Massachusetts Institute of Technology

6.034 Artificial Intelligence

Solutions # **2** 6034 Item # **6**

Problem 1

Part A

The rule interpreter matches all rules against all assertions in the database. Since the rule antecedents contain patterns, in general there are (often) multiple (sets of) assertions which can match the antecedents of a rule. When a rule name is listed multiple times, it means there are many of these instantiations, each of which binds the rules variables to different values.

The rule instance fired will be the one constructed from the first triggered rule in the list of rules together as instantiated with the variable bindings from the assertion earliest in the list of assertions. The input assertions are added to the assertions list by `cons` and so, for example, the last assertion in the input list occurs earlier in the list of assertions. More recently derived assertions are also earlier in the list.

Part B

There are many ways in which the current implementation is wasteful. Here's a couple:

- There is no memory from cycle to cycle. Rules and assertion that failed to match last time will continue to fail, yet they are tried again and again.
- The code computes all the bindings and instances for a rule and then throws out all but the first.

Fixing these problems is relatively straightforward but would complicate the code, which is why we haven't done it.

Part C

The growth of the worst-case time to find a rule to fire is on the order of RS^k. This follows because a rule has to consider every assertion as a possible match for EACH of the IF expressions. For each match for the first assertion, it has to scan all the other assertions to get consistent matches, and so on. This defines a tree of depth k and branching factor S. This is precisely what `apply-filters` does. That's where all the time in running the interpreter goes.

The function `match-pattern-to-assertions` matches one pattern (an antecedent from a rule) against (in the worst case) all the assertions. It takes a bindings-list as an argument, so it checks that the any output binding is consistent with that of other antecedents of the rule. So, treating the cost of `match` as a constant, we can say the one call to `match-pattern-to-assertions` does order S work. However, for a rule with k antecedents, we have to take each of the variable bindings produced by previous calls to `match-pattern-to-assertions` and recursively call `match-pattern-to-assertions` for each of these accumulated bindings. That is what `apply-filters` and `filter-bindings-list` does. So, a call to `apply-filters` generates on the order of S^k work. Then `try-rule` is called, in the worst case, one for each rule – giving us the final (horrifying) worst-case cost.

Note that the worst case happens when most of the patterns in the rules match assertions in the database since each match creates a new branch in the matching tree mentioned above and each failure terminates a branch. Because of this, one wants to order the patterns in the IF part of the rule so that the ones that are least likely to be true or that match the fewest assertions are at the beginning of the list. Otherwise, if you have an assertion at the beginning of the list that will match hundreds of facts then each match will have to be checked against subsequent patterns.

This is why the RETE algorithm is so important in practice.

Problem 2

Here's a very simple set of rules for the auto diagnosis problem. These basically identify the conditions under which each repair is justified.

```
(remember-rules
 '(R1 IF (engine-starts yes)
      THEN (repair is nothing))
 '(R2 IF (engine-starts no)
 (headlights-work no)
      THEN (repair is charge-battery))
 '(R3 IF (engine-starts no)
 (engine-rotates no)
         (headlights-work yes)
      THEN (repair is fix-starter-motor))
 '(R4 IF (engine-starts no)
 (engine-rotates yes)
         (headlights-work yes)
 (fuel yes)
      THEN (repair is fix-ignition))
 '(R5 IF (engine-starts no)
 (engine-rotates yes)
         (headlights-work yes)
 (fuel no)
      THEN (repair is add-fuel)))
```

Part A

Rules like the following:

```
(R1 IF (headlights-work yes)
```

```
        THEN (battery-charged yes))
    (R2 IF (engine-rotates yes)
        THEN (battery-charged yes)
             (starter-motor-works yes))
    (R3 IF (engine-starts yes)
        THEN (battery-charged yes)
             (starter-motor-works yes)
     (ignition-works yes)
     (fuel yes))
```

capture what is known to work on the basis of the positive evidence, what is known to work. We would still need the rules that work from the negative evidence, that is, what is known not to work.

Problem 3

Part A

If you attempt to run the logic rules using the "first rule-instance" conflict-resolution strategy, you get an infinite loop. The "explosive" rules simply add irrelevant assertions until they fill up memory (which happens very quickly).

Part B

The rules A1 (And introductions), A3 (Double Negation), A13 (Implication Introduction) and A14(Implication Distribution) are all explosive, in the sense that the form of the new assertion added (together with previous assertions) guarantee that the rule will trigger again on the new assertion.

Part C

In an interpreter with rule-ordering as the only conflict-resolution strategy, the presence of any one of these rules guarantees an infinite loop (except for simple problems) that can be solved without needing any of the rules. Note that any rules placed after the first of these "explosive" rules will never be executed.

Problem 4

Here we see an implementation of recency-based conflict resolution.

