Knowledge-based AI

Assignment 2: Diagnose This

Group 63

Julian Dinnissen *s1135596*
Sten Nellen *s1151303*
Radboud University *November 14, 2025*

Contents

1 Introduction

Model-based diagnosis is a technique to identify malfunctioning components in complex systems, such as electronic circuits, by operating on conflict sets derived from observations of the system's behavior.

A **conflict set** identifies a set of components such that at least one of them must be faulty [1]. A **hitting set** is a set that has at least one element in common with every conflict set. A **minimal hitting set**, or **diagnosis**, is a hitting set from which no element can be removed without it ceasing to be a hitting set [1].

The efficiency of generating these diagnoses via the hitting set tree algorithm depends on the processing order of the conflict sets. This project explores this by implementing and experimenting with various search heuristics. Therefore, our research question is: **How do different heuristics affect the correctness of the diagnoses and the computational efficiency of the hitting set algorithm?** We answer this question by evaluating our implementation on several test circuits.

2 Methods

2.1 Specification

This research will implement the hitting set tree algorithm to find all hitting sets from a collection of conflict sets, which will then be brought down to the minimal hitting sets (diagnoses), using a standard algorithm. The hitting set algorithm will be evaluated with different heuristics for building the tree to evaluate how they affect the algorithm's runtime complexity. Furthermore, the hitting sets generated by each heuristic are checked by seeing if the resulting minimal hitting sets are the same across all the tested methods.

The components that will be implemented and evaluated are the following:

1. **Hitting Set Tree Algorithm**: An algorithm to find all hitting sets from a collection of conflict sets.

2. **Minimal Hitting Set Filter**: An algorithm to filter the hitting sets, keeping only the sets that are minimal.

3. **Heuristics**: Different heuristics for choosing the next conflict set to expand the tree. The performance of these heuristics will be compared based on the total number of nodes generated in the hitting set tree, which serves as a proxy for computational complexity.

2.2 Design

2.2.1 Hitting Set Algorithm

The hitting set tree algorithm finds a complete set of hitting sets that hit all conflict sets, with complete meaning that it contains at least all minimal hitting sets. Based on the given heuristic, the algorithm picks the first conflict set (the root), after which it creates a child node for every component in the conflict set. In each child, the remaining conflict sets are determined by removing every conflict set that contains the component. This process repeats recursively in every branch, meaning that a new conflict set is selected from the remaining conflict sets based on the heuristic, child nodes are added, and the remaining conflict sets filtered again. Branches terminate when there are no more conflict sets remaining. The algorithm uses Breadth First Search to expand the entire tree. Once the entire tree is expanded, the path from the root to a leaf node represents a complete hitting set.

Algorithm 1 Hitting Set Tree Generation

```
1: function GenerateHittingSetTree(conflict_sets, heuristic)
2:     initial_conflict_set ← HEURISTIC(conflict_sets)
3:     root ← HittingNode(initial_conflict_set, conflict_sets)
4:     queue ← [root]
5:     while queue is not empty do
6:         node ← queue.POP(0)
7:         if node is not terminal then
8:             for component in node.conflict_set do
9:                 remaining_cs ← {cs ∈ node.remaining | component ∉ cs}
10:                next_cs ← HEURISTIC(remaining_cs)
11:                child_node ← HittingNode(next_cs, remaining_cs, parent=node)
12:                node.ADD_CHILD(component, child_node)
13:                queue.APPEND(child_node)
14:    return root
```

2.2.2 Minimal Hitting Set

After the hitting sets have been generated, the collection is minimized to a set of minimal hitting sets using a separate algorithm. To make this step as efficient as possible, the hitting sets are first sorted by length in ascending order, after which each hitting set is compared to the already found minimal hitting sets (starting with an empty list). If the candidate is found to be a superset of any set already in the `minimal_hs` list, it is discarded. Otherwise, it is added to the list of minimal hitting sets. In this manner, each hitting set only needs to be compared to a shorter list of already found minimal hitting sets, instead of having to compare every hitting set against every other hitting set.

