Project 2

Automated Reasoning

Hao Huang (NetID: hhuang40)

March 6, 2018

1. Basic Model Checking

In this section, we will present the main idea of *truth-table enumeration algorithm* (`TT-ENTAILS`) and several key points in implementing this algorithm.

1.1 Main Idea of TT-ENTAILS

The main idea in this algorithm is *enumeration* and *recursion*. First, all propositions in knowledge base (KB) and the query are converted into symbols. An empty model (no assignment) is built. Then, "true" and "false" values are assigned to each variable recursively. Once a value is assigned to a variable and this assignment is also satisfied by the current model, it will be added to the current model. The recursion continues until all variables are assigned (and also satisfied by the model), or no consistent assignment is found any more. The pseudocode is shown in Figure 1

function TT-ENTAILS?(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  $symbols \leftarrow$ a list of the proposition symbols in KB and α
  return TT-CHECK-ALL($KB, \alpha, symbols, \{\}$)

---

function TT-CHECK-ALL($KB, \alpha, symbols, model$) **returns** *true* or *false*
  if EMPTY?($symbols$) **then**
    if PL-TRUE?($KB, model$) **then return** PL-TRUE?($\alpha, model$)
    else return *true* // *when KB is false, always return true*
  else do
    $P \leftarrow$ FIRST($symbols$)
    $rest \leftarrow$ REST($symbols$)
    return (TT-CHECK-ALL($KB, \alpha, rest, model \cup \{P = true\}$)
      and
      TT-CHECK-ALL($KB, \alpha, rest, model \cup \{P = false\}$))

Figure 1: Pseudocode of TT-ENTIAL algorithm

As we can see, the function `TT-CHECK-ALL` is call recursively. During each call, an assignment is checked whether it is satisfied by the current model. The function `PL-TRUE` is specifically responsible for checking whether an assignment is hold within a model.

1.2 Kep Points in Implementation

To implement this algorithm, we add two classes, `TTModel` and `TTEnum`, into the provided framework.

1. TTModel
   This class implements the interface `Model`. It has a protected member attribute `assignment` which is a hash map mapping each symbol to its value ("true" or "false"). The `set()` and `get()` functions are used to add/change symbol values and retrieve symbol values. Two overload functions `satisfies(Sentence)` and `satisfies(KB)` will check whether the given sentence or knowledge base is satisfied by the current model. For more information, please refer to the source code `pl.sln.TTModel.java`.

2. TTEnum
   In this class, an `ArrayList` is utilized to store all symbols derived from a knowledge base and a query. The other part is translated from the pseudocode in Figure 1, which is straight-forward and easy to understand.

   However, one thing to notice is that parameters are passing by reference in Java. Therefore, in the recursive call in `TT-CHECK-ALL`, we should make a copy of the "model" parameter, pass these two objects with the same value into two `TT-CHECK-ALL` functions, as shown in Figure 2 and Figure 3. For more information, please refer to the source code `pl.sln.TTEnum.java`.

$$\begin{aligned}&\textbf{return}\ (\text{TT-CHECK-ALL}(KB, \alpha, rest, \boxed{model} \cup \{P = true\})\\ &\qquad\textbf{and}\\ &\qquad\text{TT-CHECK-ALL}(KB, \alpha, rest, \boxed{model} \cup \{P = false\}))\end{aligned}$$

Figure 2: The two "model"s are different objects with the same value

```
TTModel model_copy = new TTModel(model);
model.set(symbols.get(idx - 1), true);
model_copy.set(symbols.get(idx - 1), false);
```

Figure 3: Correpsonding Java snippet for Figure 2

2. Advanced Propositional Inference

In this section, we will present the main idea of `resolution` algorithm (`PL-RESOLUTION`) and several key points in implementing this algorithm.

2.1 Main Idea of PL-RESOLUTION

Suppose a knowledge base is represented by `KB` and a query is represented by α, This algorithm is used to test whether `KB` entails α (`KB` $\models \alpha$).

