CS152 7.2 PCW

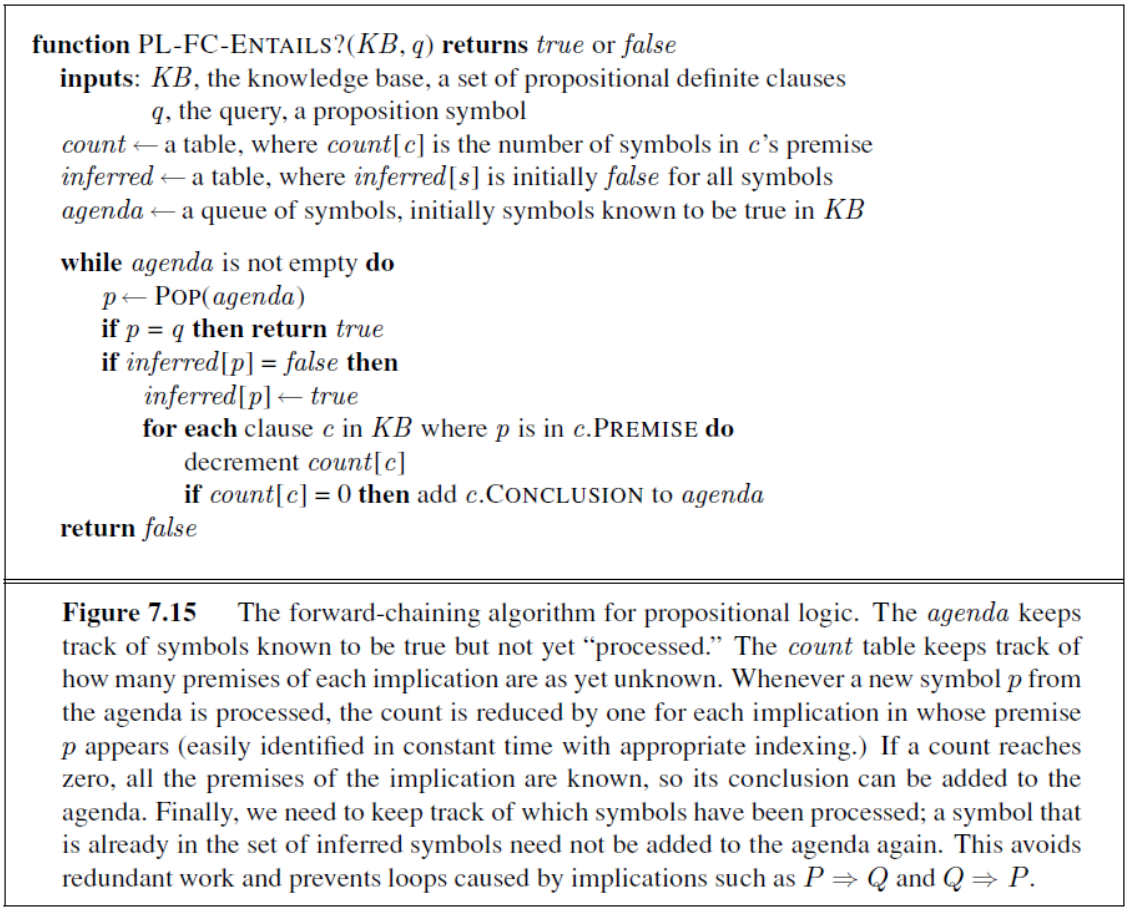
October 23, 2019

1 Forward chaining implimentation

Implement a forward chaining solver in Python for ascertaining the truth of a symbol given a knowledge base of definite clauses, following the algorithm in Fig 7.15 of Russell & Norvig. You could implement a class called Symbol to represent a logical symbol, as well as a class called DefiniteClause to represent the structure of a definite clause, with a premise set (the body of the clause) and a symbol for the conclusion (the head of the clause). A knowledge base can then be represented as a list of definite clauses.

```
In [44]: from PIL import Image
         Image.open("CS152 7.2.png")
```

