Massachusetts Institute of Technology
16.412J / 6.834J – Cognitive Robotics – Spring 2023
Diagnosis: Implicates and Implicants for Constraint Programming

INTRODUCTION

An implicate is a logical consequence of the theory represented as a disjunctive clause. It is true for every assignment in which the original theory is true. For example, for the following theory,

$$\{(Switch = Off) \Rightarrow (Power = Low),$$
$$(Switch = On) \Rightarrow (Power = High),$$
$$(Power = Low) \Rightarrow (Light = Dark),$$
$$(Power = High) \Rightarrow (Light = Lit)\}.$$

$\neg(Switch = On) \vee \neg(Light = Dark)$ is an implicant because it is true for every assignment.

In particular, prime implicates are those where no subset of them is an implicate. This means that we if we remove any clauses of the disjunction, it will no longer be always true for every assignment. Since they succinctly, but comprehensively describe the theory, prime implicates can be used to decrease the size of a model's representation, allowing for memory and time efficiency.

When we define models, they many times have redundant clauses. In large models, these redundancies cause time and memory deficiencies. However, if we use the prime implicate generator algorithm, we can use prime implicates to fully describe the model, reducing the size of it significantly.

It is important to note that implicates are the negation of conflicts. This happens because, by negating the conflict, we resolve it, making assignments in the theory true. Conflicts are assignments to decision variables that cannot all hold true[1]. Minimal conflicts are conflicts with the smallest possible number of variable assignments, because their subsets are not conflicts. If we negate minimal conflicts, we get prime implicates!

PRIME IMPLICATE GENERATOR ALGORITHM[2]

The key to this method is that the algorithm continuously prunes the candidate tree to save both time and space. Instead of going through all possible domain values, it prunes anything that is redundant. That way we do not have to go over the full search space. We will not implement the algorithm in full in the homework, but we will go over some of the key steps in the algorithm to demonstrate how it works.

The main goal of the method is to find prime implicates. The model does this by finding minimal conflicts, and negating them to turn the minimal conflicts into prime implicates. And, as shown in the model overview diagram (Figure 1), the model generates minimal conflicts using two sub-

[1] Conflict-Directed A*: A Gentle Introduction – by Steve Levine
[2] Elliott, P. H. (2004). An efficient projected minimal conflict generator for projected prime implicate and implicant generation (Doctoral dissertation, Massachusetts Institute of Technology).

methods: a candidate tester and a candidate generator. As a reminder, candidates are assignments of values to the variables of a model.

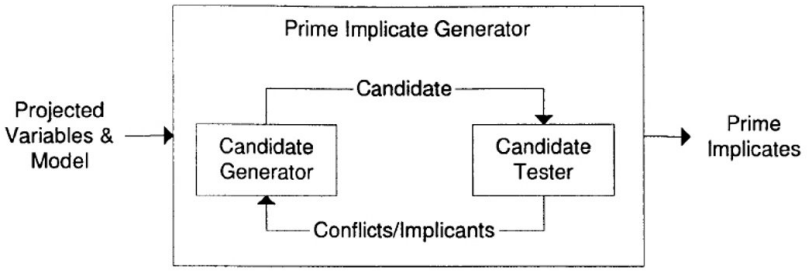

Figure 1: Overview of the Prime Implicate Generator Model

In the case of the mini problem set, we will not generate candidates. Those will be given to you. The candidates provided are represented in the tree below (Figure 2).

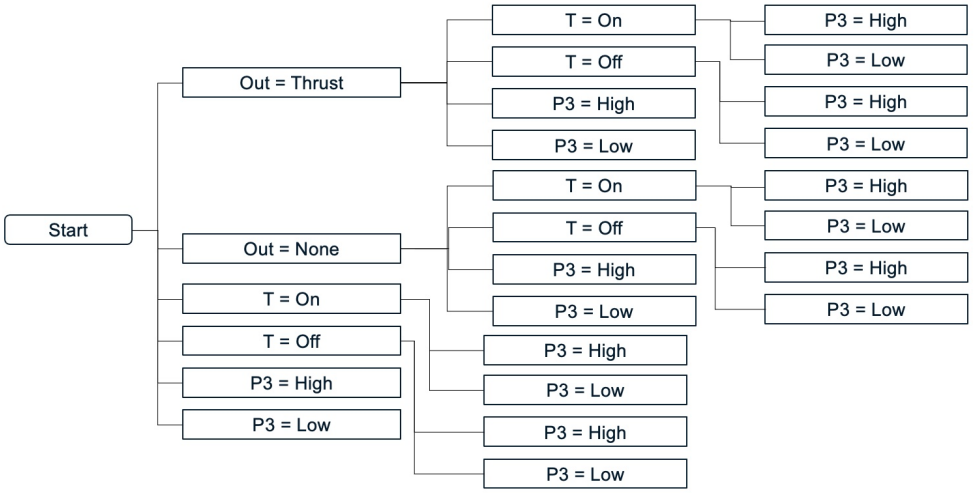

Figure 2: Candidate Tree Used in the Homework

Instead, in the homework we will test candidates to determine if they are valid, unsatisfiable, or satisfiable:

- Valid
  - A valid candidate is one where all of its remaining unassigned variables can take on any of their possible domain values. In other words, all the extensions of the candidate will be consistent with the model's theory and constraints. Valid candidates are equivalent to prime implicants.
  - The generator needs to keep track of existing prime implicants, so the valid candidate is passed back to the generator to help with pruning.
- Unsatisfiable
  - For unsatisfiable candidates, there exists no extension to any of the variables that is consistent with the theory. In other words, no matter how you try to extend the variables to other domain values, the result will always be inconsistent with the model's constraints.

    - Unsatisfiable candidates are minimal conflicts, which means it is one of the solutions that we are looking for. So, the minimal conflict is added to the solution set, and also passed back to the generator to help with pruning.
- Satisfiable
    - Formally, satisfiable candidates are defined as candidates that can be extended to become a superset of known implicants. However, for this homework, they can be considered to be all candidates that are not valid or unsatisfiable.
    - For this algorithm, it is useful to think of candidates as a set of partial variable assignments that can lead to either a conflict or a valid solution
    - Satisfiable candidates do not tell us anything concrete about minimal conflicts or prime implicants, but their data is still useful. So, they are converted to implicates and passed back to the generator

Now that we know the concepts, we can go into the details of how to check each candidate. In reality, the algorithm does not check this by looking at the children, but we will do in order to simplify the implementation.

First, we check for validity. A candidate is valid if none of it's children are conflicts. So, the pseudocode to check validity is the following:

is_valid(candidate, sat, children)

```
    if any is_conflict(sat, child):
        return False
    else
        return True
```

Next, we test for unsatisfiability. Here, we check to see if the candidate is a conflict. This means that also all of its children will be conflicts. The pseudocode to check unsatisfiability is the following:

is_unsatisfiable(candidate)

```
    if is_conflict(candidate):
        return True

    if any child is not a conflict:
        return False
```

The remaining candidates, those that aren't valid and aren't unsatisfiable, are satisfiable.

We now can build our prime implicate finder! Here is the pseudocode:

prime_implicate_finder(sat, candidateList)

```
    generatorAdditions = [ ]
    convertToImplicate = [ ]
    solutions = [ ]

    for candidate in candidateList:
        if is_valid(candidate):
```

Add *candidate* to *generatorAdditions*
else if *is_unsatisfiable(candidate):*
Add *candidate* to *solutions*
else:
Add *candidate* to *convertToImplicate*

return *generatorAdditions, convertToImplicate,* and *solutions*