DeepSequent

David Nadrchal

April 2023

Contents

1 Introduction

DeepSequent is a program for proving validity of QBF formulas using sequence calculus.

It uses several methods which will be discussed in more detail in further sections.

1.1 Terminology

- **Proof tree** - Tree graph with the assigned problem at its root. For every applicable action, there is an edge connecting the root to the state of the proof after applying this action. Deeper nodes are connected with their successors in the same way. Basically the game tree for the sequence calculus.
- **Proof state** - Every node of the proof tree
- **Leaf state** - Proof state whose successors have not been yet derived

2 General project structure

2.1 Files included

- **solver.py**: Main file. Calls the parser to create an object representation of the formula and that solves it with the corresponding method.
- **formula_parser.py**: Contains function *parse* which takes a file directory as its argument and returns its object representation. The parsing process is described in more detail later.
- **connectives.py**: Contains classes for operators and for literal.
- **rules.py**: Contains rules for deriving new proof states.
- **axioms.py**: Contains rules successfully closing the branches.

2.2 Building blocks

This subsection provides an exhaustive list of objects and classes with which DeepSequent operates.

- State: Corresponds to a single *proof state*. Contains a list of formulas of which the list one is the goal. Class definition can be found in *Implementation/rules.py*.
- Literal: This object will be renamed to 'Variable' in the near future, because that's what it is. It contains the value which identifies the corresponding variable. The class also has a static attribute used_tokens which comprises all variables present in the current problem. Class is defined in *Implementation/connectives.py*.

2.2.1 Connectives

The following classes can be found in *Implementation/connectives.py*.

- Universal quantifier: Has attribute *variables* which is the list of all all variables bounded by corresponding quantifier instance and also attribute *successor* containing the link for its uppermost successor.
- Existential quantifier: Has attribute *variables* which is the list of all variables bounded by corresponding quantifier instance and also attribute *successor* containing the link for its uppermost successor.
- And: Simple object with a single property *operands* containing list of its operands. And without operands is used as $\top$.
- Or: Simple object with a single property *operands* containing list of its operands. Or without operands is used as $\bot$.

- Xor: Simple object with a single property *operands* containing list of its operands.
- Not: Simple object with a single property *operand* a link to its operands.

2.2.2 Rules

The following classes can be found in *Implementation/rules.py*. Those classes have only static methods because there was no use in creating their instances.

All rules operate only on the uppermost connectives of state's formulas.

- Negation rules

  Eliminate double negation: This rule takes a formula and removes all negations from its top if there are even number of them. Otherwise it leaves it negated with a single negation.

  Negate goal: This rule is applicable only if the goal is not a false constant. It sets goal to be false and moves the previous one among assumptions negated.

  Negate a sequent: This rule is applicable only if the goal is a false constant. It sets goal to be a negation of a condition formula and removes this formula from the assumption.

- De Morgan's laws: Those are well known and are implemented in the most standard way so they hopefully do not need any special explanations.

  And

  Or

  Universal

  Existential

- Quantifier replacement rules: Those rules find all possible truth value assignments of variables bounded by the given quantifier, replace those variables in successor formula by corresponding truth constants and use it as an operand of a corresponding connective.

  Universal: Corresponding connective is *And*.

  Existential: Corresponding connective is *Or*.

- Assumption rules

  And: Splits formula into its operands.

  Or: For each operands creates a separate branch where this operand replaces original *Or* formula. All branches must be proven.

  Xor: For each operands creates a separate branch where this operand replaces original *Xor* formula. Other operands are added to assumption negated. All branches must be proven.

Universal: Instantiates. Removes the quantifier and replaces the variables bounded by it by other variables used in the other formulas of the proof.

Existential: Skolemnizies. Removes the quantifier and replaces the variables bounded by it by newly introduced skolemn variables.

- Goal rules

  And: For each operands creates a separate branch where this operand replaces original *And* formula. All branches must be proven.

  Or: For each operands creates a separate branch where this operand replaces original *Or* formula. This operation is indeterministic so proving one branch is enough.

  Xor: Set $\bot$ as the goal and create two branches, each with a formula corresponding to *Xor* negation. All branches must be proven.

  Universal: Skolemnizies. Removes the quantifier and replaces the variables bounded by it by newly introduced skolemn variables.

  Existential: Instantiates. Removes the quantifier and replaces the variables bounded by it by other variables used in the other formulas of the proof.

