

Generating Minimum-Cost Fault-Free Rule Bases Using Minimum Spanning Trees

Nabil Arman

Palestine Polytechnic University, Hebron, Palestine

narman@ppu.edu

Abstract: *This paper presents a new approach/algorithm for generating minimum-cost fault-free rule bases, which have no redundancy, circularity, inconsistency, contradiction/conflict and unreachability. The approach makes use of minimum spanning trees to check a rule base for different kinds of faults. The rule base is represented using a directed weighted graph. The algorithm devises a spanning tree/forest of the underlying directed graph by treating the directed graph as an undirected graph, and checks for various faults and properties. The algorithm devises a new rule base (which is a subset of the original rule base) that is equivalent, in terms of its reasoning capabilities, to the original rule base, with the properties that the new rule base is free from redundancy and circularity and has the minimum cost. It also determines the set of rules that cause redundancy and circularity faults. After determining the new rule base, checking for the remaining faults, namely inconsistency, contradiction, and unreachability, can be performed in a straightforward manner using the generated structures.*

Keywords: *Rule Bases Fault Detection, Minimum Spanning Trees.*

Received: December 15, 2005 | *Revised:* June 01, 2006 | *Accepted:* December 15, 2006

1. Introduction

The development of efficient algorithms to verify rule-based systems against different kinds of faults within the context of large rule-based systems have attracted a large amount of research efforts due to the important role of rule-based systems in various application domains, including Expert Systems (ESs), Active Database Systems, and Information Distribution Systems (IDSs) to name a few [1-11]. Verification is important to ensure the high quality of rule-based systems and to achieve an acceptable level of performance of these systems.

The effects of faults may appear in the performance of rule-based systems. Such faults may cause incorrect or undesired actions. Sometimes, these effects may be harmless, such as redundancy that may cause the system's performance to be inefficient. On the other hand, contradiction faults may lead to incorrect conclusions. It is worth mentioning that some redundancy faults may be included intentionally to gain some performance, instead of going into a long chain process to reach some conclusion/goal. However, in such cases the designer must be conscious of the presence of such faults and their consequences from the practical point of view.

Many approaches and algorithms for fault detection have been presented and proposed in the literature. The Expert System Validation Associate (EVA) program was developed at Lockheed [11]. EVA program was used to check for rule redundancy, inconsistency and contradiction. A decision-table-based processor for checking completeness and consistency in rule-based systems was presented in [5]. The COVER tool was presented in [9]. The tool was designed to build upon the best features of earlier systems. It is used to check rules based on a subset of first-order logic. A Petri net based approach for verifying rule bases was presented in [1]. A transition directed graph (TDG), which represents rule sets, was presented in [4,6,11]. TDG was used in the development of a set of algorithms to detect inconsistency, contradiction, circularity, unreachability, and redundancy in chained inference rules. These programs employed different approaches for detecting some faults. Based on these approaches many automated tools have been developed and used to inspect a rule-based system for known potential faults. However, none of these automated tools and approaches considered rule costs in the verification process. Different rules may be applicable at a certain point in time, but these rules may have different costs. Therefore, certain rules may be preferred over others due to the cost consideration.

To explain the meaning of the costs associated with the rules, an IDS environment is considered. An IDS is a subsystem within a Command, Control, Communication, Computer and Intelligence (C4I) System that is employed to control message passing and communication between IDS nodes. A rule-based information distribution system is an IDS that uses distributed information distribution rules (IDRs) to control message passing (communication) between distributed nodes. Information Distribution Commands provide the capability to describe the handling of information in IDSs, which have commands called Information Distribution Commands (IDC). IDCs have the following structure:

If *condition* then *action*

where, *condition* is the criterion that must be satisfied so that the action is carried out, and *action* is the set of instructions to be executed. There are two basic actions that can be executed: *send* and *notify* [3]. While IDCs with notify actions are called triggers since their purpose is to provide automatic notification capabilities to other nodes, those with send actions are called distribution rules since their purpose is to control the information exchanged between nodes. The exchange of information has a cost function that may have many parameters, including the data transfer rate, communication bandwidth, the distance between nodes in the distributed environment, and so on.

In this paper, the principles behind the new approach are presented. Section 2 discusses rule-based systems faults. In section 3, the structure to be used in the algorithm is presented. Section 4 presents the new algorithm in detail and provides a complete example of how the algorithm builds the minimum spanning tree/forest. In section 5, detecting rule base faults is explained. Section 6 contains the computational complexity analysis of the algorithm. Finally, section 7 presents the conclusion.

