Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Artificial Intelligence Fundamentals

Laboratory work #1

Expert Systems

Author:
Elena Papuc
std. gr. FAF-201

Supervisor:
Diana Marusic

Chișinău 2023

1 Results

In order to create an expert system, first of all, we need to create a goal tree. A Goal Tree is primarily a tool for rational analysis of all prerequisites i.e. Necessary Conditions to achieve a Goal, and their dependencies [1]. The Goal Tree is represented in Figure 1.

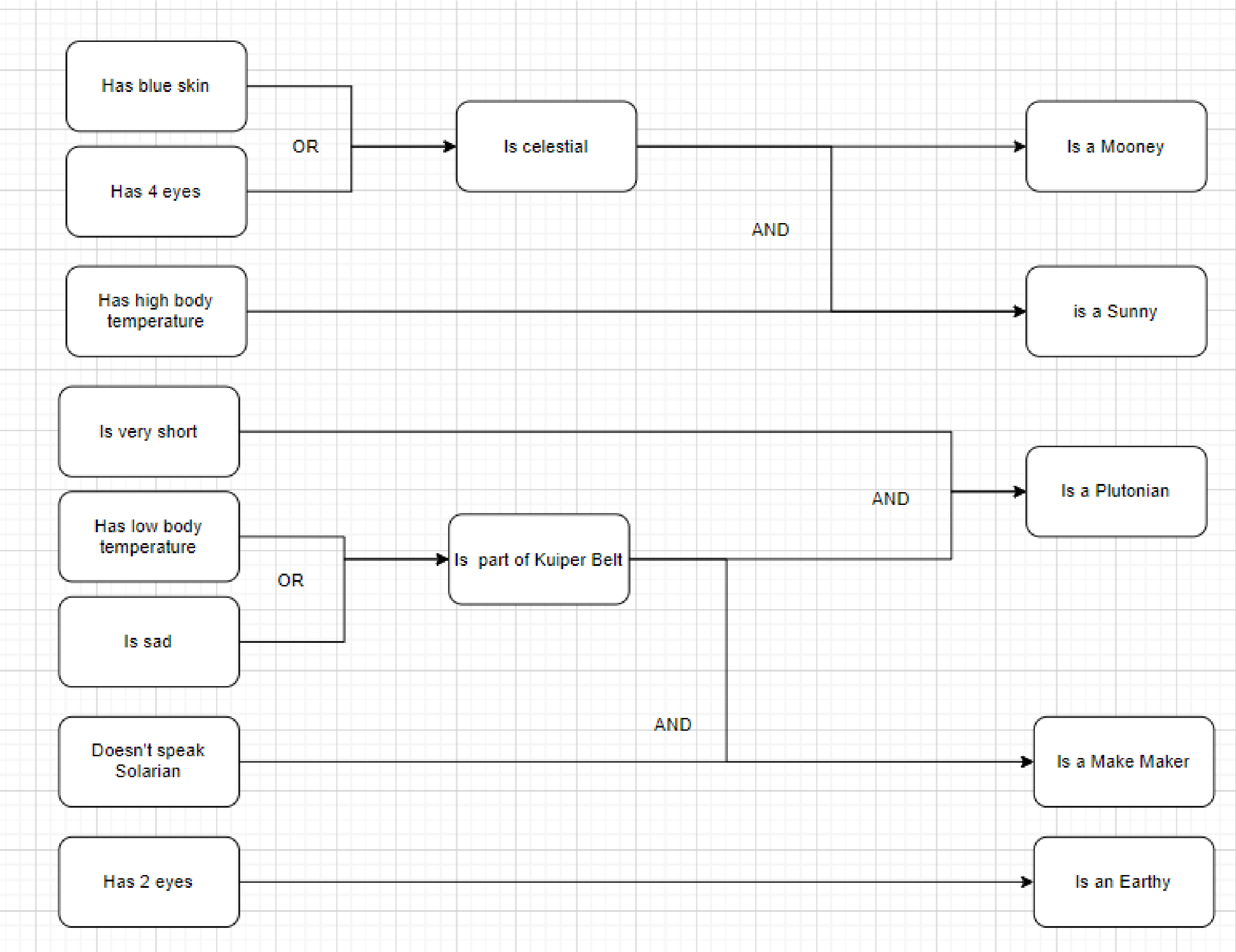

Figure 1: Goal Tree for Tourist Expert System

Based on the goal tree, I created a set of rules that I wrote into the structure represented in Listing 1. The IF, OR, AND classes were already defined.

```
RULES = (

    IF(OR('(?x) has blue skin',
          '(?x) has 4 eyes'),
       THEN('(?x) is celestial')),

    IF(AND('(?x) is celestial'),
       THEN('(?x) is a Mooney')),

    IF(AND('(?x) is celestial',
           '(?x) has high body temperature'),
       THEN('(?x) is a Sunny')),

