node with in-degree zero, the DAG may have more than one such node. The "tree-like DAG" is a DAG with only one node of in-degree zero. This root node has a natural analogue in the design schematic; it is the entire system not broken into subcomponents.

The main difference between a tree and a tree-like DAG is that the nodes of the tree are constrained to have an in-degree of one whereas there is no such constraint on a tree-like DAG. This would correspond in a design schematic to the requirement that a component be a subcomponent of only one other component. There are components, such as the electric power, which are clearly subcomponents of many other components. Attempts to represent this in a tree structure would cause one node to be constructed for each occurrence of the component and thus lead to inefficiencies in the search process. These inefficiencies are caused by the fact that a fault search algorithm will examine each node in the data structure and thus examine a common subcomponent a number of times, even after its status has been established. The data structure of choice is thus a tree-like Directed Acyclic Graph.

3.2 Choice of an Associated Data Structure

One of the main objectives in the use of a tree-like DAG is the avoidance of multiple representations of the same component in the design schematic. This becomes a concern only when a component is to be added to the DAG. The question: Is it already in the DAG?

In order to develop an answer to the above question, one must look more closely at the structure of a node in a graph. Typically the node has an ID and a label. The ID might be thought of as the "variable name" used in the computer program to reference the node. The label might be thought of as the "schematic name" of the node; i.e., the name of the component in the design schematic which the node represents. Thus, in the above example, node V1 might represent the Pointing Control System. The node ID is "V1". The label is "Pointing Control System".

Suppose one wanted to determine whether or not to create a node with the label "Pointing Control System". It would be necessary to determine whether or not the DAG contained a node with the identical label and to create a new node only in the case that an existing node were not found. The most straightforward approach is extremely inefficient. This approach is to search the DAG every time a node with a new label is considered for insertion.

A more efficient approach to this problem is to create an associated data structure to maintain a list of the labels of nodes in the DAG. This list could be consulted more quickly than the DAG could be searched exhaustively for the label. The only constraint to this approach is that the DAG and associated data structure must be treated as a single abstract data structure with well defined constructor functions. Were either the DAG or the associated data structure manipulated separately, an inconsistency would probably arise and the DAG would become of little use.

3.3 VAXLISP Implementation

The Directed Acyclic Graph was implemented in VAXLISP as a COMMON LISP structure. The following code describes the node structure.

```
(Defstruct (Component
                (:conc-name Node-)
                :predicate)
"A node for representing a component in the design schematic"
    (Name Nil)              ;The name in the design schematic
    (Subcomponents Nil)     ; Note the default values.
    (Contained-In Nil)
    (Search-Seq  0))
```

The first slot in the structure contains the schematic name of the component represented by the node. In other words, this is the label of the node. The second slot will contain a structured list, discussed below, of the subcomponents of this component. The third slot contains a list of the components of which this component is itself a subcomponent. The search sequence entry is an integer used by the search algorithm to avoid excessive searching. Its use is also described below.

The associated data structure is a COMMON LISP Hash Table. The hash table is organized by (key, value) pairs. The key of this table will be the label or schematic name of each node. The value will be the ID of the node. Note that the node ID is used internally by the program and is not intended for communication with the user of the system.

The following COMMON LISP construct establishes a hash table of 197 entries as a global variable. The number 197 is chosen as a prime number with a value of about 200. The size of hash tables conventionally is set as a prime number with a value about twice the number of expected entries.

```
(Defvar *Component-List* (Make-Hash-Table: Size 197))
```

In order to understand how the hash table is used, one must understand the operation of the basic retriever function Gethash.

```
(Gethash Schematic-Name *Component-List*)
```

This function caused the hash table *Component-List* to be searched for an entry with a key given by the value of the variable Schematic-Name. If such an entry is found, the value (which here is the ID of the node associated with the schematic name) is returned. If no entry is found, the value Nil (here equivalent to logical FALSE) is returned.

One should also understand the general function Setf.

```
(Setf (Gethash Schematic-Name *Component-List*) Node-ID)
```

This function sets the entry (Schematic-Name, Node-ID) in the hash table. Note that in general, Setf takes as its first argument a retriever function and as its second argument a value to be given to the variable accessed by the retriever function.

