Knowledge Representation Report on Hitting Set Algorithm

Willem Blokland S1055337
Youri Joosten S1035806
Luke van Leijenhorst S1045958

June 13, 2022

1 Introduction

In this report we will look at our implementation of the hitting set tree algorithm. Before we do that however, we will first go over the existing code, the provided problems and we will discuss how we generated our conflict sets used to test our implementation. First we will go over the Diagnosis.scala file, next we will go over the Conflicts.scala file and discuss how we can use this file to generate conflict sets. We will discuss our implementation by going through the functions we created and finally we will evaluate our results.

1.1 Diagnosis.scala

First we will look at the Diagnosis.scala file where the three problems are given. The problems are Diagnostic Problems (DP) which consist of a set of observations (OBS) and a system (SYS). The observations are implemented as a list of formulae in the code. A system (SYS) consists of the components (COMP) of the problems and a system description (SD) which describes the expected/normal behaviour of these components. The components are simply a list of terms while the system description consists of a list of formulae. The three problems are visualised in figure 1 where components are represented by logic gates, observations by 0's, 1's and question marks if unknown. The system description is the default working of these AND and OR gates which is also formally described in the code:

```
val and_gate_ascii = hof"!x (and(x) & -ab(x) -> (in1(x) & in2(x) <-> out(x)))"
val or_gate_ascii = hof"!x (or(x) & -ab(x) -> (in1(x) | in2(x) <-> out(x)))"
```

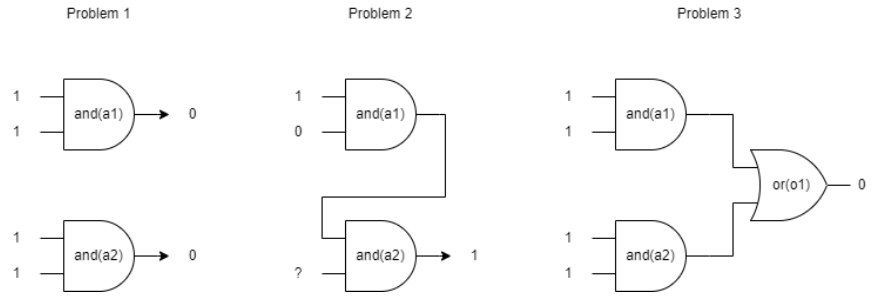


Figure 1: The three diagnostic problems in diagram form

1.2 Conflicts.scala

1.2.1 Explanation

This file contains the theorem prover that can be used to generate the conflict sets. It consists of a lot of helper functions and the main loop for the prover in the functions $tp()$ and $tpf()$.

The $tpf()$ takes both the diagnostic problem and a set HS which contains the set of components from COMP which are assumed to be defect (i.o.w. $HS \subset COMP$ where, if $x \in HS$, x is abnormal). Then, after extracting the individual elements (SD, COMP, OBS) from the diagnostic problem, the $tp()$ function with the parameters $HS, SD, COMP, OBS$ is called which brings us to the main function for the prover.

In this main function the goal is to find malfunctioning components. To do this, the prover checks whether the system behaves as expected (or to be exact, it checks whether there is no contradiction to be derived when assuming normal behaviour) by taking the union of the system description, the set of non-defect (normal) components and the observations (in set notation: $SD \cup HS \cup OBS \not\models \bot$). The set of normal components is simply calculated by taking the full set of components (COMP) and subtracting the set HS.

If $\bot$ CAN'T be derived, that means that there are no contradictions found and thus the empty set is returned for the conflict sets. If $\bot$ CAN be derived, that means a contradiction was found so there is a defect component. From the proof that was used to derive this contradiction, the antecedents are extracted from the sequents after filtering out empty sequents and tautologies, and then finally, the abnormal components are extracted from these filtered antecedents. These abnormal components make up a conflict set.

1.2.2 Generated conflict sets

Next, we used this theorem prover to generate a conflict set for the three problems. For problem two we also provided a 'proof' by natural language that this is an actual conflict set:

- Problem 1: Conflict set: a1
- Problem 2: Conflict set: a1, a2
- Problem 3: Conflict set: o1, a1

For the second problem we show this is true by a proof by natural language:
If some set X is a conflict set, that means there is at least one $x \in X$ for which x is faulty. So from the conflict set of problem 2, we know that it must be true that a1 and/or a2 is faulty. If we look at the middle diagram in figure one, we see that the expected output of the first AND gate (a1) should be 0 if the component works properly. However, since the output of the second AND gate (a2) is one, that would mean that both inputs to a2 would have to be 1 assuming this component works properly. However, we just established this is not true if a1 is functioning properly. So we know that if we assume a2 to be a normal functioning component, a1 would be faulty. If we assume a2 to be defect, it is possible that a1 was functioning normally by outputting a 0, but since component a2 would be defect it could still output a one. So if we assume a1 to be functioning normally, a2 would be faulty. So the conflict set must contain both a1 and a2 to be sure that at least one component is faulty.

2 Implementation

Now we will go through our implementation by going through the functions we created in the same order they are called.

In the `main.scala` class a `val diagnosis` is created that receives the function `mainHTalgorithm()` which takes a diagnostic problem and returns a diagnosis as output.