```

```
    IF(AND('(?x) is a Mooney',
           '(?x) has high body temperature'),
       THEN('(?x) is a Sunny')),

    IF(OR('(?x) has low body temperature',
          '(?x) is sad'),  # Z3
       THEN('(?x) is part of the Kuiper Belt')),

    IF(AND('(?x) is part of the Kuiper Belt',
           '(?x) is very short'),
        THEN('(?x) is a Plutonian')),

    IF(AND('(?x) has 2 eyes'),
       THEN('(?x) is an Earthy')),

    IF(AND('(?x) is part of the Kuiper Belt',
           '(?x) does not speak Solarian'),
       THEN('(?x) is a Make Maker')),

)
```

Listing 1: Rules for Tourist Expert System

In order to achieve forward chaining, I used the already defined function, I added to the apply() function a block of code to be able to delete the facts that don't show the final result.

I added to the apply() method in order to be able to remove the rules that participated in the discovery of a new rule, for this I created new bindings between the antecedent rules and the existing rules, then, I populated them with the character name and created a set for deletion. This is represented in Listing 2.

```
...
bind = RuleExpression().test_term_matches(self.antecedent(), new_rules)
                for c in bind:
                    for b in self.antecedent():
                        to_delete.add(populate(b, c))
...
```

Listing 2: Deletion of Antecedent Rules

In Listing **??** I represented the data used to check the forward chaining.

```
DATA = (
    'Mark has low body temperature',
    'Mark is sad',
    'Mark does not speak Solarian',
    )
```

Listing 3: Data to check forward chaining

The result can be seen in Figure 2

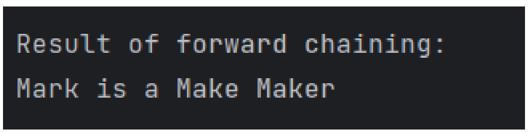

Figure 2: Forward Chaining Result

I also implemented backward chaining. Backward chaining acts as an encyclopedia. We have a hypothesis, and based on that, we go backwards into the goal tree and see all the facts that lead up to our hypothesis.

I implemented this using a recursive function to check whether the nodes are related or not It does so by receiving as an input the initial hypothesis, treating it as a consequential, seeing if it matches any other consequentials, populating the antecedents of that matching rule with the matches, and adding them to a new set, then we check if the new facts are consequentials in themselves, until we have no more facts that are consequentials. The function is represented in Listing 4

```python
def recursive(facts):
    new_facts = set()
    for fact in facts:
        for rule in rules:
            for action in rule._action:
                match_result = match(action, fact)

                if match_result:

                    if isinstance(rule._conditional, AND):
                        for r in rule._conditional:
                            new_facts.add(populate(r, match_result))
                            final_facts.add(populate(r, match_result))
                    elif isinstance(rule._conditional, OR):
                        #if the statement is united through or, we can't be sure if
                        #the tourist posseses only one or both of the qualities
                        or_statement = ' EITHER/OR '.join(populate(r, match_result) for r in rule._conditional)
                        final_facts.add(or_statement)
                        for r in rule._conditional:
                            new_facts.add(populate(r, match_result))


                    bindings.append(match_result)
                    if verbose:
                        print("Matched:", action, "with", hypothesis)

    # Check if new facts were added
    if new_facts:
        # Recur with the new facts
        recursive(new_facts)
```

Listing 4: Recursive function for backward chaining

The result of backward chaining for the hypothesis "Mark is a Make Maker" is represented in Figure 3.

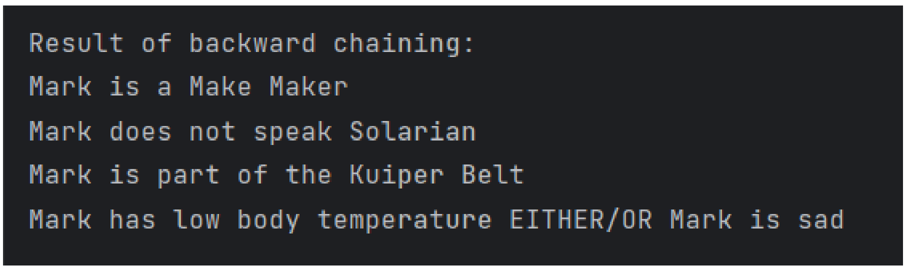

Figure 3: Backward chaining result

For the rules represented through OR, meaning that a tourist can have one of the characteristics, all of them or multiple, that's why I separate them through EITHER/OR.