First, (`KB` $\models \neg\alpha$) is converted into CNF. For each pair of clauses in CNF, we test whether this pair contains complementary literals. If yes, we combine this pair of clauses together

and remove complementary literals to generate a new clause, and then add this new clause to the set if it is not contained in the set. The process continues until:

- No more new clause can be add, which means `KB` $\not\models \alpha$;
- An empty clause is produced after two clauses are resolved, which means `KB` $\models \alpha$.

Notice that this algorithm adopts *contradiction* idea. Pseudocode is shown in Figure 4

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses $\leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new $\leftarrow \{\}$
  loop do
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents $\leftarrow$ PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new $\leftarrow$ *new* $\cup$ *resolvents*
    if *new* $\subseteq$ *clauses* **then return** *false*
    clauses $\leftarrow$ *clauses* $\cup$ *new*

Figure 4: Pseudocode of PL-RESOLUTION algorithm

In this algorithm, the function `PL-RESOLVE()` returns the set of **all** possible clauses generated by resolving two argument clauses. One more thing to notice is that every step we only remove two complementary literals even if there are more complementary literals. Other parts of this algorithm is self-explained.

2.2 Key Points in Implementation

To implement this algorithm, we add one class, `PLProver`, which implements the interface `Prover` in the provided framework.

1. `PLProver`
   We use a hash set to represent clauses generated from a `KB` and a query. Another hash set is initialized to be empty to store all clauses yielded by resolution. For each pair of clauses from the `KB` and query, the function `PL-RESOLVE()` is called to carry out resolving process. In general, the translation from pseudocode to Java code is straight-forward.

   However, for some complicate examples, it takes extremely long time for this algorithm to generate an answer. To optimize this algorithm, we add one constraint before adding the generated "resolvents" to "new": the length of the generated

clause should not be longer than two original clauses. The reason is obvious: if we want to generate an empty set (if possible), the new generated clauses should become shorter and shorter for each step. Test shows that this constraint greatly accelerates running speed while still yields correct a result. Corresponding code snippet is shown in Figure 5. For more information, please refer to the source code `pl.sln.PLProver.java`.

```
if (len <= ci.size() && len <= cj.size()) {
    // add to resolvents
    resolvents.add(ci_copy);
```

Figure 5: Java snippet in `PL-RESOLUTION` algorithm

In addition, in order to make it easier for implementation, we add two constructors to `Clause` class: one is a default constructor, another is a copy constructor. This is a trivial modification to the provided framework, we will not explain it in detail here.

3. Samples

In this section, we will use six examples to test the implementation of two algorithms described above.

3.1 Modus Ponens

In this example, there are two sentences in knowledge base: P and $P \implies Q$, and the query is Q. We use both model checking and inference rule to check whether Q is true or not. The result is shown in Figure 6. It is a rather simple example, and for more information, please refer to the source code `pl.examples.ModusPonensKB.java`.

```
P
(IMPLIES P Q)


Test using model checking:
Q is true


Test using propositional inference:
Q is true
```

Figure 6: The result of Modus Ponens example

Note that the first several lines exhibit sentences in knowledge base. The remaining examples follow the same style.

3.2 Wumpus World (Simple)

In this example, there are seven symbols and each of them is represented by a capital letter (P and B) with two subscript. Five sentences are constructed according to the problem description. The query is whether $P_{1,2}$ is true or not. It is also a simple problem and more information can be found in the source code `pl.examples.WumpusWorldKB.java`. The result is shown in Figure 3

```
(NOT P1,1)
(IFF B1,1 (OR P1,2 P2,1))
(IFF B2,1 (OR P1,2 (OR P2,2 P3,1)))
(NOT B1,1)
B2,1


Test using model checking:
P1,2 is false


Test using propositional inference:
P1,2 is false
```

Figure 7: The result of Wumpus World (Simple) example

3.3 Horn Clauses

In this example, we devise five symbols and each symbol is represented by a string describing the attribute of the unicorn, as shown in Figure 8. The knowledge base contains four sentences derived from the problem description. We have three queries, and each of them is tested using both model checking and inference rule. The result is shown in Figure 9. More information can be found in the source code `pl.examples.HornClauseKB.java`.

```
Symbol myth = intern("Unicorn is mythical");
Symbol mor = intern("Unicorn is mortal");
Symbol mam = intern("Unicorn is mammal");
Symbol horn = intern("Unicorn is horned");
Symbol magic = intern("Unicorn is magical");
```

