Hitting Set Algorithm in Prolog

Project Report

Kirsten Hagenaars - s1020609
Thomas Kolb - s1027332
Andrea Minichová - s1021688

March 17, 2021

1 Introduction

This report discusses our implementation of the hitting set algorithm in Prolog. We start by discussing an important part of the underlying theory, the conflict sets. Then we explain a function producing such conflict sets and its place within the algorithm. Further, we present our implementation in terms of the predicates used and the reasoning behind them. Then we assess the correctness on a few test cases. We then reflect on the code and discuss possible limitations and improvements. In the last section, we place the project within the scientific context and present some relevant scientific literature.

2 Hitting Set algorithm

2.1 Conflict sets

When we try to compute the consistency-based diagnoses it is often infeasible to compute every single possible hypothesis and find out which hypotheses are diagnoses. The hitting-set algorithm uses the concepts conflict sets and hitting sets to efficiently compute all the diagnoses for a given problem. A conflict set is defined as a set of components where if we assume that each component works normally leads to an inconsistency given a set of observations. Expressed in a formula:

$$SD \cup OBS \cup \{\neg Ab(c) | c \in CS\} \models \bot$$

This means that at least one of the components in the set `CS` should be abnormal, thus it will be present in one of our diagnoses. A hitting set is a combination of one element from each conflict set. If we have all conflict sets we can compute all the possible hitting sets, of which some will be our diagnoses.

To compute some conflict sets for the given problems we used the `tp` function, which outputs a conflict set. After compiling `diagnosis.pl` and querying

`Prolog` using the `tp` function with the `HS` argument being emtpy (i.e. no hitting set was given to the function), we observed the following conflict sets:

1. CS = $\{a_1\}$ for `problem1`,
2. CS = $\{a_1, a_2\}$ for `problem2`, and
3. CS = $\{a_1, o_1, a_2\}$ for `problem3`.

More conflict set are possible that were not given by `Prolog`. We have decided to make a proof for the conflict set $\{a_1, o_1, a_2\}$ of `problem3`.

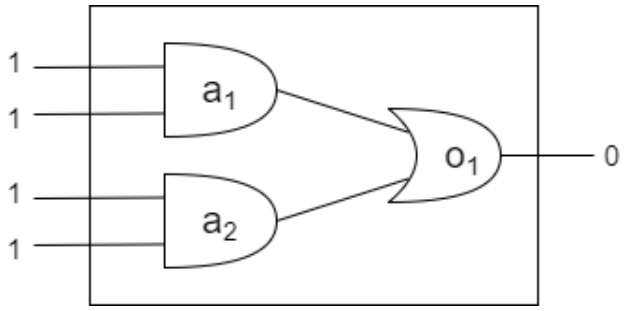


Figure 1: System for `problem3`

In figure 1 the system of `problem3` is visualized, the observations of this system are the following: components `a1` and `a2`, the `AND` gates, both have the first and second input `1`, the component `o1`, the `OR` gate, has the output `0`. For $\{a_1, o_1, a_2\}$ to be a conflict set the following must hold:

$$SD \cup OBS \cup \{\neg Ab(c) | c \in CS\} \models \bot$$

Thus given the system, its observations and the conflict set $\{a_1, o_1, a_2\}$, if we state that all the components in the conflict set are not defective, it should lead to an inconsistency. Since none of the components are broken we can predict that the output of `AND` gate components `a1` and `a2` are `1`. Looking at our system we know that this also means that the first and second input for the `OR` gate `o1` are both `1`. We predict that the output of `o1` will be `1`. This output of `1` is different than the observation of output `0`, meaning that we have an inconsistency. Which means that our set $\{a_1, o_1, a_2\}$ is indeed a conflict set. This also means that at least one of the components in this conflict set will be present in a diagnoses.

2.2 Use of the tp function

The tp function is a logical reasoning program that we used in our Prolog program that generates the minimal diagnoses of a given problem. There are two approaches for implementing the hitting-set algorithm. The first approach assumes that we already have the set F which is a set that includes all the conflict sets of a given problem. The second approach does not assume this and instead we iteratively generate new conflict sets given the specifications of the system using the tp function.