As mentioned above, the Directed Acyclic Graph and Hash Table must be accessed as a single abstract data type. A typical function is that which creates a new node. It first checks the hash table to avoid making a duplicate. If it continues, it first updates the hash table and then creates the associated node.

```
(Defun Create-Component (Schematic-Name)
"Creates a structure node with a component with
a given name"

  (Unless (Gethash Schematic-Name *Component-List*)
    (Let
      ((Node-ID (Gensym "NODE-")))
      (Setf  (Gethash  Schematic-Name  *Component-  List*)
         Node-ID))
      (Set Node-ID
         (Make-Component: Name Schematic-Name)) )))
```

The basic structure of the above function is the UNLESS clause, which has as its skeletal structure

(Unless (Something)(Action))

The action form is to be executed if the first form returns Nil, which in this case means that the Schematic-Name is not found in the hash table. The action form works with a temporary binding of the variable Node-ID to a symbol returned by the LISP function Gensym. What we are doing here is creating a new variable name; e.g. NODE-2940, to be used as the ID of the node generated by the function Make-Component, which is the constructor function for the DAG.

3.4 Structure of Slots for Component Lists

Each node in the Directed Acyclic Graph has two slots containing lists of components. These slots are (Contained-In) and (Subcomponents). The structure of the Contained-In slot is a list of the form (Contained-In (V1 V2 V3)) which is merely a list of those components of which the given component is a subcomponent or to which it passes data.

The Subcomponents slot must contain more information than just a list of the subcomponents. This slot should also contain information indicating the dependence of the component on the functioning of its subcomponents. Basically, there are two types of subcomponent lists: AND (the default) and OR. The two lists types may be characterized as follows:

AND - A component modeled by an AND subcomponent list is not more functional than the least functional of its subcomponents. This criterion merely states the fact that most components depend on the proper functioning of all the subcomponents.

OR - A component modeled by an OR subcomponent list is not less functional than the most functional of its subcomponents. This criterion allows for the proper modeling of redundant subcomponents.

While one should note that there might exist real hardware components which have both redundant and non-redundant subcomponents, these can be modeled effectively as a collection of pure AND and OR nodes.

An additional feature thought to be important in the representation of subcomponent lists is an ability to express the dependence of a component on the proper functioning of each subcomponent. Thus each subcomponent is itself represented by a list of the form

(Component-Name Sensitivity-Factor)

Thus, a typical subcomponent slot might resemble one of the following:

(Subcomponents (AND (V1 100) (V2 80) (V3 55)))
(Subcomponents (OR (V5 95) (V6 75)))

3.5 Combination Rules for Subcomponent Functionalities

The functionality of a given component is obviously dependent upon the functionality of its subcomponents. The exact nature of this dependence is different for AND and OR nodes. The combination rules for each class of nodes according to the nature of the subcomponent lists will be discussed in this section.

The basic rule for functionality of an AND node is built around the minimum value function MIN. The formula is given by

F = MIN [A(f1,s1), A(f2,s2), ... A(fn,sn)]

where for each subcomponent we have fi = the functionality of that subcomponent si = the sensitivity factor describing the dependence of the component upon that subcomponent

The A function should follow these two rules

A(fi,0) = 1 implying that if the component does not depend on the proper functioning of that subcomponent the subcomponent should be combined as functional. This is an extreme case.

A(fi,1) = fi implying that complete dependence should use the functionality factor itself.

There are a number of candidates for the A function, but the following seems to be the best:

A(f,s) = 1 + (f - 1) * s

The reasons for selection of this function are:

1) There is no theoretical reason to prefer a more complex function.
2) This function shows the correct behavior.
3) The free parameter "s" can be chosen for each subcomponent and component pair so as to show the desired dependency over a reasonable range of functionalities.

The rule for functionality of an OR node is achieved by similar logic:

F = MAX [f1*s1, f2*s2,....fn*sn]

3.6 Search Algorithms

Having established the data structure for representing the components to be examined by a fault diagnosis algorithm, it is now time to discuss the design of an algorithm to search through the data structure and isolate the faulty component. This design actually has two such algorithms both built around the concept of a search sequence number.

Search sequence numbers are a generalization of the concept of node marking found in many graph and tree search algorithms. Node marks generally are thought of as Boolean variables having the values TRUE or FALSE. An alternate representation of the node mark would be a search sequence number having only the permissible values 0 or 1.

In a node marking scheme, one also has two search algorithms. The first and simplest visits all nodes in the structure and sets the value of the mark to FALSE or not visited. The second and more focused search visits and marks all unmarked nodes which meet a predefined search criterion. Note that this gives rise to an overhead roughly equal to the search time for a specific item.

