Ahsanullah University of Science and Technology (AUST)

Department of Computer Science and Engineering

Term Assignment-1

Course No. : CSE4108
Course Title: Artificial Intelligence Lab

Date of Submission: 03/04/2021

Submitted By: Lab Group: A1

Section: A

Name: Umme Habiba

ID: 170104004

Submitted To: Dr. S.M.A Al Mamun

Tonmoy Hossain Dihan

Sajib Kumar Saha Joy

Topic-2: Backward Chaining.

Backward Chaining: Backward chaining is the logical process of inferring unknown truths from known conclusions by moving backward from a solution to determine the initial conditions and rules. Another way Backward chaining is a kind of AND/OR search- the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the left-hand-side of the clause must be proved.

Backward chaining is used in artificial intelligence applications for logic programming, reasoning, and behavior analysis. It's part of a system that aims to teach robots how to infer and make logical conclusions.

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a depth-first search strategy for proof.

Basic steps of backward chaining:

Query→goal→conclusion→premise→subgoal→conclusion→

premise→subgoal→backtracking→……..until proved or KB exhausted

An example of backward chaining using python will go as follows:

Sample Input: A knowledge base is internally declared in the program consisting of rules and facts. Desired *goal* is taken as user input.

```
KB= [['A'],
     ['B'],
     ['C'],
     ['D'],
     ['P','Q'],
     ['C','L','P'],
     ['D','M','P'],
     ['B','L','M'],
     ['A','P','L'],
     ['A','B','L'],
     ['A','D','G'],
     ['G','B','D']]
```

Accurate Output: From KB if 'M' is entered as user input, after applying backward chaining algorithm the output should return True, which means 'M' can be proved.

Major Steps: To find goal state 'M', here we will follow FOL-BC-ASK which is a backward chaining algorithm. The FOL-BC-ASK algorithm returns multiple results. Its general form is analogous to and/or search. Or part allows proof of a goal by any KB rule and part requires all the rules to be proved. Here, FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause of the brand-new variables, and then, if the right-hand-side of the clause does indeed unify with the goal, proving every conjunction in the right-hand-side, using FOL-BC-AND. That function in turn works by proving each of the conjunctions in turn, keeping track of the accumulated substitution as we go.

Mathematical calculation will be as follows:

KB:

A
B
C
D
P=>Q
C^L=>P
D^M=>P
B^L=>M
A^P=>L
A^B=>L
A^D=>G
G^B=>D

#Query : KB|=M?

1. ? M
2. B^L=>M , found in KB
3. ?B, found in KB
4. ?B
5. B, found in KB
6. ?L
7. A^P=>L, found in KB , alternates are there
8. ?A ^P
9. ?A ?P
10. A, found in KB
11. ?P
12. C^L=> P, found in kb, alternates are there
13. ?C^L
14. ?C ?L
15. C,found in KB
16. ?L [Backtrack needed go to line 6]

The search tree for input ‘M’ is given below.

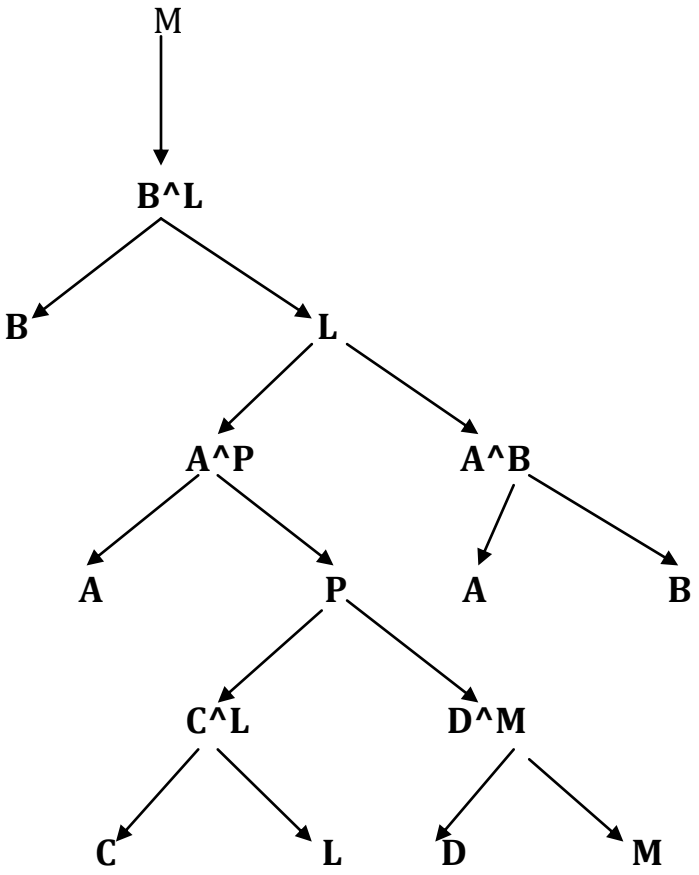


Fig: Search tree for input 'M'

Implementation in Python:

```
KB= [['A'],
     ['B'],
     ['C'],
     ['D'],
     ['P','Q'],
     ['C','L','P'],
     ['D','M','P'],
     ['B','L','M'],
     ['A','P','L'],
     ['A','B','L'],
     ['A','D','G'],
     ['G','B','D']]
```