To make it an interactive system for the backward chaining it is enough to ask the user a question about the name of the tourist, and one about the species they belong to. Also, I created a function that discovers whether the species starts with a vowel or consonant to be able to use the correct attribute. This is shown in Listing 5

```
#Function to see if the species starts with a vowel
def starts_with_vowel(word):
    # Convert the word to lowercase to handle both cases
    lower_word = word.lower()

    # Define a set of vowels
    vowels = {'a', 'e', 'i', 'o', 'u'}

    # Check if the first letter is a vowel
    return lower_word[0] in vowels

    ...
    #Question to find out the tourist's name
    tourist_name = input("What is the name of the tourist you are trying to find more information about?\n")
    #Question to find out the tourist's species
    tourist_species = input("What species is the tourist?\n1.Earthy\n2.Mooney\n3.Sunny\n4.Make Maker\n5.Plutonian\n")
    print("          ")

    #deciding which attribute to use
    if starts_with_vowel(tourist_species):
        attribute = "an"
    else:
        attribute = "a"

    #the hypothesis is formed based on the data that was gotten previously
    hypothesis = f"{tourist_name} is {attribute} {tourist_species}"
    print(f'Result of backward chaining: ')

    #backward chaining reveals a set of facts about the tourist
    backward_chain_data = backward_chain(RULES, hypothesis)

    for data in backward_chain_data:
        print(data)
```