Algorithm 2 Minimize Hitting Sets

```
1: function MinimizeHittingSets(hitting_sets)
2:     sorted_hs ← SORT_BY_LENGTH(hitting_sets)
3:     minimal_hs ← []
4:     for candidate in sorted_hs do
5:         is_minimal ← true
6:         for minimal in minimal_hs do
7:             if minimal ⊆ candidate then
8:                 is_minimal ← false
9:                 break
10:        if is_minimal is true then
11:            minimal_hs.APPEND(candidate)
12:    return minimal_hs
```

2.2.3 Custom Circuits

We have created our own simplified circuit for testing purposes. This circuit helps us find any coding errors in the hitting set algorithm and can be seen in figure 1.

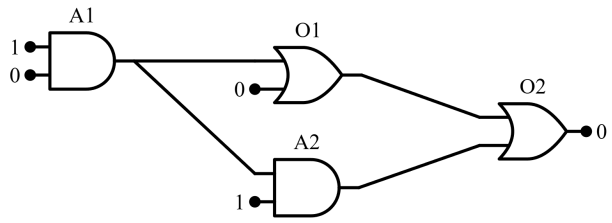

Figure 1: Circuit 8

2.3 Implementation

The algorithms were implemented in Python 3.12, making use of the object-oriented structure provided in the assignment repository. The data structures and algorithms were implemented without the use of external libraries, as per the assignment requirements.

The full code is provided with the supplementary materials, while this subsection provides an overview of design choices made for the implementation.

2.3.1 Tree Structure

The hitting set tree is implemented using a `HittingNode` class. Each instance of this class represents a node in the tree and stores essential information:

- `parent`: A reference to the parent node, allowing for traversal up the tree.
- `conflict_set`: The specific conflict set chosen for the node to branch on.
- `conflict_sets_left`: A list of conflict sets that still need to be hit by the node's children.
- `children`: A dictionary mapping each component of the node's `conflict_set` to its corresponding child `HittingNode` object.

This structure allows a recursive or iterative approach to build the tree and later retrieve the hitting set paths. For performance analysis, a class variable, `_nodes_created`, was included to count the total number of node instantiations during the tree's construction.

Besides just storing data, this class also contains parts of the logic for the hitting set algorithm, such as tree expansion and hitting sets retrieval, written in the methods `add_child` and `get_hitting_sets` respectively.

2.3.2 Algorithms

The main function running both the hitting set and minimal hitting set algorithm is `run_hitting_set_algorithm`. It initializes the root node and processes the queue, as described in Algorithm 1.

However, the code for tree construction is written in the `HittingNode` class itself, simplifying the main function. The main loop calls the `add_child` method on the current node for each component in its conflict set. As shown in Listing 1, this method is responsible for calculating the remaining conflict sets for the new child and then applying the given heuristic to select the next conflict set for that child to branch on. This design keeps the node-level logic self-contained within the class.

```
def add_child(self, branch: any, heuristic: callable):
    # Calculate the child's specific list of remaining conflicts.
    conflict_sets_left = [cs for cs in self.conflict_sets_left
                          if branch not in cs]

    # Apply the heuristic to find the best conflict set for the child.
    child_conflict_set = heuristic(conflict_sets_left)

    # Create the new child node with its own calculated data.
    node = HittingNode(child_conflict_set, conflict_sets_left, self)
    self.children[branch] = node
```

Listing 1: The `add_child` method, showing the heuristic application and state calculation in the node class.

Once the tree is fully constructed, the retrieval of the final hitting sets is initiated from the root node via the `get_hitting_sets` method. This method recursively traverses the tree's branches, collecting the components along each path from the root to a leaf node. Each complete path forms a single hitting set. Finally, the resulting collection of all hitting sets is passed to the standalone `minimize_hitting_sets` function, which implements the optimized filtering process described in Algorithm 2. The function returns a tuple containing a list of hitting sets and a list of minimal hitting sets.

