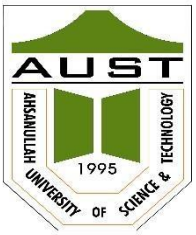
AHSANULLAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

DHAKA-1208, BANGLADESH.



Department of Computer Science and Engineering

Spring 2019

Program: Bachelor of Science in Computer Science and Engineering

Course No: CSE 4108

Course Title: Artificial Intelligence Lab

Term Project No: 01

Term Project Name: Backward Chaining

Date of Submission: 22/07/2019

Submitted to

Dr. S.M. Abdullah Al-Mamun

Professor, Department of CSE, AUST.

Md. Siam Ansary

Adjunct Faculty, Department of CSE, AUST.

Submitted By

Name : Robiul Hasan Nowshad

Student ID : 16.01.04.061

Lab Group : B1

BACKWARD CHAINING

Backward Chaining

Backward chaining (or backward reasoning) is an inference method described colloquially as working backward from the goal. It is used in automated theorem prove, inference engines, proof assistants, and other artificial intelligence applications.

How it works

Backward chaining starts with a list of goals (or a hypothesis) and works backwards from the consequent to the antecedent to see if any data supports any of these consequent. An inference engine using backward chaining would search the inference rules until it finds one with a consequent that matches a desired goal. If the antecedent of that rule is known to be true, then it is added to the list of goals (for one's goal to be confirmed one must also provide data that confirms this new rule).
The FOL-BC-Ask is a backward chaining algorithm given bellow. It is called with a list of goals containing an element, the original query, and returns the set of all substitutions satisfying the query.

Algorithm

Function FOL-BC-Ask(KB,goals,@) **returns** a set of substitutions
inputs: KB, a knowledge base
goals, a list of conjuncts forming a query (@ already applied)
@, the current substitution, initially the empty subs. { }
local variables: answers, a set of substitutions, initially empty { }

if goals is empty **then return** {@}
Q<-- Subst(@,First(goals))
for each sentence r **in** KB where STANDARDIZE-APART(r)=(p1 ^…^pn -->q) and @<-- Unify (q,q') succeeds
new_goals<--[p1,..., pn | Rest(goals)]
answers<-- FOL-BC-Ask(KB, new_goals, Compose(@',@)) U answers
return answers

Implementation

Input instruction:
Program will be given an input file. The first line of the input will be the number of queries (n). Following n lines will be the queries, one per line. For each of them, you have to determine whether it can be proved form the knowledge base or not.
Next line of the input will contain the number of clauses in the knowledge base (m).
Following, there will be m lines each containing a statement in the knowledge base. Each clause is in one of these two formats: 1- p1 ∧ p2 ∧ ... ∧ pn => q
2- facts: which are atomic sentences. Such as p or ~p All the p s and also q are either a literal such as HasPermission(Google,Contacts) or negative of a literal such as ~HasPermission(Google,Contacts).

- Queries will not contain any variables.
- Variables are all single lowercase letters
- All predicates (such as HasPermission) and constants (such as Google) are case-sensitive alphabetical strings that begin with uppercase letters.
- Each predicate has at least one argument. (There is no upper bound for the number of arguments). Same predicate will not appear with different number of arguments.
- See the sample inputs for spacing patterns.
- All of the arguments of the facts are constants. i.e. you can assume that there will be no fact such as HasPermission(x,Contacts) (which says that everyone has permission to see the contacts!) in the knowledge base.
- You can assume that the input format is exactly as it is described. There are no errors in the given input.

Sample Input:
input.txt
1
B(John,Bob)
13
A(x,y) => B(x,y)
G(x,y) => B(x,y)
C(c,d) => A(c,d)
B(x,y) => G(x,y)
F(y) => G(x,y)
G(x,y) ^ F(y) => H(x,y)
I(z) => F(z)
F(z) => I(z)
C(Jie,Joe)
C(Melissa,Mary)
G(John,Bob)
I(Amy)
F(Bob)