Listing 5: Elements for asking questions about backward chaining

The result can be seen in Figure 4

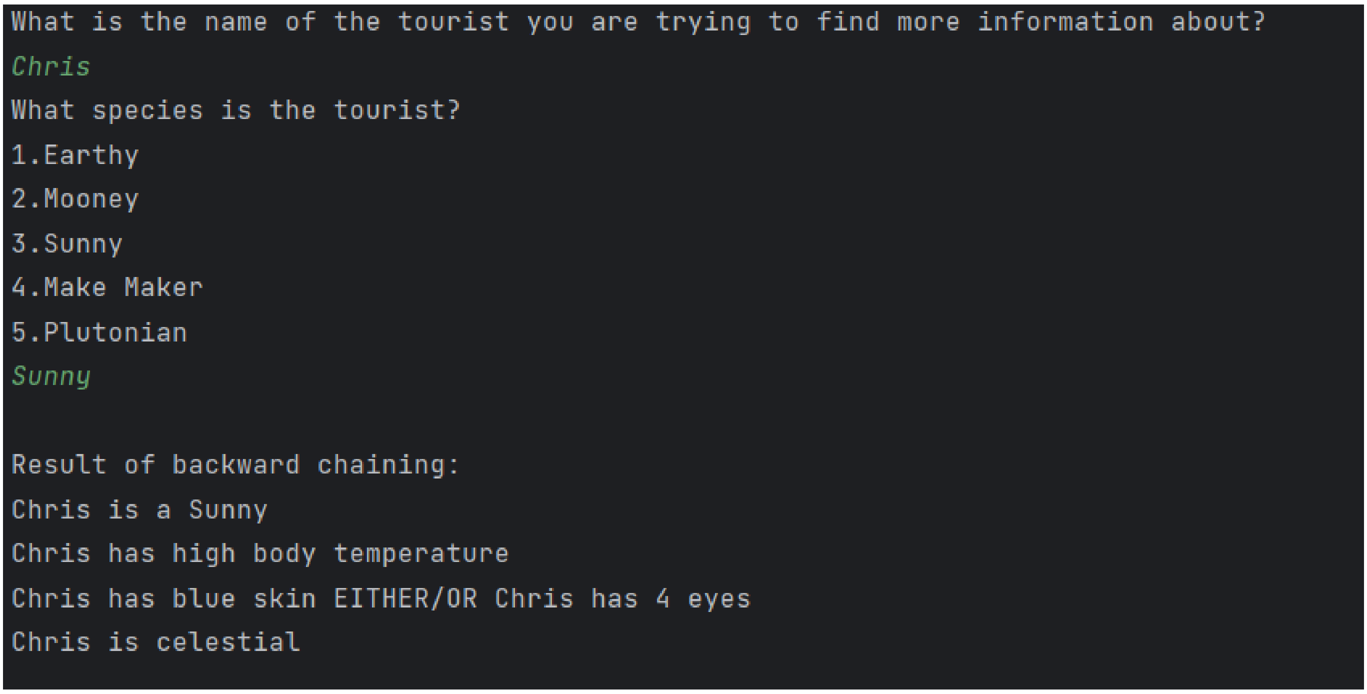

Figure 4: Backward chaining result with questions

The next listing Listing 6 showcases how questions are generated within the system. Each tree can have self generated questions. The explanations are in the commentaries

```
from production import *
from rules import RULES

#dictionary that unites statements with questions
dictionary = {}

#Method to check if there are any statements that could become multiple questions in the
#list, it returns a set of values that are not the subject or the verb of a sentence, and have more than 2 words
#i called them complements, they will help determine whether some sentences are similar in meaning
def check_for_multiple_choice():

    set_of_complements = []
    dict = {}
    for r in RULES:
        for statement in r.antecedent():
            words = statement.split()
            complement = ' '.join(words[2:])
            dict[complement] = statement

            if len(complement.split()) >= 2:
                set_of_complements.append(complement)

    return set_of_complements,dict
#taking the resulting complements and the dictionary, this function generates sentences that are multiple choice
#it takes the similar complements and unites them with 'or'. It then sends them to the question generator
def generate_multiple_choice():

    comp = check_for_multiple_choice()[0]
```

```

    similar_elements = find_similar_elements(comp)
    dict = check_for_multiple_choice()[1]

    intermediary_list = []
    for pair in similar_elements:
        sentence = dict[pair[0]]
        words = sentence.split()
        new_sentence = words[0] + ' ' + words[1] + ' '+' or '.join(pair) + ' or other '
        intermediary_list.append(new_sentence)
        for i in pair:
            dictionary[dict[i]] = new_sentence
    statement_to_question(intermediary_list)
    print(dictionary)

    return intermediary_list

#Method that takes complements and checks if they are similar in any way, this will help sort the sentences that can become
#multiple choice
def find_similar_elements(elements):
    similar_pairs = []


    # Iterate through the elements
    for i in range(len(elements)):
        for j in range(i + 1, len(elements)):
            # Split the elements into words and compare
            words1 = set(elements[i].split())
            words2 = set(elements[j].split())
            differing = words1.symmetric_difference(words2)


            # Check if the elements differ by only one word
            if len(differing) == 2:
                similar_pairs.append((elements[i], elements[j]))
            # Find and print similar pairs
    return similar_pairs




#The question generator takes sentences and generates questions,
#it also adds them to the dictionary of questions, each statement has a corresponding question
def statement_to_question(statement_list):

    questions = set()
    for statement in statement_list:
        words = statement.split()
        subject = words[0].capitalize()
        verb = words[1]
        complement = ' '.join(words[2:])
        if verb.lower() in ['is', 'am', 'are', 'was', 'were']:

            question = f"{verb.capitalize()} {subject} {complement}?"
        else:
            if verb == 'has':
                verb = 'have'
```

```python
                question = f"Does {subject} {verb} {complement}?"
            elif verb == 'does':
                question = f"Does {subject} {complement}?"
            else:
                question = f"Does {subject} {verb[:-1]} {complement}?"

        if statement in dictionary:
            dictionary.update({statement: question})
        for key in dictionary:
            if dictionary[key] == statement:

                dictionary.update({key : question})

    print(dictionary)
    return questions

def create_questions():
    generate_multiple_choice()
    list_of_statements = []
    for rule in RULES:
        for a in rule.antecedent():
            if a not in dictionary:
                dictionary[a] = 'None'
                list_of_statements.append(a)
    statement_to_question(list_of_statements)
    return dictionary
```

Listing 6: Generating questions

In Figure 5 we can see the obtained questions.

```
(?x) has 4 eyes : Does (?x) have 4 eyes or 2 eyes or other?
(?x) has 2 eyes : Does (?x) have 4 eyes or 2 eyes or other?
(?x) has high body temperature : Does (?x) have high body temperature or low body temperature or other?
(?x) has low body temperature : Does (?x) have high body temperature or low body temperature or other?
(?x) has blue skin : Does (?x) have blue skin?
(?x) is celestial :  Is (?x) celestial?
(?x) is sad :  Is (?x) sad?
(?x) is part of the Kuiper Belt :  Is (?x) part of the Kuiper Belt?
(?x) is very short :  Is (?x) very short?
(?x) does not speak Solarian : Does (?x) not speak Solarian?
```

Figure 5: Dictionary

References

[1] CHRIS HOHMANN, What is a Goal Tree?, *https://hohmannchris.wordpress.com/2014/03/07/what-is-a-goal-tree/*