Lab Report -2

Name: Md Yeasin Arafat

ID: 211-35-687

Section: A

Course name: Artificial Intelligence Lab

Course Code: SE334

Date: 12/11/2023

Problem Statement.

In problem statement, while we were talking about the problem in class. We notice that we have to come up with a solution. where the weathere is rainy or not based on some information we have.

In our problem,

we know that if it didnot rain, then Harry will visit Hagrid today. Instead of Hagrid harry visited Dumbledore today. But not both. Then we know that Harry visited Dumbledore today.

From above information, we have to find the weather status. And to find that information, we have to use the term knowledge-based agents. which will help us to find the correct information based on some given information.

The knowledge-based agents, that reason by operating on internal representations of knowledge.

The agent will work on some logic and come up with a solution that will be a correct information. To find out the correct information we will use propositional logic where the agent will make a model for possible world. By using that model, the agent will come up with a solution. And the being checked by the agent is called model checking algorithm.

Based on model checking algorithm, ~~the~~ agent will find the correct solution. That would be the correct information.

```python
import itertools
class Sentence():

    def evaluate(self, model):
        """Evaluates the logical sentence."""
        raise Exception("nothing to evaluate")

    def formula(self):
        """Returns string formula representing logical sentence."""
        return ""

    def symbols(self):
        """Returns a set of all symbols in the logical sentence."""
        return set()

    @classmethod
    def validate(cls, sentence):
        if not isinstance(sentence, Sentence):
            raise TypeError("must be a logical sentence")

    @classmethod
    def parenthesize(cls, s):
        """Parenthesizes an expression if not already parenthesized."""
        def balanced(s):
            """Checks if a string has balanced parentheses."""
            count = 0
            for c in s:
                if c == "(":
                    count += 1
                elif c == ")":
                    if count <= 0:
                        return False
                    count -= 1
            return count == 0
        if not len(s) or s.isalpha() or (
            s[0] == "(" and s[-1] == ")" and balanced(s[1:-1])
        ):
            return s
        else:
            return f"({s})"


class Symbol(Sentence):

    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return isinstance(other, Symbol) and self.name == other.name

    def __hash__(self):
        return hash(("symbol", self.name))

    def __repr__(self):
        return self.name

    def evaluate(self, model):
        try:
            return bool(model[self.name])
        except KeyError:
            raise EvaluationException(f"variable {self.name} not in model")

    def formula(self):
        return self.name

    def symbols(self):
        return {self.name}
```

```python
class Not(Sentence):
    def __init__(self, operand):
        Sentence.validate(operand)
        self.operand = operand

    def __eq__(self, other):
        return isinstance(other, Not) and self.operand == other.operand

    def __hash__(self):
        return hash(("not", hash(self.operand)))

    def __repr__(self):
        return f"Not({self.operand})"

    def evaluate(self, model):
        return not self.operand.evaluate(model)

    def formula(self):
        return "¬" + Sentence.parenthesize(self.operand.formula())

    def symbols(self):
        return self.operand.symbols()


class And(Sentence):
    def __init__(self, *conjuncts):
        for conjunct in conjuncts:
            Sentence.validate(conjunct)
        self.conjuncts = list(conjuncts)

    def __eq__(self, other):
        return isinstance(other, And) and self.conjuncts == other.conjuncts

    def __hash__(self):
        return hash(
            ("and", tuple(hash(conjunct) for conjunct in self.conjuncts))
        )

    def __repr__(self):
        conjunctions = ", ".join(
            [str(conjunct) for conjunct in self.conjuncts]
        )
        return f"And({conjunctions})"

    def add(self, conjunct):
        Sentence.validate(conjunct)
        self.conjuncts.append(conjunct)

    def evaluate(self, model):
        return all(conjunct.evaluate(model) for conjunct in self.conjuncts)

    def formula(self):
        if len(self.conjuncts) == 1:
            return self.conjuncts[0].formula()
        return " ∧ ".join([Sentence.parenthesize(conjunct.formula())
                            for conjunct in self.conjuncts])

    def symbols(self):
        return set.union(*[conjunct.symbols() for conjunct in self.conjuncts])


class Or(Sentence):
    def __init__(self, *disjuncts):
        for disjunct in disjuncts:
            Sentence.validate(disjunct)
        self.disjuncts = list(disjuncts)
```

```
def __eq__(self, other):
    return isinstance(other, Or) and self.disjuncts == other.disjuncts

def __hash__(self):
    return hash(
        ("or", tuple(hash(disjunct) for disjunct in self.disjuncts))
    )

def __repr__(self):
    disjuncts = ", ".join([str(disjunct) for disjunct in self.disjuncts])
    return f"Or({disjuncts})"

def evaluate(self, model):
    return any(disjunct.evaluate(model) for disjunct in self.disjuncts)

def formula(self):
    if len(self.disjuncts) == 1:
        return self.disjuncts[0].formula()
    return " ∨  ".join([Sentence.parenthesize(disjunct.formula())
                        for disjunct in self.disjuncts])

def symbols(self):
    return set.union(*[disjunct.symbols() for disjunct in self.disjuncts])
class Implication(Sentence):
    def __init__(self, antecedent, consequent):
        Sentence.validate(antecedent)
        Sentence.validate(consequent)
        self.antecedent = antecedent
        self.consequent = consequent

    def __eq__(self, other):
        return (isinstance(other, Implication)
            and self.antecedent == other.antecedent
            and self.consequent == other.consequent)

    def __hash__(self):
        return hash(("implies", hash(self.antecedent), hash(self.consequent)))

    def __repr__(self):
        return f"Implication({self.antecedent}, {self.consequent})"

    def evaluate(self, model):
            return ((not self.antecedent.evaluate(model))
            or self.consequent.evaluate(model))

    def formula(self):
        antecedent = Sentence.parenthesize(self.antecedent.formula())
        consequent = Sentence.parenthesize(self.consequent.formula())
           return f"{antecedent} => {consequent}"

    def symbols(self):
        return set.union(self.antecedent.symbols(), self.consequent.symbols())
class Biconditional(Sentence):
    def __init__(self, left, right):
        Sentence.validate(left)
        Sentence.validate(right)
        self.left = left
        self.right = right

    def __eq__(self, other):
            return (isinstance(other, Biconditional)
            and self.left == other.left
            and self.right == other.right)

    def __hash__(self):
        return hash(("biconditional", hash(self.left), hash(self.right)))
```

```
def model_check(knowledge, query):
    """Checks if knowledge base entails query."""

    def check_all(knowledge, query, symbols, model):
        """Checks if knowledge base entails query, given a particular model."""

        # If model has an assignment for each symbol
        if not symbols:

            # If knowledge base is true in model, then query must also be true
            if knowledge.evaluate(model):
                return query.evaluate(model)
            return True
        else:

            # Choose one of the remaining unused symbols
            remaining = symbols.copy()
            p = remaining.pop()

            # Create a model where the symbol is true
            model_true = model.copy()
            model_true[p] = True

            # Create a model where the symbol is false
            model_false = model.copy()
            model_false[p] = False

            # Ensure entailment holds in both models
            return (check_all(knowledge, query, remaining, model_true) and
                    check_all(knowledge, query, remaining, model_false))

    # Get all symbols in both knowledge and query
    symbols = set.union(knowledge.symbols(), query.symbols())

    # Check that knowledge entails query
    return check_all(knowledge, query, symbols, dict())
```