Source Code:

```
#Robiul Hasan Nowshad(Aust37)

from copy import deepcopy

kb = {}
list_of_predicates = []
list_of_explored_rules = []

def fetch_rules(goal):
    global kb
    global list_of_predicates

    print("fetch_rules for goal:- ", goal)
    list_of_rules = []
    predicate = goal.partition('(')[0]
    print("\t", predicate, kb[predicate]['conc'])
    list_of_rules = list_of_rules + kb[predicate]['conc']
    return list_of_rules


def subst(theta, first):
    print("\tsubst: ", theta, first)
    predicate = first.partition('(')[0]
    list = (first.partition('(')[-1].rpartition(')')[0]).split(',')
    print("\t", list)
    for i in range(len(list)):
        if variable(list[i]):
            if list[i] in theta:
                list[i] = theta[list[i]]
    print("\t", predicate + '(' + ','.join(list) + ')')
    return predicate + '(' + ','.join(list) + ')'


def variable(x):
    if not isinstance(x, str):
        return False
    else:
        if x.islower():
            return True
        else:
            return False


def compound(x):
    if not isinstance(x, str):
        return False
    else:
        if '(' in x and ')' in x:
            return True
        else:
            return False


def list(x):
    if not isinstance(x, str):
```

```
        return True
    else:
        return False


def unify_var(var, x, theta):
    print("IN unify_var", var, x, theta)
    if var in theta:
        print("var in theta", var, theta)
        return unify(theta[var], x, theta)
    elif x in theta:
        print("x in theta", x, theta)
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        print("not in theta", theta[var])
        return theta


def check_theta(theta):
    for entry in theta:
        if variable(theta[entry]):
            if theta[entry] in theta:
                print("in check_theta. theta changed")
                theta[entry] = theta[theta[entry]]
    return theta


def unify(x, y, theta):
    print("\tunify", x, y, theta)
    if theta == None:
        print("\tin theta is None")
        return None
    elif x == y:
        print("\tin x=y")
        return check_theta(theta)
    elif variable(x) is True:
        print("\tin variable(x)")
        return unify_var(x, y, theta)
    elif variable(y) is True:
        print("\tin variable(y)")
        return unify_var(y, x, theta)
    elif compound(x) and compound(y):
        print("\tin compound")
              x_args = []
              temp = x.partition('(')[-1].rpartition(')')[0]
              for item in temp.split(','):
                  x_args.append(item)
              y_args = []
              temp = y.partition('(')[-1].rpartition(')')[0]
              for item in temp.split(','):
                  y_args.append(item)
              x_op = x.partition('(')[0]
              y_op = y.partition('(')[0]
              return unify(x_args, y_args, unify(x_op, y_op, theta))
          elif list(x) and list(y):
              print("\tin list")
              return unify(x[1:], y[1:], unify(x[0], y[0], theta))
          else:
```

```
            print("\tin else")
            return None


    def fol_bc_ask(query, theta):
        global kb
        global list_of_predicates
        global list_of_explored_rules

        print("Backward Chaining")
        list_of_rules = fetch_rules(query)
        for rule in list_of_rules:
            print("taken RULE", rule)
            list_of_explored_rules = []
            list_of_explored_rules.append(query)
            print("\t",query, "added to list_of_explored_rules")
            lhs = rule.partition('=>')[0]
            rhs = rule.partition('=>')[2]
            print("lhs: ", lhs, " rhs: ", rhs)
            print("theta in rule", theta)
            theta1 = unify(rhs, query, theta)
            if theta1 != None:
                list_of_premises = lhs.split('^')
                print("list_of_premises: ", list_of_premises)
                theta2 = fol_bc_and(theta1, list_of_premises)
                if theta2 != None:
                    return theta2

        print("None of the rules worked out", query)
        return None

    def fol_bc_and(theta, list_of_premises):
        global kb
        global list_of_predicates

        print("\tand: ", list_of_premises)
        print("\ttheta: ", theta)
        if theta == None:
            return None
        else:
            if list_of_premises != []:
                temp_list = []
                for each_premise in list_of_premises:
                    temp = subst(theta, each_premise)
                    temp_list.append(temp)
                list_of_premises = temp_list
                first_premise = list_of_premises[0]
                rest_premise = list_of_premises[1:]
                subs = list_of_premises[0]
                if subs != '()':
                    if subs in list_of_explored_rules:
                        print(subs, "already in list_of_explored_rules"

                        return None
                    else:
                        print(subs, " added to list_of_explored_rules")

                        list_of_explored_rules.append(subs)
                    theta = fol_bc_or_sub(subs, {}, rest_premise)
```

```
                else:
                    return theta
            return theta


    def fol_bc_or_sub(query, theta, rest):
        global kb
        global list_of_predicates

        print("\tOR sub")
        list_of_rules = fetch_rules(query)
        print("\tLIST_OF_RULES", list_of_rules)
        for rule in list_of_rules:
            print("\tRULE", rule)
            lhs = rule.partition('=>')[0]
            rhs = rule.partition('=>')[2]
            print("\n\tlhs: ", lhs, " rhs: ", rhs)
            print("\ntheta in rule", theta)
            theta1 = unify(rhs, query, deepcopy(theta))
            if theta1 != None:
                list_of_premises = lhs.split('^')
                print("\tlist_of_premises: ", list_of_premises)
                theta2 = fol_bc_and(theta1, list_of_premises)
                theta3 = fol_bc_and(theta2, rest)
                if theta3 != None:
                    return theta3

        print("\tNone of the rules worked out", query)
        return None


    def add_to_kb(knowledge_base):
        global kb
        global list_of_predicates

        for sentence in knowledge_base:
            if '=>' not in sentence:
                predicate = sentence.partition('(')[0]
                if predicate not in list_of_predicates:
                    conc = []
                    prem = []