The tp function in Prolog has the following inputs:

1. SD: This is the system description of the problem, a set of formulas in first-order predicate logic describing the normal structure and behaviour of the system.

2. COMPS: List of first-order logic constants describing the active components of the system.

3. OBS: List of first-order predicate logic formulas describing the observed findings of the systems, the observed inputs and outputs of components.

4. HS: List of components which we assume to be functioning abnormally, we will not have to add these components to the output conflict set of the tp function since we already know that these components are functioning abnormally. Where in contrast we do not know which component(s) in the conflict set is functioning abnormally.

The output of the tp function is the conflict set CS. This is a conflict set of the system with specifications SD, COMPS and OBS where the components in HS are not included, since these components are already in the diagnosis that we are building. If the tp functions does not return a value for our variable CS (and thus returning *false*) this means that the tp function was not able to find any more components that would form a conflict set CS. In other words there were no components left that assuming they work normally would result in an inconsistency, meaning that we found all the components that form a diagnosis of the system.

We use the tp function in every iteration of the algorithm. In the hitting-set algorithm we produce a tree where all the leaves will include a diagnosis. We build the tree in a depth-first-search order. At each node of the tree we run the tp function once to retrieve a conflict set CS with the system specifications and the hitting set HS, which is essentially the path from the current node to the root node.

3 Explanation of our implementation

3.1 `makeHittingTree`

The `makeHittingTree(+SD,+COMP, +OBS, +LABEL, -TREE)` predicate takes a diagnostic problem and the label of the root of the desired hitting tree and returns the hitting tree. Both the nodes and leaves of such a hitting tree have a label, which is a set of components. These components are those that are currently already added to what will become the diagnosis. To be more clear, the components in the label of a node are the labels of the edges of the path from the root to this node, where the labels of the edges refers to the labelling of edges in the hitting trees provided in the slides of this course. This ensures that the labels of the leaves of the hitting tree hold the diagnoses.

The `makeHittingTree` predicate starts by assuming that the root of the desired tree is a node. It calls the `tp` predicate, where the label of the root is passed as to the `HS` parameter of `tp`, to receive a conflict set. For each element in this conflict set, we want to add a child to this node. This is done using the `makeChildren` predicate, which we elaborate on later. In case the label of the current node is already a hitting set, the `tp` predicate will fail instead of giving a conflict set. When this happens, we know that this current node is actually a leaf (this is our base case).

The `makeChildren(+SD, +COMP, +OBS, +LABEL, +CS, -CHILDREN)` predicate takes a diagnostic problem, the label of the current node and a conflict set and returns the list of children of this node. Note that the children of a node are represented by a list, this choice was made bacause each node may have any amount of children. For each component in the given conflict set, we create a child. The label of this child is the union of the label of its parent and the component from the conflict set it corresponds to. For each child, a call to the `makeHittingTree` predicate is made to recursively create a hitting tree from each child.

3.2 `gatherDiagnoses`

The `gatherDiagnoses(+TREE, -D)` predicate takes a hitting tree and returns the corresponding set of diagnoses. As explained before, the labels of the leaf nodes of the hitting tree are the possible diagnoses of the diagnostic problem. All this predicate needs to do is to retrieve the labels of all the leaf nodes.

Recall that every node contains a list of children. When `gatherDiagnoses` is given a node, we recurse on all of its children, since we need to reach all leaf nodes. This functionality is achieved by creating two cases for `gatherDiagnoses`, one where the given node has a single child and one where the given node has multiple children. When there is a single child we simply recurse on this child. When there are multiple children, the diagnoses resulting from recursing on

these children are combined using Prologs `append` function.

When `gatherDiagnoses` is given a leaf, we can immediately return its label. This is our base case.