In the search sequence approach, there is a global variable which counts sequentially the searches undertaken during the current session. This variable is passed to the search procedure as a parameter in its list of arguments. As the search procedure visits each node it checks that node's search sequence number. If the node's search sequence is equal to the current value of the sequence number used by the search procedure, the node is considered to have been searched previously by this invocation of the search procedure and the node in not further examined. Otherwise the node is marked with the current search sequence number and is evaluated for expansion.

The associated search algorithm, called SWEEP, functions by performing a simple Depth First Search of the data structure and resetting the search sequence number of each node to zero. After having done this, it resets the global search sequence variable to one. Note that this procedure is undertaken to reduce the overhead seen in the simple node marking algorithm. It order to implement this, one must pick a maximum search sequence (say 1000). For a maximum search sequence of 1000, the SWEEP algorithm is called only once per one thousand directed searches. This procedure is similar to using a module counting sequence, but is robust against the ambiguities caused in such schemes when the count exceeds the module base (here 1000) and reverts to a small number (1001 becoming 1). Although it is provable that the search context will allow a high probability of disambiguating such sequence numbers, this more robust approach will guarantee against them will very little additional computational effort.

The search algorithm is a Best First Search with iterative deepening. It is called with two parameters - a node ID and a search sequence number. At each level, the node is examined to see if it is marked with the current sequence number. Should it be so marked, the next node in the search priority list is examined. Should the node not be so marked, it is given the current search sequence number and examined. Part of this examination will be obtaining the subcomponent list and comparing the components in that list to those in the search priority list. Nodes seen in both list will be marked with a high search priority; i.e. moved to the front of the search priority list. The reason for this is the common observation that if two failed components have a subcomponent in common, then that subcomponent is quite suspect.

3.7 Search Heuristics

There is one fault in the above described search algorithm. That fault is due to the fact that the Directed Acyclic Graph being searched is based on the design schematic of the device under diagnosis.

The advantage of basing a search strategy on the design schematic is obvious. Such an approach allows a very efficient and focused search. This is even true when one allows for faults which commonly occur in the inter-component connections. One can easily write algorithms to assume that certain components are functional but are detached from the main circuit in that their output is becoming lost.

What cannot be handled efficiently by the algorithmic approach are those cases in which the device under diagnosis has components which do not show in the design schematic. A simple example of such a component is a bridging fault or short circuit, both of which represent a connection which is not present in the schematic.

In the algorithmic approach, the design schematic is used as a basis for focusing the search. If this focusing basis be lost, the algorithm will devolve into an exhaustive search of all components and consequently become extremely inefficient. It is for this reason that a heuristic approach to the search must be devised.

The primary use of search heuristics will be to maintain the focus of the search algorithm when it becomes obvious that the fault in the system cannot be explained under the assumptions imposed by the design schematic. These heuristics will make use of device design information and expert engineering judgement in order to postulate the most likely deviation from the design schematic. Although some work has been done in applying these heuristics to simple digital circuits, much work is yet to be done before applying them to devices of the complexity of the Hubble Space Telescope.

4. Conclusions and Recommendations

While it seems obvious that an automated fault diagnosis system would be of considerable benefit in the operation of the Hubble Space Telescope, it is also apparent that an algorithmically based fault diagnosis system will not be sufficiently sophisticated to fulfill the mission.

One of the major modifications which will be necessary to this research is the design of a heuristic to evaluate the list of subcomponents for each node expanded in order to select the node to be next examined. It is this heuristic which will be based on knowledge of the physical design of the component represented by the node being examined and which must make plausible inferences as to additional unrepresented circuit faults.

This research has identified a considerable amount of the algorithmic structure which must underlie a heuristically based fault diagnosis system. It is recommended that future research be undertaken in order to complete the algorithmic approach and extend it to the more satisfactory heuristic approach.

5. References

1. Davis, R.; Diagnostic Reasoning Based on Structure and Behavior, Artificial Intelligence 24 (1984) 347-410

2. Genesereth, M. R.; The Use of Design Descriptions in Automated Diagnosis, Artificial Intelligence 24 (1984) 411-436.

3. Keravnou, E. T. and Johnson L.; Competent Expert Systems, published by McGraw-Hill, 1986.

4. de Kleer, J. and Williams, B.C.; Diagnosing Multiple Faults, Artificial Intelligence 32 (1987) 97-130.

5. Reiter, R.; A Theory of Diagnosis from First Principles, Artificial Intelligence 32 (1987) 57-95.

6. Steele, G. L.; Common Lisp, the Language, published by Digital Press, 1984.