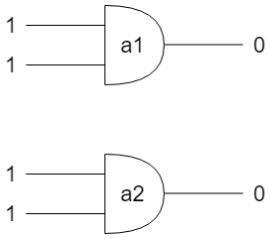
Representation and Interaction
Prolog Assignment - Report

Pleun Scholten s4822250
Elena Kreis s4841670

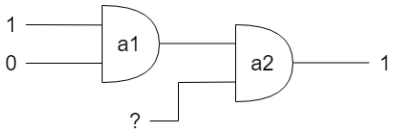
1 Problem description

A diagnostic problem is a tuple of a system together with a list of observed in- and outputs, where the observed output given the input does not match the predicted behaviour of the system. A diagnosis is given through a set of components that are assumed to be working abnormally and can explain the inconsistency of prediction and observation.
A possible way of obtaining the diagnoses of a system is to construct a hitting-set tree, the implementation whereof is the focus of this report.

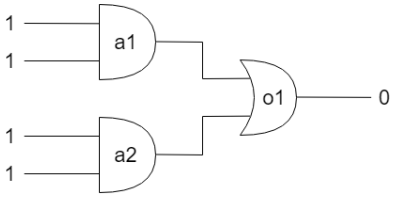
The following four diagnostic problems are given:



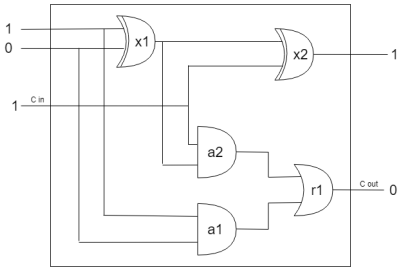
(a) Problem 1: Two disconnected and-gates.



(b) Problem 2: Two connected and-gates.



(c) Problem 3: A problem consisting of two and-gates and an or-gate.



(d) Problem 4: A full adder.

Figure 1: The four different test diagnostic problems.

2 Proof for Conflict Set

Calling `problem2(SD, COMP, OBS), tp(SD, COMP, OBS, [], CS).` returns the conflict set `CS = [a1, a2]`.

The definition of conflict sets is as follows:
Let $CS \subseteq COMPS$ be a set of components, then CS is called a conflict set iff $SD \cup OBS \cup \{\neg Ab(c) | c \in CS\}$ is inconsistent. This means that a conflict set is a set of components, that, if assumed to be working normally, cause an inconsistency with the observed output of the system.

Informal proof:

1. If both $a1$ and $a2$ are assumed to be normal, then the output of $a1$ should be 0, since $in1(a1) = 1$ and $in2(a1) = 0$ and $1 \wedge 0$ should evaluate to 0.
2. This means that no matter the $in2$ of $a2$, its output should be 0 as well, since $in1(a2) = out(a1) = 0$ and $\_ \wedge 0$ should evaluate to 0.
3. But this contradicts the observation that $out(a2) = 1$, and therefore the set $[a1, a2]$ is a conflict set.

3 The `tp` function

The `tp` function is used to generate a conflict set for a diagnostic problem. A diagnostic problem consists of a system definition (`SD`), a set of components (`COMP`), the observed in- and output (`OBS`) and an optional set of components assumed to be abnormal (`HS`). The idea behind the `tp` function is to find inconsistencies between observed in- and outputs and predictions of the systems, when all components are assumed to be working normally. Each inconsistency can be captured in terms of a conflict set, which is defined as $CS \subseteq COMP$ with $SD \cup OBS \cup \{\neg Ab(c) | c \in CS\} \models \bot$, so a subset of components, that if assumed to be working normally, causes an inconsistency between the observations and predictions.

The `tp` function works as follows:

1. The list of components that is assumed to work normally is specified, i.e. the list of components excluding the components in `HS`.
2. A theory is made, based on the system definition, the components and the observations.
3. This theory is put in clausal normal form, meaning there are no free variables, and all clauses, i.e. disjunctions of literals, are in conjunction.
4. A proof is found based on the negation of this theory. This corresponds to proving that the assumption that the subset of components is working normally leads to an inconsistency.

5. The components from the proof are extracted, meaning that the components which are used in proving the inconsistency are returned as the conflict set.

By accepting that certain components are abnormal, i.e. giving them as parameter `HS`, different conflict sets can be found.

4 The hitting-set algorithm

4.1 Representation of trees

The hitting-set algorithm was implemented using tree structures, as described in the assignment under section 2.
Trees are represented by a node in the format `node(Children, S, Hn)`, consisting of:

- A list of children `Children`, each child being a node itself.
- A label `S`, being either a conflict set or the constant `check`.
- A list of edge labels `Hn` containing all edge labels from the root node to that node

Specifically a hitting tree is then represented by its root node with `Hn`=[ ].