3.3 `getMinimalDiagnoses`

The `getMinimalDiagnoses(+D, -MD)` predicate takes a set of diagnoses and returns a set of minimal diagnoses, in other words, it removes the supersets from a set of sets. As suggested, we searched online for inspiration on how to implement this functionality. We were not particularly inspired by what we found, so we decided to just implement this ourselves.

The `getMinimalDiagnoses` predicate iterates over all elements in `D`. It starts by checking whether there exists a subset of the current set (`SET`) in the set of elements following `SET` in `D`, which is done by the `existsSubset` predicate. If this is the case, we don't add `SET` to the minimal diagnoses. If this is not the case (if `existsSubset` fails), we backtrack to a different case of `getMinimalDiagnoses` where we do add `SET` to the minimal diagnoses. Afterwards, we use the `removeSupersets` predicate which removes the supersets of `SET` from the set of elements following `SET` in `D`. Note that `removeSupersets` only removes strict supersets of `SET` and not duplicates, since this is already taken care of by not adding `SET` to the minimal diagnoses if `existsSubset` succeeds.

We now further clarify how this predicate works. A set S needs to be removed when there exists a set T which is a subset of S. In our implementation, if S precedes T, S is not added to the resulting set because T is found by the `existsSubset` predicate. If T precedes S, then S is removed by the `removeSupersets` predicate, where removing actually means not to add it to the list of minimal diagnoses.

3.4 `main`

Finally, the `main(+SD, +COMP, +OBS, -MD)` predicate acts as a main function. It takes a diagnostic problem as input (`SD, COMP, OBS`) and returns a set of minimal diagnoses of this problem (`MD`).

This predicate starts by retrieving the hitting tree (`TREE`) from the given diagnostic problem by using the `makeHittingTree` predicate. Note that the empty set is passed as the `LABEL` parameter. This is done because this value corresponds to the label of the root of the tree, and since no components have been added to the diagnoses yet, this corresponds to the empty set for the root node.

After retrieving the hitting tree, this tree is passed to the `gatherDiagnoses` predicate, which returns the list of diagnoses (`D`). This list of diagnoses is then

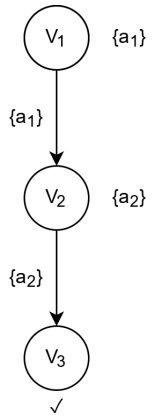
passed to the `getMinimalDiagnoses` predicate, which returns the list of minimal diagnoses (`MD`). After applying the three predicates, we use the cut operator. This is done to ensure that Prolog will only search for one set of minimal diagnoses for a given diagnostic problem (we cannot use `;` for Prolog to continue searching). We want this behaviour because there only exists one correct set of minimal diagnoses for a given diagnostic problem (bearing in mind that sets are unordered).

4 Tests

Given are four different diagnostic problems for testing and evaluation of the program. For each problem, we present the corresponding hitting tree inspired by the hitting trees in the slides of this course. Then we relate it to the hitting tree given by the `makeHittingTree` predicate, as well as the diagnoses obtained from `gatherDiagnoses` and the minimal diagnoses from `getMinimalDiagnoses`. Finally, we explain why the output is correct. This is to demonstrate that the program works for these diagnostic problems.

4.1 Problem 1

The hitting tree for `problem1`:



The corresponding output:

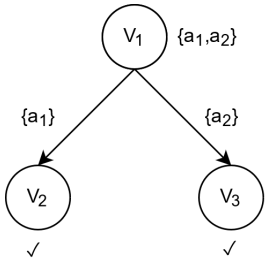
```
TREE = node([node([leaf([a1, a2])], [a1])], []),
D = MD, MD = [[a1, a2]].
```

This problem is characterized by two disconnected AND gates which produce faulty outputs. Since the gates are disconnected, the conflict sets are two singleton sets, each containing one of the gates. The `tp` function, therefore, first returns set $\{a_1\}$, and in the next iteration set $\{a_2\}$ This means that the tree can only have one branch at a time and, subsequently, depth of 2. The diagnosis is

then equivalent to this one path in the tree. Since the diagnosis only contains one set, the diagnosis is the minimal diagnosis. It is, therefore, trivial that our program returned the correct output for the problem.

