CS 1571: Homework 3

Instructed by *Prof. Diane Litman*

Grader: *Ahmed Magooda*

Due on Nov. 13, 2017

Zac Yu (LDAP: zhy46@)

November 13, 2017

1 Report for FOL Inference: Forward Chaining

1.1 Abstract

We explored using the forward chaining algorithm to solve a simplified first-order logic (FOL) system. Specifically, given a number of rules (consist of the premises, a number of atoms, and the conclusion, an atom) followed by facts (atoms), we maintain a knowledge base for the system that contains all possible inferences at any given time. We showed that such system can effectively answer given query to prove a fact. In addition, we backtracking, we can produce a complete proof, if exists, consists of relevant facts and rules, in order, that produce the query using the generalized *modus ponendo ponens* rule. In addition, we explored the possibility to improve the efficiency of the system leveraging the incremental forward chaining algorithm. We analyzed the performance difference between the two implementations.

1.2 Implementation Decisions

1.2.1 Input and Data Structures

We first observed that the naming of predicates, constants, and variables are irrelevant to the FOL system itself. Therefore, we used two global symbol tables, for predicates and constants, to register the mappings from IDs (increasing non-negative number) to the symbols (strings, respectively the predicate names and the constant names) upon the first encounter. For variables, we noticed that it is not necessarily to preserve the original mapping. Instead, we can normalize the variable names within each rule (see details in the next paragraph).

Each atom can be represented by a tuple of a predicate ID and a list of arguments. Each argument can be either a constant represented by its global ID or a variable. Each rules is a tuple of left-hand side (LHS), the premises, and the right-hand side (RHS), the conclusion. Since the ordering of the atoms that make up the LHS is not important, to normalize the rules, we can sort them by their predicate IDs; after that, we can further normalize the variable names using increasing non-negative numbers as well. Lastly, each variable binding is a mapping (in a dictionary) from a variable to a constant ID.

For example, after processing input line of rule `Owns(Nono,x) ^ Missile(x) -> Sells(West,Nono,x)`, we will register predicate symbols `Owns = 0`, `Missile = 1`, and `Sells = 2`; constants `Nono = 0` and `West = 1`; and rule {`LHS=[(predicate=0, arguments=[(value=0, type=CONSTANT), (value=0, type=VARIABLE)]), (predicate=1, arguments=[(value=0, type=VARIABLE)])], RHS=(predicate=2, arguments=[(value=1, type=CONSTANT), (value=0, type=CONSTANT), (value=0, type=VARIABLE)])`}.

1.2.2 Forward Chaining

We invoke the forward chaining algorithm whenever a new fact is added to the knowledge base. To do so, we implemented a recursive procedure called *find and infer*, as illustrated below.

```
Procedure FIND-AND-INFER
    Input: LHS $P = \{p_1, p_2, \ldots, p_n\}$, RHS $q$, bindings $\theta$
    if $P$ is empty then
        ADD-FACT(SUBST($\theta, q$)) return
    end
    for $p'$ in $KB.facts$ do
        $\theta'$ = UNIFY(SUBST($\theta, p_1$), $p'$) if $\theta'$ is not fail then
            FIND-AND-INFER($P - p_1, q, \theta + \theta'$)
        end
    end
```

Algorithm 1: Find and Infer

Whenever a new fact is added, the *find and infer* procedure runs for every rule in the knowledge base with an initial empty dictionary of bindings. Note that the *add fact* procedure (given below) checks if a fact is already in the knowledge base before adding it to avoid loops and redundant executions. Note that the *substitution (subst)* procedure simply substitutes the bindings into an atom while the *unification (unify)* algorithm is the one given in the homework description[1].

```
Procedure ADD-FACT
    Input: fact $p'$
    Output: a boolean value indicating if the operation is successful
    if $p'$ is not a fact then
        return False
    end
    if $p'$ in $KB.facts$ then
        return True
    end
    Add $p'$ to $KB.facts$
    for rule in KB.rules do
        FIND-AND-INFER($rule.LHS, rule.RHS, \{\}$)
    end
```

Algorithm 2: Add Fact

1.2.3 Query Answering and Output

After each forward chaining procedure, the knowledge base of our system will contain the complete set of facts based on all possible usages of existing facts and rules. Therefore, to answer to some query, we can check if the query itself is contained in the facts.