                    conc.append("=>" + sentence)
                    kb[predicate] = {'conc': conc, 'prem': prem}
                    list_of_predicates.append(predicate)
                else:
                    conc = kb[predicate]['conc']
                    prem = kb[predicate]['prem']
                    conc.append("=>" + sentence)
                    kb[predicate] = {'conc': conc, 'prem': prem}
            else:
                clauses = sentence.partition('=>')
                list_of_premises = clauses[0].split('^')
                conclusion = clauses[2]

                # for conclusion
                predicate = conclusion.partition('(')[0]
                if predicate not in list_of_predicates:
                    conc = []
                    prem = []
```

```
                        conc.append(sentence)
                        kb[predicate] = {'conc': conc, 'prem': prem}
                        list_of_predicates.append(predicate)
                    else:
                        conc = kb[predicate]['conc']
                        prem = kb[predicate]['prem']
                        conc.append(sentence)
                        kb[predicate] = {'conc': conc, 'prem': prem}

                    # for list_of_premises
                    for premise in list_of_premises:
                        predicate = premise.partition('(')[0]
                        if predicate not in list_of_predicates:
                            conc = []
                            prem = []
                            prem.append(sentence)
                            kb[predicate] = {'conc': conc, 'prem': prem}
                            list_of_predicates.append(predicate)
                        else:
                            conc = kb[predicate]['conc']
                            prem = kb[predicate]['prem']
                            prem.append(sentence)
                            kb[predicate] = {'conc': conc, 'prem': prem}


    def variable(x):
        if not isinstance(x, str):
            return False
        else:
            if x.islower():
                return True
            else:
                return False

    def standardize_variables(knowledge_base):
        label = 0
        result_knowledge_base = []
        for rule in knowledge_base:
            variable_names = {}
            lhs = rule.partition('=>')[0]
            rhs = rule.partition('=>')[2]
            premise = []
            for x in lhs.split('^'):
                premise.append(x)
            result_premise = ""
            for term in premise:
                args = []
                result_term = "" + term.partition('(')[0]
                temp = term.partition('(')[-1].rpartition(')')[0]
                result_item = ""
                for item in temp.split(','):
                    args.append(item)
                    if variable(item):
                        if item not in variable_names:
                            variable_names[item] = "x" + repr(label)
                            item = "x" + repr(label)
                            label = label + 1
                        else:
                            item = variable_names[item]
```

```
290.                         result_item = result_item + item + ","
291.                     result_item = result_item[:len(result_item) - 1]
292.                     result_term = result_term + '(' + result_item + ')' + '^
    '
293.                     result_premise = result_premise + result_term
294.                 result_premise = result_premise[:len(result_premise) - 1]
295.
296.                 conclusion = []
297.                 for x in rhs.split('^'):
298.                     conclusion.append(x)
299.                 if conclusion != ['']:
300.                     result_premise = result_premise + "=>"
301.                     for term in conclusion:
302.                         args = []
303.                         result_term = "" + term.partition('(')[0]
304.                         temp = term.partition('(')[-1].rpartition(')')[0]
305.                         result_item = ""
306.                         for item in temp.split(','):
307.                             args.append(item)
308.                             if variable(item):
309.                                 if item not in variable_names:
310.                                     variable_names[item] = "x" + repr(label)

311.                                     item = "x" + repr(label)
312.                                     label = label + 1
313.                                 else:
314.                                     item = variable_names[item]
315.                             result_item = result_item + item + ","
316.                         result_item = result_item[:len(result_item) - 1]
317.                         result_term = result_term + '(' + result_item + ')'
    + '^'
318.                         result_premise = result_premise + result_term
319.                     result_premise = result_premise[:len(result_premise) - 1
    ]
320.
321.                 result_knowledge_base.append(result_premise)
322.             return result_knowledge_base
323.
324.         #Main
325.
326.         fn="input.txt"
327.         queries = []
328.         knowledge_base = []
329.         f1=open(fn, "r")
330.         input = f1.readlines()
331.         input = [x.strip() for x in input]
332.
333.         for i in range(1, int(input[0]) + 1):
334.             queries.append(input[i].replace(" ", ""))
335.         for i in range(int(input[0]) + 2, int(input[int(input[0]) + 1]) + in
    t(input[0]) + 2):
336.             knowledge_base.append(input[i].replace(" ", ""))
337.         knowledge_base = standardize_variables(knowledge_base)
338.
339.         kb = {}
340.         list_of_predicates = []
341.         add_to_kb(knowledge_base)
342.
343.         fileOut = open("output.txt", "w")
```

```
    for query in queries:
        result = fol_bc_ask(query, {})
        if result != None:
            print("True", result)
            fileOut.write("TRUE" + "\n")
        else:
            print("False", result)
            fileOut.write("FALSE" + "\n")

    fileOut.close()
    f1.close
```

Output File:
TRUE

Here our output file contain, either our queries which given in input is True or False. The step wise process will be shown in command prompt.

Github Link: https://github.com/nowshad7/BackwardChaining
Reference: https://homepage.cs.uri.edu/~cingiser/csc481/chapter_notes/amarant.pdf