Part A

```
(define *rule-recency* (make-eq-hash-table))
(define (chain)
  ;; Initialize recency for all rules.
  (init-recency)
```

```
  (chain-internal try-first-rule 1))
(define (init-recency)
  (hash-table/clear! *rule-recency*)
  (for-each
   (lambda (rule) (hash-table/put! *rule-recency* (rule-name rule) 0))
   *rules*))
(define (chain-internal rule-instance-select-fn step)
   ...
          ;; update the recency for fired rule.
          (hash-table/put! *rule-recency* name step)
          ;; Keep chaining
          (if (not (member '(STOP) *assertions*))
              (chain-internal rule-instance-select-fn (+ 1 step))
              (format #t "~%STOP.")
              ))
        ;; No more rules...
        (format #t "~%No rule can assert anything new"))))
;; compare most recent firing step.
(define (least-recent-first r1 r2)
  (< (hash-table/get *rule-recency* (rule-name r1) 0)
     (hash-table/get *rule-recency* (rule-name r2) 0)))
(define (try-first-rule)
  ;; Check each rules and find the first that triggers.  This is less
  ;; general than try-all-rules, but much faster...
  ;; Sort the rules so least-recently fired rule is first.
  (or (there-exists? (sort *rules* least-recent-first)
                     (lambda (rule) (try-rule rule)))
      ;; always rerurns a list, empty in this case.
      '()))
```