4.2 Problem 2

The hitting tree for `problem2`:



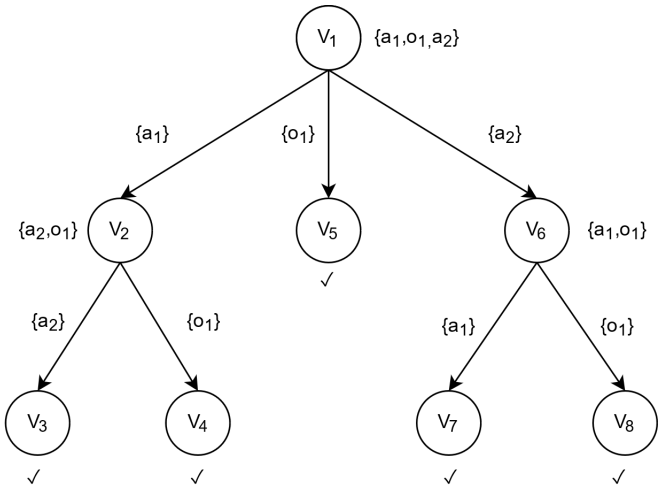
The corresponding output:

```
TREE = node([leaf([a1]), leaf([a2])], []),
D = MD, MD = [[a1], [a2]].
```

This problem is similar to the previous problem but in this case, the output of the first AND gate is the input of the second AND gate. This gives one conflict set which contains both gates, meaning that the root node has two children and the depth is 1. Following the paths from the leaves to the root, we get two diagnoses, $\{a_1\}$ and $\{a_2\}$. Since the diagnoses are disjoint, it is equivalent to the minimal diagnoses. Again, it is trivial that our program returned the correct output for the problem.

4.3 Problem 3

The hitting tree for `problem3`:



The corresponding output:

```
TREE = node([node([leaf([a1, a2]), leaf([a1, o1])], [a1]), leaf([o1]),
       node([leaf([a2, a1]), leaf([a2, o1])], [a2])], []),
D = [[a1, a2], [a1, o1], [o1], [a2, a1], [a2, o1]],
MD = [[o1], [a2, a1]].
```