In order to provide more descriptive proofs, we used a separate dictionary to keep track of the source of each fact, either given as part of the input or produced by application of some *modus ponendo ponens* rule. For the latter case, we can store the rule and bindings used to produce such result. Therefore, when answering a query, we can perform backtracking to provide the complete steps to reach the query, if the query exists in the knowledge base.

1.2.4 Incremental Forward Chaining

Since every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration $t-1$, and since we are performing forward chaining on every new incoming fact, instead of iterating over all rules, we can iterate over only rules that has some premise unifiable with the new fact. To further expedite this process and avoid infeasible unifications, we can maintain a set of relevant rules for each predicate ID when registering the rules. The incremental froward chaining version of the *add fact* procedure becomes the following.

[1]Available at `https://people.cs.pitt.edu/~litman/courses/cs1571/1571f17_HW3_program.html`

```
Procedure Add-Fact
    Input: fact $p'$
    Output: a boolean value indicating if the operation is successful
    if $p'$ is not a fact then
        return False
    end
    if $p'$ in $KB.facts$ then
        return True
    end
    Add $p'$ to $KB.facts$
    for rule in $KB.relavent_rules[p'.predicate]$ do
        for p in rule.LHS do
            $\theta$ = Unify$(p, p')$
            if $\theta$ is not fail then
                Find-And-Infer$(rule.LHS - p, rule.RHS, \theta)$
            end
        end
    end
```

Algorithm 3: Add Fact (Incremental Forward Chaining Version)

1.3 Efficiency Analysis

1.3.1 Measuring Performance

We proposed using the following metrics to measure the performance of the algorithm,

1. The number of forward-chain activations
   The homework requirement assumes one activation of the forward chaining algorithm upon each new fact being added to the knowledge base so this number should be consistent.
2. The number of attempted additions of redundant facts
   While inevitable when avoiding infinite loop, the incremental forward chaining algorithm should eliminate all other kinds of the redundancies, so not many redundant facts will be produced (and consequently attempted to be added to the knowledge base).
3. The number of performed unifications
   We expect the incremental forward chaining algorithm to attempt fewer unifications than the non-incremental version.

All the metrics above can be implemented as global counters in the system that increment when the corresponding actions take place.

1.3.2 Result

The counter values, for each test case (`ti` corresponds to `Testcase_i` and `ei` corresponds to `Example_i`), are shown in the table below.

Metrics \ Testcases	`e1`	`e2`	`t1` and `t2`	`t3` to `t7`	`t8` and `t9`
FC forward-chain activations	10	4	7	17	10
IFC forward-chain activations	10	4	7	17	10
FC redundant facts	17	4	12	96	24
IFC redundant facts	0	1	0	2	1
FC unifications	48	12	27	2155	61
IFC unifications	47	11	12	193	32

1.3.3 Analysis

We observe that the forward-chain is activated for the same number of times for both of the nonincremental and incremental versions across all test cases. This is expected because both versions of the algorithm is complete and we expect the same number of unique facts to be added to the knowledge base, each triggering one activation.
In comparison, the incremental forward-chaining algorithm performs fewer unifications and produced much fewer redundant facts. This is because only rules that are relevant to the newly introduced facts are being considered for *find and infer* in the IFC implementation. In this way, no inference rule will be used twice with the same binding, leading to the exact same conclusion. However, we also note that the redundant facts are not eliminated with IFC. This is because when there is a potential cycle, the redundancy check also serves the purpose of preventing adding the same fact in an infinite loop. For example, when we have inference rule $Same(x, y) \rightarrow Same(y, x)$ and fact $Same(A, B)$, then fact $Same(B, A)$ will be added, followed by $Same(A, B)$, which is redundant.
Finally, we observe that in test cases 3 to 7, the number of unifications are significantly larger for the nonincremental version. This is because of the large branching factor for predicates `Parent(x, y)` (unifiable with five facts), `Sibling(x, y)` and `Cousin(x, y)` (unifiable with five facts). As more inferred facts are introduced, the branches of the execution tree of *find and infer* grows exponentially. The incremental forward-chaining algorithm solves this problems by effectively pruning the branches that has been considered before.
In conclusion, the incremental forward-chaining implementation is more efficient.