2. Rule-Based Systems Faults

A taxonomy of faults that may appear in a rule base, and which can be detected by using verification algorithms, is presented in [7,8,9]. It includes:

Redundancy: Two rules conclude the same outcome from the same input data.

Subsumption: Two rules conclude the same outcome, but one has additional constraints, which may or may not be necessary. It is a specific kind of redundancy.

Contradiction/Conflict: Two rules conclude different outcomes from the same input data.

Inconsistency: An antecedent of one rule is mutually exclusive to the consequent of such rule/chain of rules.

Circularity: The rule base contains a cycle inference chain, which may cause a backward-chaining inference engine to enter an endless loop.

Unreachability: Unreachability occurs if there is no path between any two given vertices.

3. The Structure Used in the Algorithm

Many transformation techniques for rule bases have been suggested in the literature. Petri nets were described in [1]. In this approach, a rule base will be modeled as a Petri net where parameter-value pairs corresponding to places and rules are analogous to transitions. Then the transition/place relationship modeled in a Petri net can be summarized in the form of an incident matrix. Decision-table-based processors were presented in [5]. In this approach, a decision table is created from the rules in the rule base. A directed-graph based approach was presented in [6], where the rule base is modeled as a directed graph and the process of anomaly detection is reduced to reachability among nodes. A transition-directed-graph-based approach, which is similar to [6] is presented in [3,4]. In this paper, we use the transformation technique where the rule base is modeled as a directed graph. In this directed graph, nodes correspond to propositions and rule identifiers (in case a rule antecedent is a conjunction of propositions) and edges corresponding to the rules. Each rule has a weight that represents the cost of applying that rule.

4. The Spanning Tree Based Algorithm for Verifying Rule Bases

A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G . Formally, let $G=(V,E)$ be an undirected connected graph. A subgraph $T=(V,E')$ of G is a spanning tree of G iff T is a tree. An interesting property of a minimum spanning tree is that it represents the minimal subgraph G' of G such that $V(G')=V(G)$. By minimal, we mean the one with the fewest number of edges.

Representing a rule base as a weighted directed graph, a minimum spanning tree/forest of such a graph will be devised. Although spanning trees are generally obtained for undirected graphs, they still make sense for directed graphs. In our case, despite the fact that the underlying graph is directed, we treat that as an undirected graph with some kind of interpretation of edges that create cycles. A variation of Kruskal's algorithm is used, which is a greedy algorithm that builds a minimum spanning tree by maintaining a forest (a collection of trees). Initially, there are $|V|$ single-node trees. Adding an edge

merges two trees into one. The algorithm terminates when enough edges are accepted. It turns out to be simple to decide whether edge (u,v) should be accepted or rejected. The appropriate data structure/approach is the union/find algorithm. This approach, as presented in *MCST_Fault_Detection* algorithm in Figure 1, is of great importance to devise an equivalent rule base RB' , of m rules where $m \leq n$, to the given rule base RB , which has n rules, such that RB' will have the same reasoning capabilities as RB . Due to the fact that minimum spanning trees are not unique, such a devised rule base may not be unique. In this approach, rules are considered based on their costs.

The *set_union*($S,r1,r2$) procedure implemented by ($S[r2]=r1$) maintains the direction of the edges in the original directed graph, by using the straight forward implementation of the algorithm. The *find* procedure, as presented in Figure 2, determines the root of the set to which a vertex (e.g., x) belongs.

To determine whether an edge $\langle x,y \rangle$ creates a cycle in the directed graph (G) or undirected graph (G'), the procedure *find_path*, as presented in Figure 3, can be used to check if two nodes x and y are on the same path in a certain disjoint set S .

If x is reachable from y , then x and y are on the same path and adding an edge $\langle x,y \rangle$ does not create a cycle. However, it indicates that there is another path that connects x to y . Thus there is a redundancy fault pattern. On the other hand, if x is not reachable from y , then x and y are not on the same path and adding an edge $\langle x,y \rangle$ creates a real cycle. Thus, this is a circularity fault pattern.

