James Hahn
CS1571
M/W 11am-12:15pm
Diane Litman

Implementation

The implementation for this project, although straightforward, was tricky. I began with regular forward-chaining and decided to slightly deviate from the book's code and conjure up my own solution.

Basically, all the input is read and parsed into Predicate() and Clause() objects. I decided to make this program object-oriented because there is a lot of repetition of similar actions, such as unifications, printing of predicates, creating new permutations of predicates and rules, and more.

After everything is parsed, the program loops through all of the current rules and recursively generates all combinations of unifications between the each rule and all of the facts available to us at the current iteration. If there are three rules and five facts in the first iteration of the proof, there are potentially 3*5! new rules and facts that can be introduced in the following iteration. A simple fix to this will be discussed in the *Effiency of Incremental Forward-Chaining* section below.

Next, each new rule/fact that is unique (has not been seen before) is added to a corresponding list of new facts or list of new rules. Both of these new lists will be added to our overarching lists of all facts and all rules after we have visited every rule in this iteration. This ensures no rule (in an earlier iteration) will miss out on a fact that is introduced in a newer iteration, potentially creating an additional new rule or fact. For every iteration of the program, we make sure to keep track of all rules and all facts we have seen so far.

As this entire process is occurring, we keep track of the lengths of the rules and facts lists from the previous iteration and compare it to the lengths of the rules and facts lists at the end of the current iteration. If both lengths are the same, this means we have neither created a new fact nor created a new rule, thus there is nothing new to analyze. If this is the case, we must terminate the program. Otherwise, if we create a new fact that matches our goal (also a fact), we have successfully completed the proof and can print a positive statement to the user.

In general, the program created for this project is an object-oriented brute-force traversal of all rule-fact combinations. At every iteration, we iterate over all rules generated so far and try to generate all rule-fact combinations. Once we have iterated over the facts for that current iteration, we replace the old facts list with the new facts list and repeat the above process. If it reaches the point where no new rules or facts are added, the program prints out that the proof is unsuccessful to the user. Otherwise, this means the program has found the goal and it prints out that the proof is successful.

Efficiency of Incremental Forward-Chaining

Normal forward-chaining iterates over all rules in existence and each rule iterates over all rules in existence to generate all the rule-fact combinations in hopes of creating new rules or facts. Meanwhile, incremental forward-chaining is slightly different because it does not iterate over all facts in existence, but rather the facts generated in the previous iteration of the program.

Obviously, this leads to a significant boost in efficiency because every rule looks at a fewer number of facts for every iteration compared to normal forward-chaining. If the proof has a few number of rules and facts, this may not be completely obvious, but *Testcase_5.txt* is a perfect example. If you look at the below table, incremental forward-chaining attempts 3.87% of the number of unifications as normal forward-chaining for that file. This is a significant improvement for proofs that require exhausting all combinations, such as test cases five, six, and seven. Further, in proofs that do not require exhausting all combinations for both algorithms, incremental forward-chaining still performs either equivalent or fewer comparisons. Without a doubt, incremental forward-chaining is significantly more efficient than normal forward-chaining.

Test filename	**# of normal forward-chaining unification attempts**	**# of incremental forward-chaining unification attempts**
Testcase_1.txt	16	10
Testcase_2.txt	55	13
Testcase_3.txt	97	97
Testcase_4.txt	43	43
Testcase_5.txt	7031	272
Testcase_6.txt	7031	272
Testcase_7.txt	7031	272
Testcase_8.txt	7	7
Testcase_9.txt	108	27
Example1_txt	41	15
Example2_txt	14	6

To implement incremental forward-chaining in my code, it was a matter of swapping out one variable. Instead of generating all rule-fact combinations with the list of all facts to ever exist, I generated all the combinations with the list of new facts from the previous iteration of the program (in my case, the first while loop). Specifically, I swapped the variable *all_facts* (every fact generated so far) for *facts* (facts generated from previous iteration).