CSE 5360 AI-1
ASSIGNMENT – 7

ReadMe

Name: Sagar Sharma
UTA ID: 1001626958

Note: The following files should be kept in the same directory:

1. check_true_false.py
2. logical_expression.py
3. logical_expression.cpy
4. wumpus_rules.txt
5. additional_kb.txt (could be any kb following correct syntax)
6. statement.txt (could be any statement to be checked for truth value, following correct syntax)

Commandline to Run the program:

check_true_false wumpus_rules.txt [additional_knowledge_file] [statement_file]

E.g. : python check_true_false.py wumpus_rules.txt kb3.txt statement1.txt

INFERENCE ENGINE:

Note: My implementation of IE involves XOR.
Note: Programming language used: Python 2.4

Code containing Functions for all the 6 connectives used alongside the propositional logic for writing the KB and statement which is checked for truth value(entailment):

1. and
2. or
3. if
4. iff
5. xor
6. not

Code starts from here:

```
import copy
#actualModel = {}


def getSymbols(expression): # get the symbols from the wumpus rules, additional kb and statement files
    symbols=[] #create a list
#if it is a base case(symbol)
    if expression.symbol[0]:
        symbols.append(expression.symbol[0]) #append the subsequent symbols of the logical expression
    else: #otherwise it is a subexpression
        for subexpression in expression.subexpressions:
            for symbol in getSymbols(subexpression):
                if symbol not in symbols: # if symbol in subexpression not appended
                    symbols.append(symbol) # append that symbol
    return symbols


def getModel(statement):  # get model for the statement's truth values
    model = {};
    for expression in statement.subexpressions:
        if expression.symbol[0]: # for true symbols
            model[expression.symbol[0]] = True
        elif expression.connective[0].lower() == 'not': # for false symbols
            if expression.subexpressions[0].symbol[0]:
                model[expression.subexpressions[0].symbol[0]] = False


    return model
```

```
def extendModel(model,symbol,value): #extend the model for all the symbols
    newModel = copy.deepcopy(model)
    newModel[symbol] = value
    return newModel

def plTrue(statement,model):
    #print_expression(statement,"")

    if statement.symbol[0]:
        return model[statement.symbol[0]]
# and connective
    elif statement.connective[0].lower() == 'and':
        result = True
        for exp in statement.subexpressions:
            result = result and plTrue(exp,model)
        return result
# or connective
    elif statement.connective[0].lower() == 'or':
        result = False
        for exp in statement.subexpressions:
            result = result or plTrue(exp,model)
        return result
# xor connective
    elif statement.connective[0].lower() == 'xor':
        result = False
        for exp in statement.subexpressions:
            isExpTrue = plTrue(exp,model)
            result = (result and not isExpTrue) or (not result and isExpTrue)
        return result
```

```
# if connective
    elif statement.connective[0].lower() == 'if':
        left = statement.subexpressions[0]
        right = statement.subexpressions[1]
        isLeftTrue = plTrue(left,model)
        isRightTrue = plTrue(right,model)
        if( isLeftTrue and not isRightTrue):
            return False
        else:
            return True
# iff connective
    elif statement.connective[0].lower() == 'iff':
        left = statement.subexpressions[0]
        right = statement.subexpressions[1]
        isLeftTrue = plTrue(left,model)
        isRightTrue = plTrue(right,model)
        if(isLeftTrue == isRightTrue):
            return True
        else:
            return False
# not connective
    elif statement.connective[0].lower() == 'not':
        return not plTrue(statement.subexpressions[0],model)
# checking for truth value of the statement wrt kb
def check_true_false(knowledge_base, statement):
    model = getModel(knowledge_base)
    symbols = getSymbols(knowledge_base)
    for symbol in getSymbols(statement):
        symbols.append(symbol)
```

```
    # remove symbols that are already present in the model
    for symbol in model:
        if symbol in symbols:
            symbols.remove(symbol)
    # print symbols

    truthOfStatement = TTCheckAll(knowledge_base, statement, symbols, model)
# checking the truth value of the statement and the negation of the statement

    negation = logical_expression()
    negation.connective[0] = 'not'
    negation.subexpressions.append(statement)

    truthOfNegation = TTCheckAll(knowledge_base, statement, symbols, model)
# checking the truth value of the negation of the statement
# writing the results in the results.txt file after checking the truth value for the statement and negation of statement
    result = open("result.txt","w+")
    if truthOfStatement and not truthOfNegation:
        result.write("definitely true")
    elif not truthOfStatement and truthOfNegation:
        result.write("definitely false")
    elif not truthOfStatement and not truthOfNegation:
        result.write("possibly true, possibly false")
    elif truthOfStatement and truthOfNegation:
        result.write("both true and false")
# ttchecksall checks for the truth value of the kb , the statement and negation of the statement in all the possible models
```

```
def TTCheckAll(KB,statement, symbols, model):
    if not symbols:
        print "truth of statement is: %s" %plTrue(statement,model)
        if plTrue(KB,model): # check for truth values of the kb
            print"Kb is true"
            return plTrue(statement,model)
        else:
            return True

    else:
        p =symbols.pop(0)

        return TTCheckAll(KB, statement, symbols, extend(model,p,True)) and TTCheckAll(KB, statement, symbols, extend(model,p,False))
def extend(model,symbol,value):
    model[symbol] = value
    return model
```