The process of detecting various kinds of faults by formulating faults as reachability problems in the graph-based representation should be augmented/extended by a check of the in-degree of the rule identifier vertices that comprise a certain path in the fault patterns. Although the formulation gives a necessary condition for the existence of various kinds of faults in a rule base, the condition is not sufficient as long as rules with multiple antecedents are considered. To deal with this additional issue, we can compute the in-degree of the rule identifier vertices in the path(s) of the fault pattern to determine whether a certain fault satisfies the necessary or the sufficient conditions of representing a real fault. This information can be collected during the generation of the spanning tree/forest and thus does not represent an expensive computational step. In addition, in real-world rule bases, the number of redundant and circularity fault patterns is relatively small and is assumed to be a constant number. In generating the new rule base, the assumption is that Linear-Edge-Rule-Relationship holds for the graph representing the rule base. This property says that if the graph has m edges and the rule base has n rules, then $O(m)=O(n)$.

```

MCST_Fault_Detection(G, RB, T, RB', C, R)
initialize (S) /*S: Disjoint Set */
Sort rules in ascending order based on their costs
T={ }
for all rules r do
  for all edges comprising rule r do
    choose next edge (rule)  $\langle u,v \rangle$  with lowest cost
    delete  $\langle u,v \rangle$  from E
    u_set=find(u,S)
    v_set=find(v,S)
    if  $\langle u,v \rangle$  does not create a cycle in T then
      add  $\langle u,v \rangle$  to T
      edges_accepted= edges_accepted+1
      set_union(S,u,v)
    else
      if find_path(u,v,S)=='C' then
        add r to C
      else
        add r to R
      endif
    endif
  end for
end for
end MCST_Fault_Detection

```

Figure 1. *MCST_Fault_Detection* Algorithm

```

find(x,S)
if(S[x] <= 0) then
  return x
else
  return(find(S[x],S))
end if
end find

```

Figure 2. *find* Procedure

```

find_path(x,y,S)
while (S[x] != 0 & S[x] != y)
  x=S[x]
if (S[x] == y) then
  return 'R' /* A redundancy pattern */
else
  return 'C' /* A circularity pattern */
end if
end find_path

```

Figure 3. *find_path* Procedure

Example: Consider the following rule base (RB), where A,B,C, \dots etc. are propositions and the numbers in parenthesis after the rule identifiers are the costs:

$r1(5): A \rightarrow B$ $r2(9): B \rightarrow C$ $r3(9): C \rightarrow \neg A$
 $r4(16): B \rightarrow D$ $r5(30): \neg A \rightarrow C$ $r6(40): A \rightarrow C$
 $r7(50): F \rightarrow G$ $r8(58): C \rightarrow D$ $r9(60): D \rightarrow \neg A$
 $r10(60): C \rightarrow A$

The rule base is represented as a directed graph as shown in Figure 4 (a). The algorithm accepts $r1$ since it has the lowest cost and doesn't create a cycle as in Figure 4 (b). It also accepts $r2$ and $r3$ for the same reason as shown in Figure 4 (c). The algorithm accepts $r4$ and rejects $r5=\langle \neg A,C \rangle$, since it creates a cycle, determined by computing

$u_set=find(\neg A,S)=A$ and $v_set=find(C,S)=A$. Since $u_set=v_set$, then there is a cycle in the undirected graph. Starting from $\neg A$ we can reach C , thus $C=\{r5\}$. The algorithm also rejects $r6=\langle A,C\rangle$, since it creates a cycle, determined by computing $u_set=find(A,S)=A$ and $v_set=find(C,S)=A$. Since $u_set=v_set$, then there is a cycle in the undirected graph. Starting from A we can not reach C , thus $R=\{r6\}$. It also accepts $r7$ since it doesn't create a cycle. The algorithm rejects $r8=\langle C,D\rangle$ since it creates a cycle, determined by computing $u_set=find(C,S)=A$ and $v_set=find(D,S)=A$. Since $u_set=v_set$, then there is a cycle in the undirected graph. Starting from C we can not reach D , thus $R=\{r6,r8\}$. The algorithm rejects $r9=\langle D,\neg A\rangle$ since it creates a cycle, determined by computing $u_set=find(D,S)=A$ and $v_set=find(\neg A,S)=A$. Since $u_set=v_set$, then there is a cycle in the undirected graph. Starting from C we can not reach D , thus $R=\{r6,r8,r9\}$. Finally, the algorithm rejects $r10=\langle C,A\rangle$ since it creates a cycle, determined by computing $u_set=find(C,S)=A$ and $v_set=find(A,S)=A$. Since $u_set=v_set$, then there is a cycle in the undirected graph. Starting from C we can reach A , thus $C=\{r5,r10\}$. The final result of all these steps is shown in Figure 4 (d). Therefore, the rule base comprising rules $r1,r2,r3,r4$, and $r7$ is generated using our approach. This rule base is fault free and has the same reasoning capabilities as the original rule base with the minimum cost.