4.2 `makeHittingTree`

The function `makeHittingTree/4` is called with a diagnostic problem (`SD, COMP, OBS`) and returns a hitting tree (`HT`). It does so by calling a helper function `makeHittingTree/3`.
`makeHittingTree/3` calls `tp` for a new conflict set `S` and labels the current node with it. It then calls a helper function `getChildren/4` which will create a list of child nodes, each connected by an implicit edge labelled with an element of the conflict set `S` that the parent node was labelled with. `getChildren` will call `makeHittingTree/3` for each child to create the subtree of that child, resulting in the hitting tree being created depth-first.
A leaf node, labelled with `check` is created when the call to `tp` in the recursive case of `makeHittingTree/3` fails and the base case below is executed. This means that when no conflict set can be returned for the list of edge labels (`Hn`) so far, then we have arrived at a hitting set.

4.3 `gatherDiagnoses`

Since `Hn` specifies the complete path from root to the node, all diagnoses can be found by finding only the leaf nodes. In order to find all leaf nodes, two recursive steps need to be taken: one down the hitting tree, and one along the hitting tree, which corresponds to making a recursive call for every child the selected node has.

The function `gatherDiagnoses/2` corresponds to the recursive call down the hitting tree. The function `gatherChildDiagnoses/2` corresponds to the recursive call along the hitting tree, calling `gatherDiagnoses/2` for each child, and appending the found conflict sets. A conflict set is found when a leaf node is reached, which is indicated by the label `check`.

4.4 `getMinimalDiagnoses`

The minimal diagnoses are found by removing all supersets from the list of diagnoses. A superset is either bigger or equal in length to its subset. First, the list of all diagnoses is sorted in order of longest to shortest diagnosis. This happens in the `getMinimalDiagnoses/2` function. This is done because when the first diagnosis in the list is the longest, it will be evaluated against every smaller diagnosis.

In the helper function `getMD/2`, every element of the list is evaluated. If it is a superset of any other element behind it in the list, it gets removed from the list. This is why it is necessary to sort the list beforehand: the supersets are removed from the list. If the element is not, it is definitely a minimal diagnosis and is kept in the list.

The sorting that happens at the beginning of the `getMinimalDiagnoses` function is done using an adapted version of the insertion sort algorithm that was written by Roman Barták, found here *(last accessed 03-04-2019)*. The adaptation was made to sort on the length of lists in a list of lists instead of the elements of a list.

4.5 `solveDP`

The `solveDP` predicate acts like the main function of the program. It can be called with a diagnostic problem (`SD, COMP, OBS`) and returns the minimal diagnoses for that problem (`MD`).

The function calls all parts of the algorithm in the correct order, so first it calls `makeHittingTree`, it feeds the result of that into `gatherDiagnoses` and that result is in turn fed into `gatherMinimalDiagnoses`. The result of that function is returned as the output of the `solveDP` function.

5 Results

5.1 Problem 1

The call `problem1(SD, COMP, OBS), solveDP(SD, COMP, OBS, MD).` returns: `MD = [[a2,a1]]`. Therefore there is a single minimal diagnosis $[a1, a2]$.

This is correct because the and-gates are unconnected and both produce an incorrect output. Thus, it is a correct diagnosis that both gates are behaving abnormally simultaneously.

5.2 Problem 2

The call `problem2(SD, COMP, OBS), solveDP(SD, COMP, OBS, MD).` returns: `MD = [[a2],[a1]]`. Therefore there are two minimal diagnoses $[a1]$ and $[a2]$.

This is correct because the output is true, while one of the inputs for $a1$ is untrue. Because the and-gates are in parallel, one of the two gates must be behaving abnormally.

5.3 Problem 3

The call `problem3(SD, COMP, OBS), solveDP(SD, COMP, OBS, MD).` returns: `MD = [[a1,a2], [o1]`. Therefore there are two minimal diagnoses $[a1, a2]$ and $[o1]$.

This is correct because the observed output can be explained two ways. The first way would be that a1 and a2 are working correctly and have therefore both output 1, which means that the problem occurs in o1, which should now be outputting a 1 as well, but is not. This corresponds to the minimal diagnosis $[o1]$. The second explanation is that o1 is working correctly, but has gotten two zeros as input, which means that both a1 and a2 are not working correctly, which corresponds to the minimal diagnosis $[a1, a2]$.

5.4 Full Adder

The call `fulladder(SD, COMP, OBS), solveDP(SD, COMP, OBS, MD).` returns: `MD = [[a2,x2], [r1,x2], [x1]]`. Therefore there are three minimal diagnoses $[a2, x2]$, $[r1, x2]$ and $[x1]$.