Harry.py

```
from logic import *

rain = Symbol("rain")
hagrid = Symbol("hagrid")
dumbledore = Symbol("dumbledore")

knowledge = And(
    Implication(Not(rain), hagrid),
    Or(hagrid, dumbledore),
    Not(And(hagrid, dumbledore)),
    dumbledore
)

rain = (model_check(knowledge, rain))
if rain:
    print("It rained today!")
else:
    print("It didn't rain!")

# print(model_check(knowledge, rain))
```

Explanation of code.

Explanation of logic:

Sentence class

→ This class appears to be a base class for representing logical sentences in some formal logic system.

evaluate(self, model)

→ This method is intended to evaluate the logical sentence using a given model. If the logic in this ~~class~~ method not matched then there will be an exception.

formula(self)

→ This method returns a string formula representing the logical sentence. The current implementation returns an empty string.

symbols (self)

→ This method returns a set of all symbols in the logical sentence. The current implementation returns an empty set.

validate (cls, sentence)
→ This class method validates weather a given object is logical sentence. It raises a 'TypeError' if the provided object is not an istance of the 'Sentence' class. This is a way to enforce that any input to methods expecting logical sentence must be instances of this class or its subclass.

parenthesize (cls,s)
→ This class method parenthesizes an expression if it is not already pareenthesized. It checks if the input string 's' is already parenthesized or if It's a single alphnumeric character.

Symbol class

~~this class~~ This class appears to be a subclass of the previously mentioned 'Sentence' class. It represents a symbol in a logical sentence, typically corresponding to a variable in a logical formula.

In this class we have constructor which initialize a 'symbol' object with a given name.