5. Detecting Rule Base Faults

The algorithm as presented in section 4 determines the set of redundant rules. It also determines the set of rules that cause cycles to be present in the rule base. Once these set of faults have been determined, it would be relatively simple to determine the rest of the well-known faults in a straightforward manner as follows:

(i) Inconsistency faults: Inconsistency occurs when an antecedent of one rule is mutually exclusive to the consequent of such a rule (or a chain of rules). This means that starting from one vertex (e.g., A) we can reach to its exclusive vertex $\neg A$. To check for this kind of anomaly, we first determine the set of exclusive vertices, and then we need only to check whether the exclusive vertices are in the same disjoint set and there is a path between them (using the procedure *find_path*).

Example: In the given example, A and $\neg A$ are exclusive vertices. A and $\neg A$ are in the same disjoint set. Since there is a path from $\neg A$ to A , then there is an inconsistency anomaly.

(ii) Contradiction faults: A Contradiction/Conflict occurs when two rules conclude different outcomes from the same input data. This means that starting from one vertex/proposition (e.g. A) we can reach to two exclusive vertices (e.g., C and $\neg C$). To check for this kind of anomaly, we first determine the set of exclusive vertices, and then we only need to check whether the exclusive vertices are in the same disjoint set and none of them is the root of the set. If they are in the same set and none of them is a root, then there is a contradiction anomaly, otherwise there is no contradiction anomaly.

Example: In the given example, A and $\neg A$ are exclusive vertices. A and $\neg A$ are in the same disjoint set. Since A is the root of the disjoint set, then this does not result in a contradiction pattern.

(iii) Unreachability faults: Unreachability occurs if there is no path between any two given vertices. To check for that, we first determine whether the two vertices are in the same disjoint set or not. If there are in the same set, we use the procedure *find_path* to determine whether there is a path between them, and in this case there is no unreachability anomaly, otherwise there is an unreachability anomaly. On the other hand, if two vertices are not in the same disjoint set, then we conclude (without using the *find_path* procedure) that there is an unreachability anomaly.

Example: In the given example, B and F are two vertices. They are in different disjoint sets, thus there is an unreachability anomaly in this case.

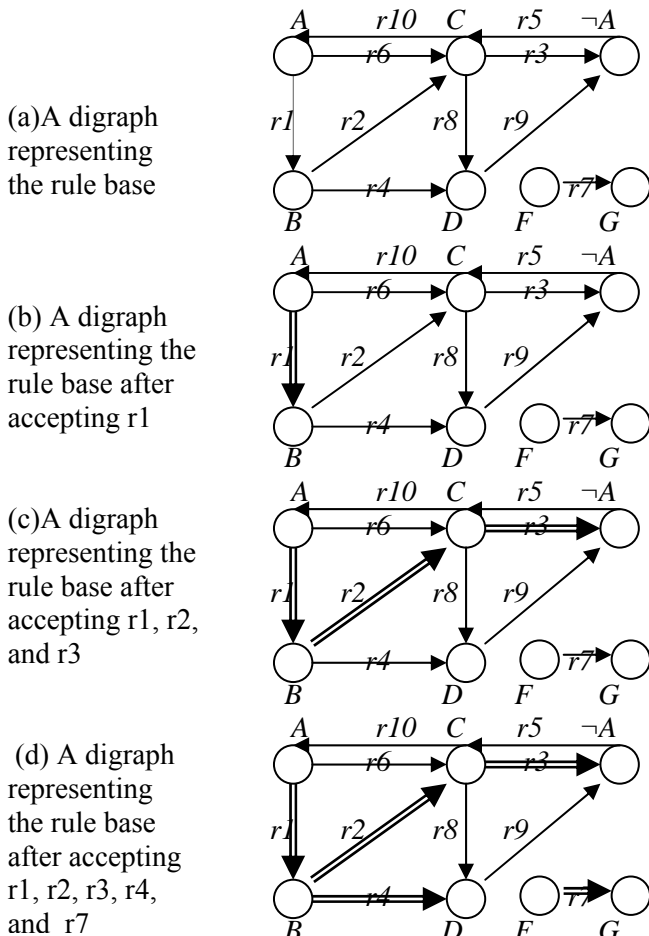


Figure 4. Execution of *MCST_Fault_Detection* on the rule base (RB), where double-lined edges/rules belong to RB' . Rules are considered based on their costs.

