MINISTRY OF EDUCATION OF REPUBLIC OF MOLDOVA
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software Engineering and Automatics

Report

on Artificial Intelligence Fundamentals
Laboratory Work nr. 1

Performed by: **Sarmaniuc Marius**, FAF-172
Verified by: **Mihail Gavrilița**, asist. univ.

Chișinău, 2021

Contents

The Task

Implement a system that will detect Luna-City tourists. Based on a set of queries the system must detect what creature the user queries (forward chaining). Also it must be able to describe a creature that the user requests (backward chaining).

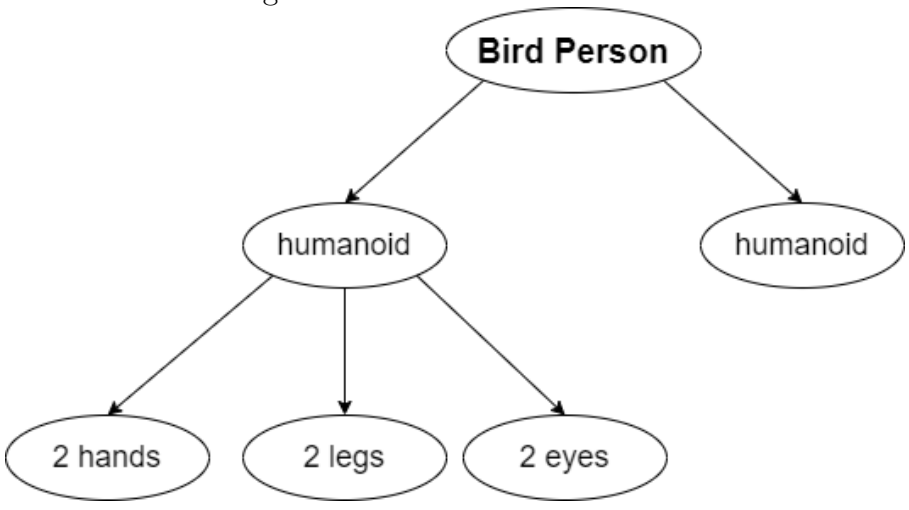
Solution Description

It was decided that the system will detect creatures based on a set of rules. For example for a creature named "Bird Person" the database will contain these rules:

- Is humanoid
  - Has two hands
  - Has two legs
  - Has two eyes
- Is flying
- Is white

Initially was tried the simplest method, to store rules in an array. But it turned out to be very tedious. And a different approach was considered where for each creature a tree of rules was built. Visually it looks like this:

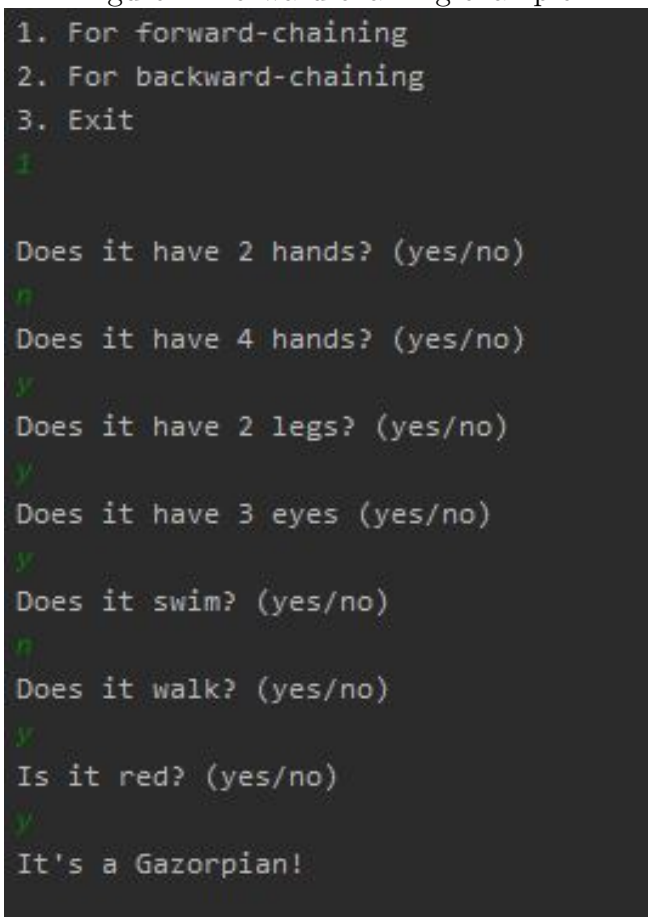
Figure 1: Bird Person Rule Tree



With this approach the system stores all creatures in an array and linearly searches through each creature. For each creature it traverses the tree of rules and gives questions based on these rules. All answers are stored because creatures share same rules and it will be a waste of compute time and user time to always ask the same question multiple times.