"--eq--" which returns a comparison

'--hash--' Overrides the hash function for instances of this class.

'--repr--' Overrides the representation of the object when using 'repr()'

'evalate(self, model)' Evaluate the logical value of the symbol based on a given model.

'formula(self)' returns the string formula representation of the symbol.

'symbol (self)' returns a set containing the name of the symbol.

'Not' class

Represents the logical NOT operation.

'operand': The logical sentence being negated.

Methods:

- evaluate(model): Evaluates the logical value of the negation based on a given model.
- formula() : Returns the string formula representation of the negation
- symbols (): Returns a set of all symbols in the negation.

'And' class

Represents the logical AND operation.

conjuncts: A list of logical sentences being conjoined.

Methods:

- evaluate(model): Evaluates the logical value of the conjunction based on a given model.
- formula(): Returns the string formula of the conjunation
- symbols(): Returns a set of all symbols
- add(conjunct): Adds a new conjunct in the conjunction.

'OR' class

Represent the logical OR operation.

disjuncts: A list of logical sentence being disjoined.

Methods:

- evaluate (model): Evaluate the logical value of the disjunction based on a given model.
- formula(): Returns the string formula.
- symbols (): Returns a set of all symbols in the disjunction.

'Implication' class

Represent the logical implication operation.

- antecedent: The antecedent (left side) of the implication
- consequent: The consequent (right side) of the implication.

Methods:

- evaluate(model): Evaluates logical value of the implication based on a given model.
- formula(): Returns the string formula of the implication
- symbols (): Returns a set of all symbols.

'Biconditional' class

Represents the logical biconditional operation.

- left: the left side of the biconditional
- Right: The right side of the biconditional

Method:

- evaluate(model): Evaluates the logical value of the biconditional based on a given model.
- formula(): Returns the string formula of the biconditional
- Symbols(): Returns a set of all symbols.

'model_check (knowledge, query)' that checks if a given knowledge base entails a specific query in the context of a model. the function utilizes a recursive helper function 'check_all (knowledge, query, symbols, model)'.

'model_check' Function

checks If the knowledge based entails the query.

Parameters:

- Knowledge: A logical sentence representing the knowledge base.

• query: A logical sentence representing the query.

Returns 'True' if the knowledge base entails the query; otherwise returns 'false'.

'check_all' Helper function.

checks entailment recursively for all possible models.

Parameter:

• knowledge: A logical sentence representing the knowledge base.
• query: A logical sentence representing query.
• symbols: A set of symbols present in both knowledge and query.
• model: A dictionary representing the current model.

Return 'True' if knowledge entails query in all possible models; otherwise, returns 'False.'

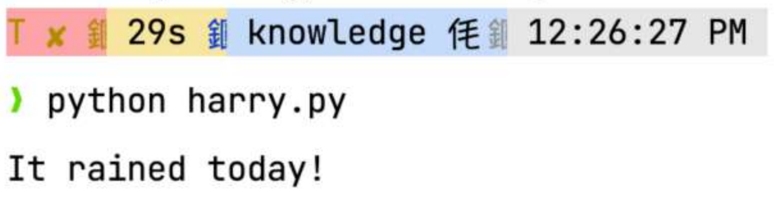
harry.py

In this code space, at first we are importing logic from logic.py. Then we define three symbols, rain, hagrid, and dumbledore. the knowledge base (knowledge) is build using logical statements about these symbols.

Knowledge Base, where we check the symbol, using Implication, And, Or, Not.
Model checking function to check weather if it rained or not based on the knowledge base.

Print if rained or not.

Output of the **harry.py** which use the **logic.py** which representes the knowledge base, model checking algorihtm of knowledge representation.
According to the given knowledge **‘It rained today’**.

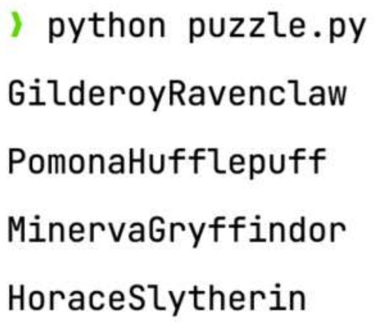


```
T x 釗 29s 釗 knowledge 侂 釗 12:26:27 PM

❯ python harry.py

It rained today!
```

Output of the **puzzle.py** which use the **logic.py** which representes the knowledge base, model checking algorihtm of knowledge representation.
According to the given knowledge.



```
❯ python puzzle.py

GilderoyRavenclaw

PomonaHufflepuff

MinervaGryffindor

HoraceSlytherin
```