6. Computational Complexity

The spanning tree verification algorithm has a worst-case complexity of $O(n \log n)$, where n is the number of rules in the rule base. Basically, this is a variation of Kruskal's spanning tree algorithm. The first *for* loop is iterated for n times. The edge components of each rule is assumed to be constant, and the second *for* loop has a complexity of $O(1)$. The complexity of *find* is $O(\log n)$, using smart union algorithms (union-by-size approach). Thus, the worst-case complexity of checking for all redundancy and circularity faults is $O(n \log n)$.

Once the spanning tree and the data structures are obtained (which can be considered as a preprocessing step), the worst-case complexity of checking for inconsistency faults is $O(\log n)$, since this can be determined by using the *find_path* procedure, which has a complexity of $O(\log n)$ using smart union algorithms. The worst-case complexity of checking for contradiction faults is $O(1)$, since a path compression technique can be used to obtain the disjoint sets. Finally, the worst-case complexity of checking for unreachability faults is $O(n)$, which is dominated by the *find_path* procedure. The spanning-tree based approach represents a major improvement over Petri nets approach, which has a complexity of $O(n^2)$ for detecting inconsistency and redundancy. In addition, Petri nets approach has no rule cost consideration.

7. Conclusion

A minimum-spanning tree based approach for verifying rule-based system. The approach uses an algorithm that checks for redundancy and circularity in a rule base and generates a new rule base that is fault free, has the same reasoning capabilities as the original rule base, and has the minimum cost. Once the minimum spanning tree and the associated data structures are built, checking for the remaining set of faults, such as contradiction/conflict, inconsistency, and unreachability can be performed in a straightforward manner.

References

- [1] R. Agarwal, A Petri net based approach for verifying the integrity of production systems, *International Journal of Man-Machine Studies*, vol. 36, pp. 447-468, 1992.
- [2] N. Arman, D. Richards, and D. Rine, Structural and Syntactic Fault Correction Algorithms in Rule-Based System, *International Journal of Computing and Information Sciences*, vol. 2, no. 1, pp. 1-12, 2004.
- [3] Y. Hwang, Detecting Faults in Chained-Inference Rules in Information Distribution Systems, *Ph.D.*

Dissertation, School of Information Technology and Engineering, George Mason University, 1997.

- [4] Y. Hwang, and D. Rine, Algorithms to detect chained-inference faults in information distribution systems, *Proceedings of the 2001 ACM symposium on Applied Computing*, Las Vegas, Nevada, United States, pp. 679 – 685, 2001.
- [5] B. Gragun, and H. Steudel, A decision-table-based processor for checking completeness and consistency in rule-based expert systems, *International Journal of Man-Machine Studies*, vol. 26, pp. 633-648, 1987.
- [6] D. Nazareth and M. Kennedy, Verification of rule-based knowledge using directed graphs, *Knowledge Acquisition*, vol. 3, pp. 339-360, 1991.
- [7] A. Preece and R. Shinghal, Practical approach to knowledge base verification, *SPIE*, vol. 1468 Applications of Artificial Intelligence IX, 1991.
- [8] A. Preece, R. Shinghal, and A. Batarek, Principles and practice in verifying rule-based systems, *The Knowledge Engineering Review*, vol. 7, no. 2, pp. 115-141, 1992.
- [9] A. Preece, R. Shinghal, and A. Batarek, Verifying Expert Systems: A Logical Framework and a Practical Tool, *Expert Systems with Applications*, vol. 5, pp. 421-436, 1992.
- [10] A. Preece, S. Talbot, and L. Vignollet, Evaluation of verification tools for knowledge-based systems, *International Journal of Human-Computer Studies*, vol. 47, pp. 629-658, 1997.
- [11] R. Stachowitz, J. Combs, and C. Chang, Validation of Knowledge-Based Systems, *2nd AIAA/NASA/USAF Symposium on Automation, Robotics, and Advanced Computing for the National Space Program*, Arlington, VA, 1987.

Nabil Arman received his BS in computer science with high honors from Yarmouk University, Jordan in 1990, an MS in computer science from The American University of Washington, DC USA in 1997, and a PhD from the School of Information Technology and Engineering, George Mason University, Virginia, USA in 2000. He is an associate professor of computer science at Palestine Polytechnic University, Hebron, Palestine. Dr. Arman is interested in database and knowledge-base systems, and algorithms.