The result looks like this:

Figure 2: Forward chaining example



Code and Mentions

All the code is written in a single file: "main.py". The system is composed of a couple of classes and the most important one is the Rule class because it is the structure that will store the rules for each creature.

Listing 1: Rule class definition

```
class Rule:
    def __init__(self, fact: Fact, condition: str = "AND", *rules):
        self.fact = fact
        self.condition = condition
        if len(rules) != 0:
            self.rules = rules
        else:
            self.rules = None
```

You can see that the Rule class contains the root of the tree and a list of nodes. The condition variable stores the type of a given rule. An OR rule or an AND rule.

Below is shown an example that declares a rule:

Listing 2: Storing Rules

```
humanoid = Rule(Fact("humanoid", "Does it look like a humanoid?", "A humanoid"),
                "AND",
                Rule(Fact("2 hands","Does it have 2 hands?", "It has 2 hands")),
                Rule(Fact("2 legs", "Does it have 2 legs?", "It has 2 legs")),
                Rule(Fact("2 eyes", "Does it have 2 eyes?", "It has 2 eyes")))
```

With the chosen approach the implementation for backward chaining is very simple, just a tree traversal:

Listing 3: Backward Chaining

```
def backward_chaining(self, depth = 0):
    for i in range(depth):
        print("\t", end='')
    print(self.fact.affirmation)
    if self.rules is None:
        return
    for rule in self.rules:
        rule.backward_chaining(depth + 1)
```

Forward chaining is also pretty easy (some code was omitted as it will be explained later):

Listing 4: Forward Chaining

```
def forward_chaining(self) -> bool:
    ...

    if self.rules is None:
        response = get_user_input(self.fact.question + " (yes/no)\n")
        ...
        return response

    if self.condition == "AND":
        for rule in self.rules:
            response = rule.forward_chaining()
            if response is False:
                ...
                return False
    elif self.condition == "OR":
        for rule in self.rules:
            response = rule.forward_chaining()
            if response is False:
                ...
                return False

    known_yes_facts.append(self.fact)
```

```
23      return True
```

As you can see, it's a simple tree traversal. At each node the system asks the user a question. If some fact is responded with "no" the system will disregard this creature and go to the next one.

There is an evident problem with this simple solution. The system doesn't remember what questions it gave and it doesn't remember what answers it was given. This is why two lists were used. In one of them the system stores affirmative facts and in the second negative facts.

Listing 5: Response Cache

```
1 # user answered with YES for these facts
2 known_yes_facts = []
3 # user answered with NO for these facts
4 known_no_facts  = []
```

On each response the system stores the answer like this:

Listing 6: Response Caching

```
1 if self.rules is None:
2     response = get_user_input(self.fact.question + "␣(yes/no)\n")
3     if response is True:
4         known_yes_facts.append(self.fact)
5     else:
6         known_no_facts.append(self.fact)
7     return response
```

With the answers stored while traversing the Tree of Rules, known facts can be omitted.

Listing 7: Backtracking

```
1 if self.fact in known_no_facts:
2     return False
3 if self.fact in known_yes_facts:
4     return True
5 for yes_fact in known_yes_facts:
6     if self.fact.equal_type(yes_fact) is True:
7         if self.fact.equal_name(yes_fact) is False:
8             return False
```

Conclusions

In this laboratory work a simple Expert System was developed in Python. As a result the first practical steps in artificial intelligence were made. Also a little bit of knowledge engineering was applied while developing the database. And a little bit of classical algorithms and data structures were applied.

At the moment it is tedious for a non-technical person to add new knowledge to the database but the system can be extended with a domain specific language and the core engine may remain unchanged.

References

[1] Sarmaniuc Marius. Source code for the laboratory work. https://github.com/SarmaniucMarius/FIA/tree/master/lab1.