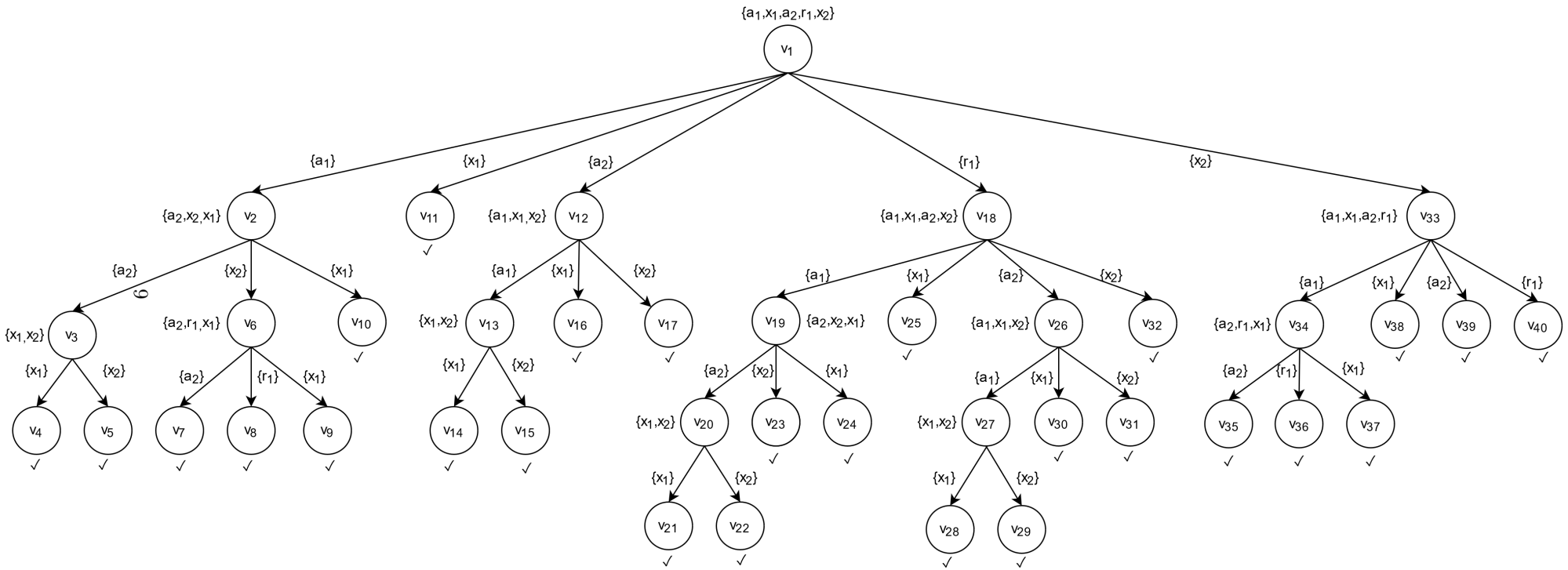
The diagram of this problem can be seen in Figure 1. There are three conflict sets, one containing all components, one excluding gate a_1 and the other excluding gate a_2. The tree was composed by making a node off of each conflict set, which can be seen in the tree diagram, as well as in the code version of the tree. It can be verified that the diagnoses was collected by following a path from each node towards the root (as already mentioned, the intermediate path is saved inside the label of each node and leaf). Comparing diagnoses to the minimal diagnoses, we can see that all supersets were removed meaning that the program returned the correct result.

4.4 Full adder

The hitting tree for `fulladder` can be found on the next page.

The corresponding output:

```
TREE = node([node([node([leaf([a1, a2, x1]), leaf([a1, a2, x2])],
       [a1, a2]), node([leaf([a1, x2, a2]), leaf([a1, x2, r1]),
       leaf([a1, x2, x1])], [a1, x2]), leaf([a1, x1])], [a1]),
       leaf([x1]), node([node([leaf([a2, a1, x1]), leaf([a2, a1, x2])],
       [a2, a1]), leaf([a2, x1]), leaf([a2, x2])], [a2]),
       node([node([node([leaf([r1, a1, a2, x1]), leaf([r1, a1, a2, x2])],
       [r1, a1, a2]), leaf([r1, a1, x2]), leaf([r1, a1, x1])], [r1, a1]),
       leaf([r1, x1]), node([node([leaf([r1, a2, a1, x1]),
```


{a1,x1,a2,r1,x2}
v1
{a1}
{x1}
{a2}
{r1}
{x2}
{a2,x2,x1}
v2
v11
{a1,x1,x2}
v12
{a1,x1,a2,x2}
v18
{a1,x1,a2,r1}
v33
{a2}
{x2}
{x1}
{x1,x2}
v3
{a2,r1,x1}
v6
v10
{a1}
{x1}
{x2}
{x1,x2}
v13
v16
v17
{a1}
{x1}
{a2}
{x2}
v19
{a2,x2,x1}
v25
{a1,x1,x2}
v26
v32
{a1}
{x1}
{a2}
{r1}
{a2,r1,x1}
v34
v38
v39
v40
{x1}
{x2}
{a2}
{r1}
{x1}
v4
v5
v7
v8
v9
v14
v15
{a2}
{x2}
{x1}
{x1,x2}
v20
v23
v24
{a1}
{x1}
{x2}
{x1,x2}
v27
v30
v31
{a2}
{r1}
{x1}
v35
v36
v37
{x1}
{x2}
{x1}
{x2}
v21
v22
v28
v29
9

```
       leaf([r1, a2, a1, x2])], [r1, a2, a1]), leaf([r1, a2, x1]),
       leaf([r1, a2, x2])], [r1, a2]), leaf([r1, x2])], [r1]),
       node([node([leaf([x2, a1, a2]), leaf([x2, a1, r1]),
       leaf([x2, a1, x1])], [x2, a1]), leaf([x2, x1]), leaf([x2, a2]),
       leaf([x2, r1])], [x2])], []),
D = [[a1, a2, x1], [a1, a2, x2], [a1, x2, a2], [a1, x2, r1],
    [a1, x2, x1], [a1, x1], [x1], [a2, a1, x1], [a2, a1, x2],
    [a2, x1], [a2, x2], [r1, a1, a2, x1], [r1, a1, a2, x2],
    [r1, a1, x2], [r1, a1, x1], [r1, x1], [r1, a2, a1, x1],
    [r1, a2, a1, x2], [r1, a2, x1], [r1, a2, x2], [r1, x2],
    [x2, a1, a2], [x2, a1, r1], [x2, a1, x1], [x2, x1],
    [x2, a2], [x2, r1]],
MD = [[x1], [x2, a2], [x2, r1]]
```