2.2.3 Axioms

Those are also well known and hopefully do not need any special commentary.

- Goal in assumption
- Contradiction in assumptions

2.3 Parsing .qcir files

The parsing procedure is as follows:

1. Check that given file is *qcir*.
2. Read the prenex: Store first quantifier in *Formula* variable.
3. Each time a quantifier is read, add it as a successor of the downer most *Formula* successor.
4. When *output(...)* is read, switch to post-prenex mode and save the output variable
5. Each time a new line is read, replace those of its variables that are stored as placeholders for more elaborate formulas by those formulas and store the result together with its placeholder.
6. When the output formula is met, set it as a successor of the downer most *Formula* successor.

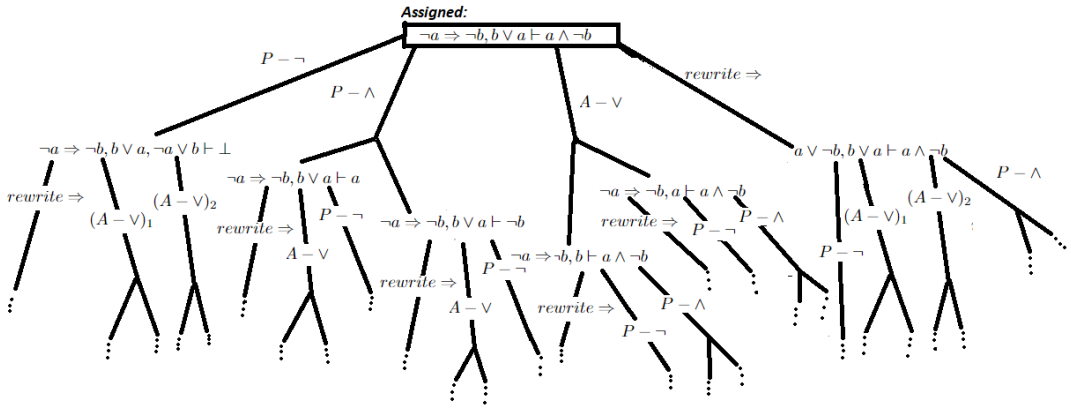
7. Create a new *State*, where the assumption is negation of *Formula* and goal is the $\bot$.

3 Brute force

3.1 Basic idea

For each leaf state, derive and store all of its successors. This approach could be complete in theory, but because of physical limitations it cannot be implemented in both sound and complete way.

Figure 1: Figure and code for search-space of brute-force search solving




```
# Algorithm: Brute force search

# Data:Graph node
# Return: proof found ? True : False
def brute_force(node):
    for action in node.possibleActions():
        succesor_state = node.applyAction(action)
        if succesor_state.fitAxioms():
            return True
        else:
            succesor_result = brute_force(succesor_state)
            if succesor_result: return True
    return False
```

3.2 Actual implementation

This subsection describes the logic of *brute_force* function which is recursively called to get a proof of the validity of a formula.

1. Check whether the given state fits the axioms. If it does, return the fulfilled axiom.
2. If not, save the state to the transposition table together with an empty string indicating that proof for this state has not been yet found.
3. Get all actions, that can be applied on the uppermost formulas of the state. Also, do some heuristics to reduce a search space a bit (e.g. by removing *Negate sequent* if there are only literals among the state's formulas).
4. For each action, get a list of formulas on which it can be applied and get the states derived by applying it.
5. If those states are already saved in the transposition table, get their proofs from there. Otherwise, apply *brute_force* to them.
6. If proofs were found for all derived states (or at least one in case of indeterministic rules applications), add the current state and action to the obtained proof. Save the proof to the transposition table and return it.
7. If no of the operations resulted in a proof or if no applicable action was found or if a recursion error occurred, return an empty string.

4 References

[1] Martina Seidl Charles Jordan Will Klieber. *Non-CNF QBF Solving with QCIR*. URL: https://fmv.jku.at/papers/JKS-BNP.pdf.

[2] Uwe Egly. *On Sequent Systems and Resolution for QBFs*. URL: https://publik.tuwien.ac.at/files/PubDat_214419.pdf.

[3] Lars Hupel Tobias Nipkow. *Logics Exercise*. URL: https://www21.in.tum.de/teaching/logik/SS18/sol03.pdf.