CIS 671: Programming Assignment No.3

Sameer Shende
Matthew Giger
Junying Wang
Saikiran Sheth

November 8, 1995

1. Discussion

In this programming assignment, we are creating an "agent" that will store a knowlege base and retrieve information from that knowlege base. This knowlege base (or KB) consists of a list of facts, some of which were directly input into the agent and some that were inferred by the rules that the agent has. This list of rules is the second piece of information that the agent keeps track of and defines exactly what the knowlege base will contain other than the facts directly input into the KB.

Each element of the list of rules is a statement of the form:

```
(<rule-name> (if (<cond1> ... <condn>) then <conclusion>))
```

The effect that we want to produce with this is that when the new rules that are input into the function are run past all of these rules and satisfy the $< cond1 >$ to $< condn >$, a new fact will be generated, and that fact will be $< conclusion >$. The result of this will be the transitive closure of all facts that are implied or given as rules. This means that the knowlege base is complete in the sense of the rules given, in other words, all inferences that can be made by a particular fact will have been made and a reintroduction of that same fact will result in no net addition to the knowlege base.

We also want to be able to ask questions that our agent can answer about the knowlege base. Questions that pertain to the types of facts that are generated by the rule list and input facts. An example would be to ask the agent who is older than a certain person. We would input: (query (older ?x Bob)) and the agent would return a list of all of the facts in the database that fit the form (older * Bob) where the * can be anything. You can also ask it more general questions like: (query (older ?x ?y)) and it will give you all of the relations between two objects that are older.

2. Implementation

(a) Rule routines.
The rule-name, rule-conditions and rule-conclusion routines extract the rule name, conditions and conclusion from a list of rules.

(b) Satisfy.
The function satisfy takes a list of propositions and facts and returns all uniform substitutions for variables in the list of propositions (pattern list) that satisfy all of them. This function was tested and the script file is enclosed for the cases. This was tested separately for the two cases when the fact list is stored linearly and nonlinearly. This was later extended to include the new facts parameter.

(c) Monitor.
The env-subst routine takes a proposition and an association list (or an environment) and binds the variables in the proposition to their respective values. This generates new facts after the matching process.

(d) Update.
The update and satisfy functions were modified to take another argument corresponding to the new facts. This was done efficiently by generating new candidate facts that involved application of a rule dependent on at least one element of the new facts or a newly implied fact.

(e) Monitor.
The monitor function was implemented and it took facts or queries from the user and performed an update on the knowledge base or responded to the query input.

(f) Kinship relationships.
The kinship relationships for parent, child, grandchild, spouse, great-grandparent, brother, sister, daughter ,son, sibling, cousin, wife, husband, mother, father, aunt, uncle, brother-in-law, sister-in-law, first-cousin and older were implemented and tested with the royal family tree as given in problem 7.6. The test cases for both the linear and non-linear fact storage were stored in the script files. It was tested by querying relationships implied by the facts and a large number of queries demonstrate this. Addition of a new fact that was not clear from the initial input created new facts and this is shown in the script files too.

(g) Optional Part 1.
A nonlinear scheme for storage and efficient retrieval (by a predicate) of a fact in the list was implemented. This is a list of facts grouped together in the form of lists with identical predicates. Instead of modifying update, satisfy and monitor programs to run with linear and non-linear storage, we created a layer to access this list and this was used in the implementation of the monitor, update and satisfy routines. These access routines (add-fact,

get-fact, empty?, remaining, etc.) were then implemented separately as linear storage and the nonlinear storage in separate files which could be loaded to get the desired storage without changing the rest of the system. By this modular approach we found that testing of both the cases was greatly simplified. The optimization applied in this case was to ignore all the unmatched predicates (and their facts) and return a greatly reduced list each time the remaining list was requested for a predicate, and also while fetching a fact using get-fact, instead of returning just the next available fact in the non-linear storage hierarchy, the fact list was searched on the predicate and the proposition with a matching predicate was returned. This reduces the number of unsuccessful matches.

(h) Optional Part 2.
The *verbose* flag was implemented as a global variable and it controls the display of rules and facts once a rule adds a new fact to the knowledge base. This feature is also demonstrated in the script files enclosed. Due to the large amount of data displayed it was not feasible to include all the script files in the report. The location of these scripts is given below and may be examined for a complete view. Some of the script files have been pruned to enclose in the report for the sake of brevity.

3. Algorithms

An optimization made to the udpate and satisfy procedures is to separate the new facts input from the facts already in the knowlege base. This is done since we already have the transitive closure of all of the facts contained in the original knowlege base, therefore the only new facts that will be introduced into the KB will come from the new facts. The optimization lies in the checking of the list of satisfying propositions for each rule against the new input facts to see if there are any mention of them. If not, the entire rule can be skipped, if so we must process like normal.

Another optimization that we made was to sort the knowlege base into a kind of a hash table. We grouped facts with the same predicate into lists and then grouped those lists to make up the knowlege base. This is done to be able to quickly skip over a large number of facts when searching for ones to satisfy the preconditions for a rule. There is a penalty when inserting a new fact into the list since we have to find the correct place to put it but that operation is not done as often as searching for rule condition satisfaction.

4. Software

The software for the project is located in /home/grads/sameer/cis671/p3 directory. The files are as follows :

- monitor.sch - the monitor routine which add facts or process queries.
- satisfy_test.sch - the satisfy routine which takes propositions and facts.
- match.sch - the match routine which matches a pattern with an expression.

- rule.sch - the rule routines for part I to extract the name, conditions and conclusions from a rule list.
- update.sch - the update routine that determines the complete set of facts given new facts.
- env.sch - the env-subst routine that creates facts given the bindings and the environment.
- kin.sch - the kinship rules that determine the relationships.
- linear.sch - the fact list layer for linear storage of facts.
- nonlinear.sch - the fact list layer for nonlinear storage of facts.
- load.sch - the script that loads the files for the monitor routine.
- fam_data.sch - the fact list for giving to the monitor routine for the kinship relationships of problem 7.6.
- rl.sch - the test file that contains test cases for testing thesatisfy and update routines.

  The script files for the monitor program on kinship relationships and other test cases are stored as follows :
- linear.script - the monitor run with the entire family tree using linear storage of facts. Shows queries and addition of new facts. Extensively tests the relationships.
- nonlinear.script - the monitor run with the kinship relationships using non-linear storage of facts as described above. It shows the new facts being fired by the rules (optional part I and II).
- final.script - the monitor run with non-linear storage and different queries and facts.
- sat.script - the satisfy and update testing scripts used for part II and IV.
- sameer.script, sam.script jwang.test - other script files showing the different phases of development and testing.

5. Conclusions

The unavailability of disjunctions in the rule condiditons makes the list list of rules much larger than it could be. For example, we must generate several different rules to say that a person is a male like if he is an uncle or if he is a father or a husband or a grandfather and so on. If we could add all of these into one rule, the processing time would have been sped up greatly due to the smaller number of rules to traverse for each new fact.

The knowledge server was designed and implemented using the forward chaining (or data-driven) algorithm.

6. Appendix

The appendix contains the source codes and the script files for the assignment.