This problem is more intricate. The correctness of the tree can be confirmed by observing the tree for the problem and the tree constructed by the program. We find that the labels of the nodes and leaves of the tree produced by the program indeed correspond to the paths from that given node towards the root. This is also equivalent to the diagnoses, meaning that the program gathered the diagnoses correctly. Lastly, the minimal diagnoses are indeed the original diagnoses without any supersets.

5 Evaluation and reflection

5.1 Complexity

The `makeHittingTree` predicate has a time complexity of $O(V * X)$, where V is the amount of nodes/leaves in the resulting hitting tree and X is the time complexity of the `tp` predicate. This is because the `tp` predicate is run for each node/leaf in the hitting tree. Expressing the time complexity as a function of the amount of components in the diagnostic problem would lead to an upper bound which is way higher than the actual upper bound. Therefore we chose to express this as a function of the size of the tree. The `gatherDiagnoses` predicate has a time complexity of $O(V)$, where V is the amount of nodes/leaves in the given hitting tree, this simply corresponds to DFS in a tree. The `getMinimalDiagnoses` predicate has a time complexity of $O(n^2)$, where n is the amount of diagnoses in the given list (or equivalently, the amount of leaves in the hitting tree). We considered sorting the list to improve this, but even then we would have to compare all sets pairwise afterwards so we did not see a possibility of reducing the time complexity to $O(n\ log\ n)$. All together, the time complexity of the entire algorithm is either $O(V * X)$ or $O(n^2)$, depending on the value of X and also depending on the ratio of V and n (the amount of nodes/leaves and the amount of leaves).

5.2 Limitations & Improvements

The algorithm that we wrote works in a depth-first-search order, meaning that we expand a branch by adding components to the hitting set of that branch until a valid diagnosis is formed. When there are no more branches to be explored and the tree is complete we end up with a set of diagnoses. This set of diagnoses often contains sets that are subsets of other sets within the set of diagnoses. For example we can have a closed branch with `[X1]` as hitting set, and in a later iteration of the algorithm we have a branch with `[X2]` as hitting set. Let's say we want to explore this branch further and add `[X1]` to this hitting set. Our algorithm will now call the `tp` function with the hitting set `[X1, X2]`, but earlier in the algorithm we decided that `[X1]` is already a valid diagnosis. So this `tp` call could be avoided because we already know the answer of this call. An improvement could be to prune this branch to make sure `tp` is not called again. Implementing this in `Prolog` might be quite difficult and requires some serious restructuring of our code, but it could, depending on the complexity of the `tp` function, improve the total complexity of the implementation.

5.3 Difficulties

The main thing that held us back during the development was `Prolog`. Trying to understand how things work in the language and how programs should be written in the language was something we spent a lot of time on. We did eventually get a good intuition of how `Prolog` works and were able to implement the algorithm.