Figure 8: Symbols in Horn Clauses example

```
(IMPLIES Unicorn is mythical (NOT Unicorn is mortal))
(IMPLIES (NOT Unicorn is mythical) (AND Unicorn is mortal Unicorn is mammal))
(IMPLIES (OR (NOT Unicorn is mortal) Unicorn is mammal) Unicorn is horned)
(IMPLIES Unicorn is horned Unicorn is magical)


Test using model checking:
Unicorn is mythical : false
Unicorn is magical : true
Unicorn is horned : true


Test using propositional inference:
Unicorn is mythical : false
Unicorn is magical : true
Unicorn is horned : true
```

Figure 9: The result of Horn Clauses example

3.4 Liars and Truth-tellers

In this example, we have three symbols, $A/B/C$, and each of them represents a proposition that the corresponding person is truthful. There are two cases in this example, and in each case, the knowledge base consists of three sentences. Sentences are constructed using connectives. In each case we have to test three queries and each query concerns the truthfulness about each person. The result is shown in Figure 10 and the source code is `pl.examples.LiarTruthTellersKB.java`.

```
Case 4-(a):
(IFF Amy is truthful (AND Amy is truthful Cal is truthful))
(IFF Bob is truthful (NOT Cal is truthful))
(IFF Cal is truthful (OR Bob is truthful (NOT Amy is truthful)))


Test using model checking:
Amy is truthful : false
Bob is truthful : false
Cal is truthful : true


Test using propositional inference:
Amy is truthful : false
Bob is truthful : false
Cal is truthful : true

----------------------------------------------------------------

Case 4-(b):
(IFF Amy is truthful (NOT Cal is truthful))
(IFF Bob is truthful (AND Amy is truthful Cal is truthful))
(IFF Cal is truthful Bob is truthful)


Test using model checking:
Amy is truthful : true
Bob is truthful : false
Cal is truthful : false


Test using propositional inference:
Amy is truthful : true
Bob is truthful : false
Cal is truthful : false
```

Figure 10: The result of Liars and Truth-tellers example

3.5 More Liars and Truth-tellers

This is an extended example of the previous one. In this example, there are twelve persons. Therefore, we devise twelve symbols and each one is a proposition stating the truthfulness of a person. For example, `Symbol a = intern("Amy is a truth-teller.")`. The description of this problem provides twelve quotations from all persons, so the knowledge base contains twelve sentences, one per quotation. Propositional connectives are used in sentences. We also have twelve queries, and each of them concerns about the truthfulness of each person. The code structure (`pl.examples.MoreLiarsTruthTellersKB`) is similar

with the previous one. The result is shown in Figure 11.

```
(IFF Amy is a truth-teller. (AND Hal is a truth-teller. Ida is a truth-teller.))
(IFF Bob is a truth-teller. (AND Amy is a truth-teller. Lee is a truth-teller.))
(IFF Cal is a truth-teller. (AND Bob is a truth-teller. Gil is a truth-teller.))
(IFF Dee is a truth-teller. (AND Eli is a truth-teller. Lee is a truth-teller.))
(IFF Eli is a truth-teller. (AND Cal is a truth-teller. Hal is a truth-teller.))
(IFF Fay is a truth-teller. (AND Dee is a truth-teller. Ida is a truth-teller.))
(IFF Gil is a truth-teller. (AND (NOT Eli is a truth-teller.) (NOT Jay is a truth-teller.)))
(IFF Hal is a truth-teller. (AND (NOT Fay is a truth-teller.) (NOT Kay is a truth-teller.)))
(IFF Ida is a truth-teller. (AND (NOT Gil is a truth-teller.) (NOT Kay is a truth-teller.)))
(IFF Jay is a truth-teller. (AND (NOT Amy is a truth-teller.) (NOT Cal is a truth-teller.)))
(IFF Kay is a truth-teller. (AND (NOT Dee is a truth-teller.) (NOT Fay is a truth-teller.)))
(IFF Lee is a truth-teller. (AND (NOT Bob is a truth-teller.) (NOT Jay is a truth-teller.)))


Test using model checking:
Amy is a truth-teller. : false
Bob is a truth-teller. : false
Cal is a truth-teller. : false
Dee is a truth-teller. : false
Eli is a truth-teller. : false
Fay is a truth-teller. : false
Gil is a truth-teller. : false
Hal is a truth-teller. : false
Ida is a truth-teller. : false
Jay is a truth-teller. : true
Kay is a truth-teller. : true
Lee is a truth-teller. : false


Test using propositional inference:
Amy is a truth-teller. : false
Bob is a truth-teller. : false
Cal is a truth-teller. : false
Dee is a truth-teller. : false
Eli is a truth-teller. : false
Fay is a truth-teller. : false
Gil is a truth-teller. : false
Hal is a truth-teller. : false
Ida is a truth-teller. : false
Jay is a truth-teller. : true
Kay is a truth-teller. : true
Lee is a truth-teller. : false
```