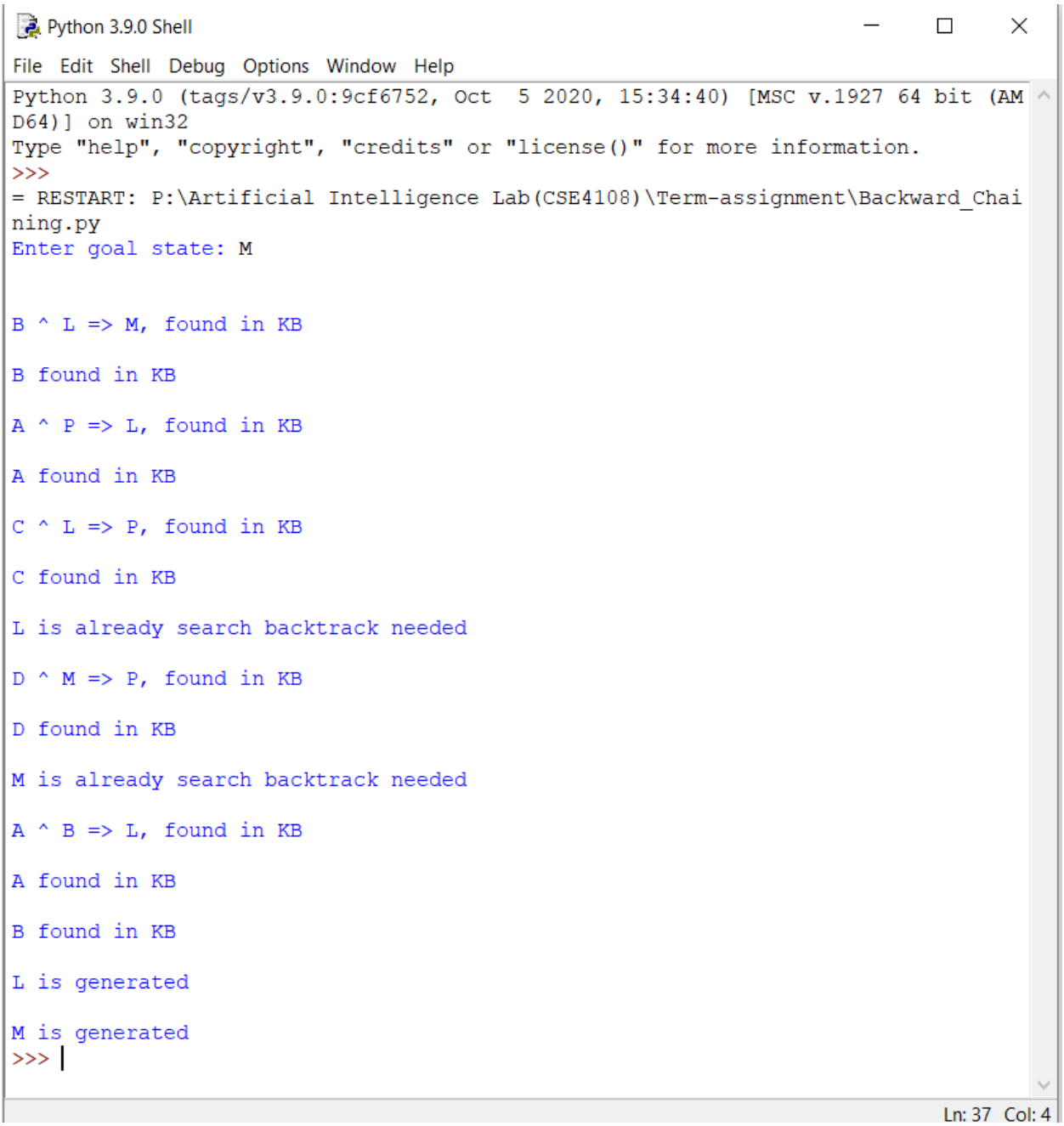
```
file=open("store.txt","a+")
running=[]
def backward_chaining(x):
   global KB,running,file
   if x in running:
      file.write("\n" +x+" is already search backtrack needed")
      return
   else:
      running.append(x)

   for i in range(len(KB)):
      if len(KB[i])== 1 and KB[i][0]== x:
         file.write("\n" +x+ " found in KB")
         running.remove(x)
         return True
      elif len(KB[i])== 2 and KB[i][1]==x:
         print(",found in KB alternate are there" + KB[i][0] + " => "+ KB[i][1])
         if backward_chaining(KB[i][0]):
            file.write("\n" +KB[i][1] + " is generated")
            return True
         else:
             file.write(KB[i][1] + " cannot be generated")
             return False
      elif len(KB[i])== 3 and KB[i][2]== x:
          file.write("\n" + KB[i][0] + " ^ "+ KB[i][1] + " => "+ KB[i][2]+", found in
KB")
          if backward_chaining(KB[i][0]) and backward_chaining(KB[i][1]):
                  file.write("\n" +KB[i][2] + " is generated")
                  return True
x=input("Enter goal state: ")
backward_chaining(x)
file.close()

a_file = open("store.txt")
lines = a_file. readlines()
for line in lines:
   print(line)
a_file. close()
```

Saved database of intermediate activities : All the steps including searching in the KB, backtracking are saved sequentially in a file named “store.txt”. Below is the sample output of the steps in the terminal when ‘M’ is the goal query.

Output:



Read from store.txt file input:

A
B
C
D
P=>Q
C^L=>P
D^M=>P
B^L=>M
A^P=>L
A^B=>L
A^D=>G
G^B=>D

Advantage of backward chaining:

- The result is already known, which makes it easy to deduce inferences.
- It's a quicker method of reasoning than forward chaining because the endpoint is available.
- In this type of chaining, correct solutions can be derived effectively if pre-determined rules are met by the inference engine.

Disadvantage of backward chaining:

- The process of reasoning can only start if the endpoint is known.
- It doesn't deduce multiple solutions or answers.
- It only derives data that is needed, which makes it less flexible than forward chaining.