Wumpus World Rules based on the description given of this specific Wumpus world problem statement

#rule 1: monster <=> stench

(iff M_1_3 (and S_1_2 S_1_4 S_2_3))

(iff M_1_4 (and S_1_3 S_2_4))

(iff M_2_3 (and S_2_4 S_2_2 S_3_3 S_1_3))

(iff M_2_4 (and S_2_3 S_3_4 S_1_4))

(iff M_3_1 (and S_3_2 S_4_1 S_2_1))

(iff M_3_2 (and S_3_1 S_3_3 S_2_2 S_4_2))

(iff M_3_3 (and S_3_2 S_3_4 S_2_3 S_4_3))

(iff M_3_4 (and S_3_3 S_2_4 S_4_4))

(iff M_4_1 (and S_4_2 S_3_1))

(iff M_4_2 (and S_4_1 S_4_3 S_3_2))

(iff M_4_3 (and S_4_2 S_4_4 S_3_3))

(iff M_4_4 (and S_4_3 S_3_4))

#rule 2: stench <=> monster

(iff S_1_2 (xor M_1_3 M_2_2))

(iff S_1_3 (xor M_1_4 M_2_3))

(iff S_1_4 (xor M_1_3 M_2_4))

(iff S_2_1 (xor M_2_2 M_3_2))

(iff S_2_2 (xor M_2_3 M_3_2))

(iff S_2_3 (xor M_2_4 M_2_2 M_1_3 M_3_3))

(iff S_2_4 (xor M_2_3 M_1_4 M_3_4))

(iff S_3_1 (xor M_3_2 M_2_1 M_4_1))

(iff S_3_2 (xor M_3_1 M_3_3 M_4_2 M_4_3))

(iff S_3_3 (xor M_3_2 M_3_4 M_2_3 M_4_3))

(iff S_3_4 (xor M_3_3 M_4_4 M_2_4))

(iff S_4_1 (xor M_4_2 M_3_1))

(iff S_4_2 (xor M_4_3 M_4_1 M_3_2))

(iff S_4_3 (xor M_4_4 M_4_2 M_3_3))

(iff S_4_4 (xor M_3_4 M_4_3))

#rule 3: pit <=> breeze

(iff P_1_3 (and B_1_2 B_1_4 B_2_3))

(iff P_1_4 (and B_1_3 B_2_4))

(iff P_2_3 (and B_2_4 B_2_2 B_3_3 B_1_3))

(iff P_2_4 (and B_2_3 B_3_4 B_1_4))

(iff P_3_1 (and B_3_2 B_4_1 B_2_1))

(iff P_3_2 (and B_3_1 B_3_3 B_2_2 B_4_2))

(iff P_3_3 (and B_3_2 B_3_4 B_2_3 B_4_3)) (iff P_3_4 (and B_3_3 B_2_4 B_4_4))

(iff P_4_1 (and B_4_2 B_3_1))

(iff P_4_2 (and B_4_1 B_4_3 B_3_2))

(iff P_4_3 (and B_4_2 B_4_4 B_3_3))

(iff P_4_4 (and B_4_3 B_3_4))

#rule 4: breeze <=> pit

(iff B_1_2 (or P_1_3 P_2_2))

(iff B_1_3 (or P_1_4 P_2_3))

(iff B_1_4 (or P_1_3 P_2_4))

(iff B_2_1 (or P_2_2 P_3_2))

(iff B_2_2 (or P_2_3 P_3_2))

(iff B_2_3 (or P_2_4 P_2_2 P_1_3 P_3_3))

(iff B_2_4 (or P_2_3 P_1_4 P_3_4))

(iff B_3_1 (or P_3_2 P_2_1 P_4_1))

(iff B_3_2 (or P_3_1 P_3_3 P_4_2 P_4_3))

(iff B_3_3 (or P_3_2 P_3_4 P_2_3 P_4_3))

(iff B_3_4 (or P_3_3 P_4_4 P_2_4))

(iff B_4_1 (or P_4_2 P_3_1))

(iff B_4_2 (or P_4_3 P_4_1 P_3_2))

(iff B_4_3 (or P_4_4 P_4_2 P_3_3))

(iff B_4_4 (or P_3_4 P_4_3))

#rule 5: only one monster

(xor M_1_3 M_1_4 M_2_3 M_2_4 M_3_1 M_3_2 M_3_3 M_3_4 M_4_1 M_4_2 M_4_3 M_4_4)

#rule 6: start squares are safe

(not M_1_1)

(not M_1_2)

(not M_2_1)

(not M_2_2)

(not P_1_1)

(not P_1_2)

(not P_2_1)

(not P_2_2)

#rule 7: all squares can't be pits

(not (and P_1_3 P_1_4 P_2_3 P_2_4 P_3_1 P_3_2 P_3_3 P_3_4 P_4_1 P_4_2 P_4_3 P_4_4))

#rule 8: atleast one square is a pit

(xor P_1_3 P_1_4 P_2_3 P_2_4 P_3_1 P_3_2 P_3_3 P_3_4 P_4_1 P_4_2 P_4_3 P_4_4)