This is correct because each minimal diagnosis corresponds to a possible explanation of the faulty outputs.
For the diagnosis $[x1]$ we assume that x1 is working abnormally, and therefore outputs a 0 instead of a 1. That means that the output of a2 is 0, the output of a1 is 0, and then both system outputs of out(x2) = 1 and out(r1)=0 are explained.
For the diagnosis $[a2, x2]$ the output of x1 would be 1, which would make the output of x2=0 but since it is faulty it outputs 1. The output of a2 should be 1, but since it is faulty it is 0, which together with out(a1)=0 explains the output of r1=0.
For the diagnosis $[r1, x2]$, similarly to above the output of x1 would be 1, which again should make the output of x2=0 but since it is also faulty in this diagnosis it outputs 1, which is what we observe. Here a2 correctly outputs 1 and the

output of a1=0. If r1 were working correctly it would output 1 but since it is included in the diagnosis the faulty output of 0 is explained.

6 Complexity Analysis

The complexity of `makeHittingTree` is dependant on whether the path to every leaf node in the tree intersects with every conflict set. The tree is always going to be at most as deep as the number of conflict sets. At the same time, there are only ever going to be as many conflict sets as there are incorrect outputs. In the worst case, every observation is incorrect, resulting in a maximum number of conflict sets. Consider O = $\{o_1, ..., o_n\}$ to be the set of outputs of a system. Then, in the worst case, the number of components involved in a wrong output o_i is equal to the total number of components minus those components whose output is another output $o_j \in \text{O}\backslash\{o_i\}$ of the system. Thus, the number of components involved in an output is equal to the number of components minus the number of outputs + 1. In relation to the dimensions of the hitting-set tree, this means that the depth of the tree is equal to the number of outputs, and the branching factor of the tree equal to the number of components minus the number of outputs + 1.
The number of a nodes in a tree with depth d and branching factor b is $\sum_{i=0}^{d} b^i$. The number of nodes in the tree would therefore be $\sum_{i=0}^{nrOutputs}(nrComponents - nrOutputs + 1)^i$.
The `makeHittingTree` function also makes use of the `tp` function, the complexity of which would also influence the complexity of `makeHittingTree`. `tp` is called for each node, therefore the final worst-case runtime complexity is $O((\sum_{i=0}^{nrOutputs}(nrComponents - nrOutputs)^i) \cdot Complexity(\texttt{tp}))$.

The `gatherDiagnoses` function goes through every node in the tree, in the process of finding all leaf nodes. Thus, its complexity is $O(n)$ where n is the number of nodes in the hitting tree. n is again calculated using the formula above. Thus, the runtime complexity is $O((\sum_{i=0}^{nrOutputs}(nrComponents - nrOutputs)^i)$.

Insertion sort has a worst case time complexity of $O(L^2)$ where L is the length of the list of diagnoses[1] . The number of diagnoses is equal to the number of leaf nodes, which in the worst case is equal to the branching factor of the tree to the power of the tree depth, i.e. b^d. As discussed above, the branching factor of the tree is equal to the number of components minus the number of outputs + 1, and the depth of the tree is equal to the number of outputs. The rest of the `getMinimalDiagnoses` function consists of the `reverse` function and the helper function `getMD`, which only loop through the same list of diagnoses once. Thus, the worst case time complexity would be $O((nrComponents - nrOutputs)^{nrOutputs^2}) = O((nrComponents - nrOutputs)^{2 \cdot nrOutputs}) = O((nrComponents - nrOutputs)^{nrOutputs})$.

[1]The time complexity of the sorting algorithms are taken from here: `http://bigocheatsheet.com/` (last accessed 05-04-2019)

The worst case time complexity of the main function `solveDP` is simply the largest time complexity of the three functions it calls. Thus, the worst case time complexity of `solveDP` is $O((nrComponents - nrOutputs)^{nrOutputs})$.

7 Limitations

All problems that were tested returned the list of minimal diagnoses instantaneously. However, it is not known if our algorithm scales to bigger problems, specifically bigger problems with more incorrect outputs. Especially since the complexity is in the range of $O(n^m)$, it is possible that for bigger problems, the runtime would be too long.

8 Improvements

The part of the program with the largest time complexity is the sorting function we used in the `getMinimalDiagnoses` function. The algorithm could be improved by using a faster sorting function, like heap sort, which has a worst case runtime complexity of $O(n \log(n))$.

9 Conclusion

Overall, the implementation of the hitting-set tree algorithm has been shown to work correctly and efficiently on small scale diagnostic problems. All diagnoses delivered for the tested problems were minimal and returned instantaneously.

Additional information

Time taken to finish the assignment
We spent three of the working group hours (around 6 hours in total) as well as around 2 hours outside of university hours to finish the code. For the report we spent roughly 4 hours. These estimates are per person.

Suggestions for changes
We believe that the programming in Prolog taught us a lot, but we do believe that the instructions for report were unclear. Mostly regarding how much detail was expected for the different sections, e.g. the proofs or the complexity analysis.
Furthermore there was not much information on how to choose between the two options provided for the implementation part of the assignment. We struggled slightly to choose which option would be best.