Out[44]:



function PL-FC-ENTAILS?(KB, q) **returns** *true* or *false*
  inputs: KB, the knowledge base, a set of propositional definite clauses
    q, the query, a proposition symbol
  count ← a table, where *count*[c] is the number of symbols in c's premise
  inferred ← a table, where *inferred*[s] is initially *false* for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while *agenda* is not empty **do**
    p ← POP(*agenda*)
    if $p = q$ **then return** *true*
    if *inferred*[p] = *false* **then**
      inferred[p] ← *true*
      for each clause c in KB where p is in c.PREMISE **do**
        decrement *count*[c]
        if *count*[c] = 0 **then** add c.CONCLUSION to *agenda*
  return *false*

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

In [25]:
```
class DefiniteClause:
    def __init__(self, head, body=set()):
        self.body = body
        self.head = head
```

In [46]:
```
def ForwardChaining(KB, q):
    count = [len(clause.body) for clause in KB]

    inferred = {symbol: False
                for symbol in set([sym
                for clause in KB
                for sym in clause.body]
                + [clause.head for clause in KB])}

    agenda = [clause.head for clause in KB]

    while len(agenda) != 0:
        p = agenda.pop()
        if p == q: return True
        if inferred[p] == False:
            inferred[p] = True

            for i, clause in enumerate(KB):
                if p in clause.body:
                    count[i] -= 1
                    if count[i] == 0: agenda.append(clause.head)
    return False
```

In [47]:
```
Image.open("Untitled.png")
```

Out[47]:

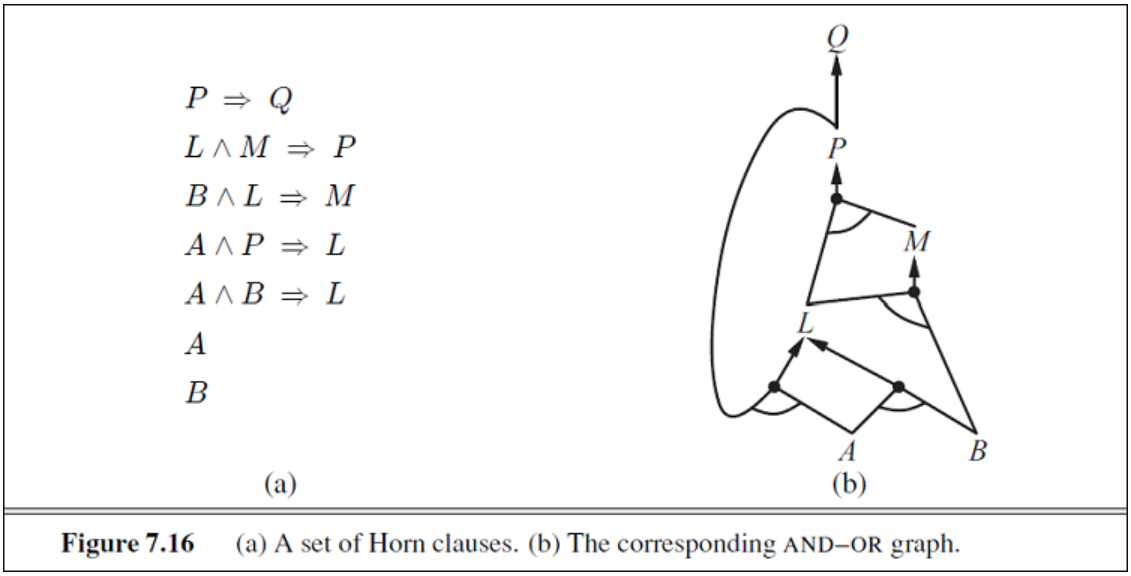


Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

```
In [48]: KB = [DefiniteClause("Q", ["P"]),
               DefiniteClause("P", ["L", "M"]),
               DefiniteClause("M", ["B", "L"]),
               DefiniteClause("L", ["A", "P"]),
               DefiniteClause("L", ["A", "B"]),
               DefiniteClause("A"),
               DefiniteClause("B")]

         print(ForwardChaining(KB, "Q"))
```

```
True
```