Figure 11: The result of More Liars and Truth-tellers example

As we can see from the result, only Jay and Kay are truth-tellers and all the others are liars.

3.6 The Doors of Enlightenment

In the problem, there are two sub-problems: Smullyan's problem and Liu's problem. In both sub-problems, there are twelve symbols. The first four symbols represent four propositions stating that X. Y, Z and W are good doors, and the last eight symbols represent eight propositions stating that A to H are knights.

In Smullyan's problem, we have eight sentences consisting a knowledge base as described in the problem description. Biconditional connective is used in every sentences. We have four queries, and each is querying whether the corresponding door is a good door. The result is shown in Figure 12. We can see that only X is a good door.

```
Case 6-(a):
(IFF A is a knight. X is a good door.)
(IFF B is a knight. (OR Y is a good door. Z is a good door.))
(IFF C is a knight. (AND A is a knight. B is a knight.))
(IFF D is a knight. (AND X is a good door. Y is a good door.))
(IFF E is a knight. (AND X is a good door. Z is a good door.))
(IFF F is a knight. (OR (AND D is a knight. (NOT E is a knight.)) (AND E is a knight. (NOT D is a knight.))))
(IFF G is a knight. (IMPLIES C is a knight. F is a knight.))
(IFF H is a knight. (IMPLIES (AND G is a knight. H is a knight.) A is a knight.))


Test using model checking:
X is a good door. : true
Y is a good door. : false
Z is a good door. : false
W is a good door. : false


Test using propositional inference:
X is a good door. : true
Y is a good door. : false
Z is a good door. : false
W is a good door. : false
```

Figure 12: The result of Smullyan's problem in Doors of Enlightenment example

In Liu's problem, The first and the last sentences are the same with Smullyan's problem. In addition, for the fragment: "G: A and ... are both knights", we only add one sentence: `add(a)`; for the fragment: "G: If C is a knight, ...", we add nothing the the knowledge base, since we cannot derive any useful information from this sentence. As a consequence, there are only three sentences in the knowledge base in Liu's problem. The result is shown in Figure 13. As we can see, the result is as the same as Smullyan's problem.

```
Case 6-(b):
(IFF A is a knight. X is a good door.)
A is a knight.
(IFF H is a knight. (IMPLIES (AND G is a knight. H is a knight.) A is a knight.))


Test using model checking:
X is a good door. : true
Y is a good door. : false
Z is a good door. : false
W is a good door. : false


Test using propositional inference:
X is a good door. : true
Y is a good door. : false
Z is a good door. : false
W is a good door. : false
```

Figure 13: The result of Smullyan's problem in Doors of Enlightenment example

Both of these two sub-problems are solved in a single class. For more detailed information about code, please refer to the source file `pl.examples.DoorEnlightKB.java`.

References

[1] Russell, S. J., Norvig, P., Canny, J. F., Malik, J. M., & Edwards, D. D. (2003). *Artificial intelligence: a modern approach* (Vol. 2, No. 9). Upper Saddle River: Prentice hall.