2.3.3 Heuristics

The different heuristics are implemented as functions, that take in a collection of conflict sets and return one of these elements. In this test scenario, there will be 4 different heuristic functions implemented, that can each be passed to the hitting set algorithm as an argument. There are four custom heuristics with a varying degree of complexity. The first heuristic selects conflict sets based on length. It does this by returning the shortest conflict set that is in the remaining list. In the same way, another heuristic selects the longest conflict set. The third heuristic selects the middle conflict set of the list, and the final heuristic selects the conflict set based on the first conflict set that contains the most common element, shown in Listing 2 the code for our fourth heuristic.

```
def most_common_set_heuristic(conflict_sets):
    counter = {}
    for setthing in conflict_sets:
        for item in setthing:
            counter[item] = counter.get(item, 0) + 1
    sorted_dict = dict(sorted(counter.items(), key=lambda item: item[1],
       reverse=True))
    for c_set in conflict_sets:
        if list(sorted_dict.keys())[0] in c_set:
            return c_set
```

Listing 2: The `most_common_set_heuristic` function, showing the heuristic based on the most common element.

2.4 Testing

To test the various heuristics, an evaluation code was written in Python, which tests each heuristic on the same set of test circuits. For every combination, the number of nodes generated is noted. At the end, the results are summarized as total, average, median, minimum, and maximum number of nodes created for each heuristic. Lower numbers indicate more efficient heuristics. To ensure correctness, the program also checks if the number of minimal hitting sets found is equal for every result.

3 Results

Results of all the heuristics tested on every circuit, including our own custom circuit.

Heuristic	Total nodes	Average node	Median node	Min/max
`shortest`	69	8.62	6	3/21
`longest`	104	13	8	3/52
`middle`	71	8.88	6	3/23
`most common`	71	8.88	6	3/23

Table 1: Total, average, median and min/max node results

The heuristic `shortest` has the least amount of total nodes, followed by `most common`, `middle` and `longest`, with `most common` and `middle` having an equal amount of nodes. The same pattern emerges for the average amount of nodes created, `shortest`, `middle` and `most common` followed by `longest`. The median node of `shortest`, `middle` and `most common` is 6 and `longest` has a median node of 8. `shortest` performs the best in the minimum and maximum amount of nodes created, with a minimum of 3 and a maximum of 21. The `most common` and `middle` heuristics both have a minimum of 3 and a maximum of 23, slightly higher than the previous one. Lastly, the `longest` heuristic with a minimum of 3 and a significantly higher maximum of 52. Similar trends can be observed over all four evaluation results: `shortest` shows a slightly higher score compared to `most common` and `middle`. `longest` performs significantly worse than the other three. Notably, the shortest heuristic shows a significant efficiency gain, generating approximately 34% fewer nodes in total compared to the worst-performing longest heuristic.

4 Discussion

There is a key limitation that is not addressed in this project. There were eight circuits for the heuristic evaluation to be tested on. An increase in circuits could lead to a further distinction between the `middle` and the `most common` heuristic. Furthermore, we provided a clear choice for heuristic when using the hitting set algorithm: The `shortest` heuristic. Additionally, the difference between the `longest` heuristic and the other three can be explained by how the hitting set algorithm operates. When using the `longest` heuristic, the algorithm starts with the longest conflict set, which results in less flexibility in later iterations. The efficiency of the `shortest` heuristic likely comes from its greedy approach of resolving the most constrained part of the problem first. By satisfying a small conflict set, the algorithm starts out with a decision that leads to fewer immediate branches than it would have generated at each individual branch later on. While the `longest` heuristic performed poorly, one could theorize scenarios in highly sparse problem spaces where choosing the largest conflict set first might be beneficial. However, our results indicate that such a scenario is not typical for standard test circuits.

5 Conclusion and Reflection

In conclusion, we find that the heuristic that picks the best order of conflict sets is the `shortest`. Furthermore, all the heuristics show the same level of correctness in identifying hitting and minimal hitting sets.

We have once again divided the project into two parts: Introduction and Methods (Sten) and the rest (Julian). Additionally, we both worked together on the programming of the hitting set algorithm and we worked separately on the custom heuristic and circuit (Julian) and minimal hitting set algorithm (Sten). Our collaboration was managed through Git, and using VS Code Live Share for collaborative programming in the early stages.

6 References

[1] Johan Kwisthout and Nils Donselaar. *Knowledge Based AI - Lecture 4: Model-based reasoning*. Lecture Slides, Radboud University. Course SOW-BKI268. 2025.

A Demo Video

Link to demo video: `https://youtu.be/IcQi52aoe78`