6 Scientific context

Apart from learning logical programming, this project also taught us more about the domain of consistency-based diagnosis. In this section, we will further discuss what is meant by consistency-based diagnosis, what was the process of research within this field and possible applications.

Consistency-based diagnosis is an instance of the AI approach towards model-based diagnosis. Model-based diagnosis serves to identify the faulty components in a system. This further allows for isolating faulty components, handling multiple faults, identifying repair actions, and automatically generating embedded software [4]. Given a model and an observed behaviour, inconsistency is found once the observed behaviour does not correspond to the predicted behaviour. This means that faulty components are present and can be defined as diagnosis. All components within the diagnosis are then faulty, while all the components not in diagnosis are working normal.

Consistency-based diagnosis serves to systematically identify all possible diagnoses and the minimal diagnoses. This is done through an algorithm which

makes use of conflict sets, hitting sets, and the hitting-set tree [3]. The foundations of this theory were laid in the late 70s and early 80s. For example in [6], a second generation expert systems were presented, where heuristics and deep reasoning were used for diagnosis and reparations of systems. However, the first solid theory and the diagnosis algorithm as we know it today was presented by Raymond Reiter in [8]. By the 90s, the theory was mature enough to sustain in large systems [1]. By that time, the theory and its application benefited from interaction with many disciplines within computer science and AI, such as knowledge representation, non-monotonic reasoning, qualitative reasoning, constraint problem solving, probabilistic reasoning, machine learning and many more [7]). For example, enhancing the consistency-based diagnosis with machine learning techniques were researched [2]. Another worthy of mention research attempts to mend the algorithm to fit application within physiology [5].

Finally, we would like to point out the importance of declarative programming languages (such as Prolog) within the domain of consistency-based diagnosis. Because models are best described by logical formalization, Prolog was often helpful in the research of the model-based diagnosis, for example, in proposing different instances of the algorithm.

7 Task division

All of the Prolog code was created collaboratively. We would call with the entire group where one person would type the Prolog code and screenshare and together we would program. This way of collaborating worked really well for us, mostly because Prolog has a steep learning curve and this way we learned a lot from each other and we could discuss each decision we made. Furthermore, this ensured that we all understand the entirety of the code. We had one such session where we did the exercises on tree structures and we had 4 such sessions to create the Prolog code for this project.

We did have a task division for writing this report in such a way that we all contributed equal parts to this report.

References

[1] Consistency-based diagnosis. https://www.infor.uva.es/~belar/Doctorado/4.1\%20CBD\%20using\%20General\%20Diagnostic\%20Engine\%20GDE.pdf. [Online; accessed 16-March-2021].

[2] Carlos Alonso, Juan J. Rodríguez, Belarmino Pulido. Enhancing consistency based diagnosis with machine learning techniques in the context of supervision. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.6098, 2004. [Online; accessed 16-March-2021].

[3] Girish Palshikar. Consistency-based diagnosis. https://www.researchgate.net/publication/281116622_Consistency-based_Diagnosis, 2001. [Online; accessed 16-March-2021].

[4] Johan de Kleer, James Kurien. Fundamentals of model-based diagnosis. https://www.sciencedirect.com/science/article/pii/S1474667017364674, 2003. [Online; accessed 16-March-2021].

[5] K L Downing. Physiological applications of consistency-based diagnosis. https://pubmed.ncbi.nlm.nih.gov/8358488/, 2004. [Online; accessed 16-March-2021].

[6] Luc Steels. Second generation expert systems. https://www.sciencedirect.com/science/article/abs/pii/0167739X8590010X, 2001. [Online; accessed 16-March-2021].

[7] Luca Console, Oskar Dressier. Model-based diagnosis in the real world: lessons learned and challenges remaining. https://www.ijcai.org/Proceedings/99-2/Papers/102.pdf, 1999. [Online; accessed 16-March-2021].

[8] Raymond Reiter. A theory of diagnosis from first principles. http://www.cs.ru.nl/~peterl/teaching/KeR/Theorist/reiteraij87.pdf, 1987. [Online; accessed 16-March-2021].