`mainHTalgorithm()` starts by creating the hitting sets, for this we have created the function: `generateConflictSets()`. This function takes the model specifications (System Description, Components and Observations) and calls the function `tp()` in the Conflicts.scala class. This will assign the generated conflict sets to a variable that we use to call the theorem prover until failure. When all conflict sets have been created a list is returned where each conflict set is represented as a list of FOLterms.
Then the second function `makeHittingTree()` is called within `mainHTalgorithm()` which takes the list of conflict sets and an empty list. The function recursively creates a rose tree where each of the visited components is kept in the left side of the tuple and each of the children is kept on the right side of the tuple. A node is only added if the considered component is not already in the list of visited components. When all conflict sets have been entered into the tree, the tree is returned and the third function `gatherHittingSets()` within `mainHTalgorithm()` is called.
`gatherHittingSets()` takes the tree returned by `makeHittingTree()` and recursively reads the hitting sets of the leaf nodes of the tree. It uses a helper function named `ThreeDimConcat` which concatenates the three dimensional list into a two dimensional list and returns this two dimensional list. When `gatherHittingSets` has read off every hitting set it returns a list containing all hitting sets represented as a list of FOLterms.
The fourth function `getDiagnosis()` within `mainHTalgorithm()` takes the list of all hitting sets generated by `gatherHittingSets()` and extracts the minimal hitting sets from the list of hitting sets following the logic: $H' \subsetneq H$. So if the length of one of the hitting sets is smaller than one of the existing hitting sets, and all components of this smaller hitting set are contained in the larger hitting set, the larger hitting sets is a superset and thus can be removed. Finally, it returns a list of just the minimal hitting sets which is also the return value of the call to `mainHTalgorithm()`.

This type of consistency-based diagnosis algorithm has a lot of real world applications, from Boolean algebra to debugging software to finding which part of a machine such as a satellite might be broken [1]. A consistency based algorithm paired with other algorithms is able to solve complex problems such as in [2] [Onan, 2015, p.3] a consistency-based subset evaluation in combination with a fuzzy-rough nearest neighbor classifier and instance selection is used for automated diagnosis of breast cancer. Consistency-based subset evaluation and is a common type of feature subset selection which is a important problem in machine learning "which requires identifying an appropriate set of features from which a classification model can be built properly".

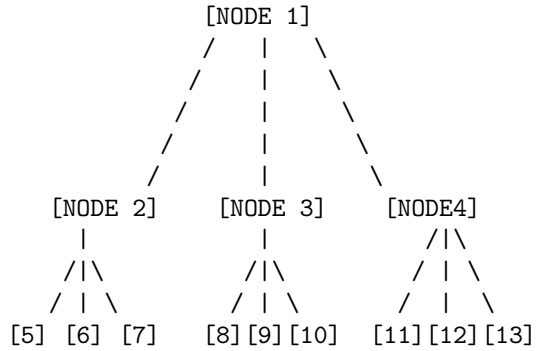
[1] https://doi.org/10.36001/ijphm.2016.v7i2.2363
[2] https://doi.org/10.1016/j.eswa.2015.05.006

3 Correctness evaluation

3.1 Results

3.1.1 Warm-up exercises

The function `makeTreeWarmup(b: Int, d: Int)` used for creating the hitting set tree takes two integer parameters b and d and outputs a rooted tree where every non-leaf node has exactly b children, every leaf node is at depth d, and every node is labelled by its depth. The tree created by this function should have exactly $(b^d - 1)/(b-1)$ nodes. When we call the function and set the parameters b & d both to three, we expect a tree with exactly $(3^3 - 1)/(3-1)$ nodes, which results in 13 nodes. This is also the output of the function `countNodes(tree: MTree[Any])` in our code. We can check if this is correct by creating the tree our self and counting the nodes:



3.1.2 Hitting tree implementation

The function `getDiagnosis(HS: List[List[FOLTerm]])` gets the number of minimal hitting sets and it takes HS, the list of all hitting sets as produced by `gatherHittingSets(hitting_tree: HittingTree[Any])`, as input parameter. The output should be a list of the minimal hitting set of HS. A hitting set H is minimal if there is no hitting set $H' \subsetneq H$. For example, $\{1,3\}$ and $\{2\}$ are minimal hitting sets of $\{\{1,2\},\{2,3\}\}$. This is also the case for the hitting sets: `List(List(o1), List(a1))` , for which the minimal hitting sets are: `List(List(o1), List(a1))`. For validating the hitting tree and the extracted hitting sets, we followed the steps of the algorithm as described on the slides by hand and got the exact same results as our implementation gave. So we are quite confident that our implementation is correct.

4 Task division

The collaboration in our group was good. We all attended the working groups together and it was easy to plan something among the three of us and stick to the planning. We all contributed to the code and report but the largest part of the code was done by Luke but he made sure we were all up to date on the code and were able to understand it.

5 References

Pill, I., Quaritsch, T., Wotawa, F. (2020). On the Practical Performance of Minimal Hitting Set Algorithms from a Diagnostic Perspective. In International Journal of Prognostics and Health Management (Vol. 7, Issue 2). PHM Society. https://doi.org/10.36001/ijphm.2016.v7i2.2363

Onan, A. (2015). A fuzzy-rough nearest neighbor classifier combined with consistency-based subset evaluation and instance selection for automated diagnosis of breast cancer. In Expert Systems with Applications (Vol. 42, Issue 20, pp. 6844–6852). Elsevier BV. https://doi.org/10.1016/j.eswa.2015.05.006