Part B

Here's a run on the logic problem (edited to shorten the huge lists):

```
(logic-run)
Logic rules recorded.
Logic test recorded:
(heads => (win me))
(tails => (not (win you)))
((not heads) => tails)
((win you) => (not (win me)))
(win you)
Step 1: Triggered rules: (a11 a11 a11 a11)
Rule a11 adds ((not heads) or (win me))
Implication definition from (heads => (win me))
Step 2: Triggered rules: (a3 a3 a3 a3 a3 a3)
Rule a3 adds (not (not ((not heads) or (win me))))
Double negation from ((not heads) or (win me))
Step 3: Triggered rules: (a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1
a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1
a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1)
Rule a1 adds ((not (not ((not heads) or (win me)))) and (not (not
((not heads) or (win me)))))
And introduction from (not (not ((not heads) or (win me)))), (not (not
((not heads) or (win me))))
Step 4: Triggered rules: (a13 a13 a13 a13 a13 a13 a13 a13 a13 a13 a13
...
a13 a13)
Rule a13 adds (((not (not ((not heads) or (win me)))) and (not (not
((not heads) or (win me))))) => ((not (not ((not heads) or (win me))))
and (not (not ((not heads) or (win me))))))
Implication introduction from ((not (not ((not heads) or (win me))))
and (not (not ((not heads) or (win me))))), ((not (not ((not heads) or
(win me)))) and (not (not ((not heads) or (win me)))))
```

```
Step 5: Triggered rules: (a5)
Rule a5 adds (not (win me))
Modus Ponens from ((win you) => (not (win me))), (win you)
Step 6: Triggered rules: (a10)
Rule a10 adds (not ((not ((not heads) or (win me))) or (not ((not
heads) or (win me)))))
De Morgan's from ((not (not ((not heads) or (win me)))) and (not (not
((not heads) or (win me)))))
Step 7: Triggered rules: (a6)
Rule a6 adds (not heads)
Modus Tollens from (heads => (win me)), (not (win me))
Step 8: Triggered rules: (a11 a11 a11 a11)
Rule a11 adds ((not ((not (not ((not heads) or (win me)))) and (not
(not ((not heads) or (win me)))))) or ((not (not ((not heads) or (win
me)))) and (not (not ((not heads) or (win me))))))
Implication definition from (((not (not ((not heads) or (win me))))
and (not (not ((not heads) or (win me))))) => ((not (not ((not heads)
or (win me)))) and (not (not ((not heads) or (win me))))))
Step 9: Triggered rules: (a3 a3 a3 a3 a3 a3 a3 a3 a3 a3 a3 a3)
Rule a3 adds (not (not ((not ((not (not ((not heads) or (win me))))
and (not (not ((not heads) or (win me)))))) or ((not (not ((not heads)
or (win me)))) and (not (not ((not heads) or (win me))))))))
Double negation from ((not ((not (not ((not heads) or (win me)))) and
(not (not ((not heads) or (win me)))))) or ((not (not ((not heads) or
(win me)))) and (not (not ((not heads) or (win me))))))
Step 10: Triggered rules: (a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1
...
a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1)
Rule a1 adds ((not (not ((not ((not (not ((not heads) or (win me))))
...
(not (not ((not heads) or (win me)))))))))
And introduction from (not (not ((not ((not (not ((not heads) or (win
...
me)))) and (not (not ((not heads) or (win me))))))))
Step 11: Triggered rules: (a13 a13 a13 a13 a13 a13 a13 a13 a13 a13 a13
...
a13 a13 a13 a13 a13 a13 a13 a13)
Rule a13 adds (((not (not ((not ((not (not ((not heads) or (win me))))
...
(not ((not heads) or (win me)))) and (not (not ((not heads) or (win
me))))))))))
Implication introduction from ((not (not ((not ((not (not ((not heads)
...
heads) or (win me)))))))))
Step 12: Triggered rules: (a5)
Rule a5 adds tails
Modus Ponens from ((not heads) => tails), (not heads)
Step 13: Triggered rules: (a10)
Rule a10 adds (not ((not ((not ((not (not ((not heads) or (win me))))
...
(not (not ((not heads) or (win me)))))))))
De Morgan's from ((not (not ((not ((not (not ((not heads) or (win
...
me)))) and (not (not ((not heads) or (win me)))))))))
Step 14: Triggered rules: (a11 a11 a11 a11)
Rule a11 adds ((not ((not (not ((not ((not (not ((not heads) or (win
...
heads) or (win me))))))))))
Implication definition from (((not (not ((not ((not (not ((not heads)
...
heads) or (win me))))))))))
Step 15: Triggered rules: (a3 a3 a3 a3 a3 a3 a3 a3 a3 a3 a3 a3 a3 a3
a3 a3 a3)
Rule a3 adds (not (not ((not ((not (not ((not ((not (not ((not heads)
...
(not (not ((not heads) or (win me))))))))))
Step 16: Triggered rules: (a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1
...
```

```
a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1)
Rule a1 adds ((not (not ((not ((not (not ((not ((not (not ((not heads)
...
me)))))))))))))
And introduction from (not (not ((not ((not (not ((not ((not (not
...
me))))))))))))
Step 17: Triggered rules: (a13 a13 a13 a13 a13 a13 a13 a13 a13 a13 a13
...
a13)
Rule a13 adds (((not (not ((not ((not (not ((not ((not (not ((not
...
me))))))))))))))
Implication introduction from ((not (not ((not ((not (not ((not ((not
...
me)))))))))))))
Step 18: Triggered rules: (a5)
Rule a5 adds (not (win you))
Modus Ponens from (tails => (not (win you))), tails
Step 19: Triggered rules: (a0)
Rule a0 adds (stop)
Contradiction (win you) (not (win you))
STOP.
(stop)
(not (win you))
(((not (not ((not ((not (not ((not ((not (not ((not heads) or (win
...
(not (not ((not heads) or (win me))))))))))))))
((not (not ((not ((not (not ((not ((not (not ((not heads) or (win
...
me))))))))))
(not ((not ((not ((not (not ((not heads) or (win me)))) and (not (not
...
((not heads) or (win me)))))))))
tails
(((not (not ((not ((not (not ((not heads) or (win me)))) and (not (not
...
me))))))))))
((not (not ((not ((not (not ((not heads) or (win me)))) and (not (not
...
(not (not ((not heads) or (win me)))))))))
(not (not ((not ((not (not ((not heads) or (win me)))) and (not (not
...
me)))) and (not (not ((not heads) or (win me))))))))
((not ((not (not ((not heads) or (win me)))) and (not (not ((not
heads) or (win me)))))) or ((not (not ((not heads) or (win me)))) and
(not (not ((not heads) or (win me))))))
(not heads)
(not ((not ((not heads) or (win me))) or (not ((not heads) or (win
me)))))
(not (win me))
(((not (not ((not heads) or (win me)))) and (not (not ((not heads) or
(win me))))) => ((not (not ((not heads) or (win me)))) and (not (not
((not heads) or (win me))))))
((not (not ((not heads) or (win me)))) and (not (not ((not heads) or
(win me)))))
(not (not ((not heads) or (win me))))
((not heads) or (win me))
(heads => (win me))
(tails => (not (win you)))
((not heads) => tails)
((win you) => (not (win me)))
(win you)
;No value
```

Part C

What we have done with the recency conflict-resolution is to enable us to find a contradiction (if the rules and assertions imply one) in FINITE time. With rule-order, the system can loop INFINITELY even when a contradiction is deducible from the rules and assertions. Note, however, that the explosive rules will continue to fire under a recency conflict-resolution strategy; cluttering the data base with useless facts and providing fodder for subsequent useless firings. Thus, the number of assertions can (and will) grow very large and so the finite time might (and generally is) very large for interesting cases. Furthermore, when a contradiction cannot be found from the rules and assertions, the system will run forever.

So, in one sense, we have solved the explosiveness problem but in another we have merely postponed it. So, we may be able to solve some relatively simple problems that would not have worked using only rule-ordering but eventually we will drown in useless assertions. A backwards chaining strategy is almost always used with logic rules to alleviate (but not eliminate) this problem.