Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Lucrare de laborator

Disciplina: *Inteligență Artificială*

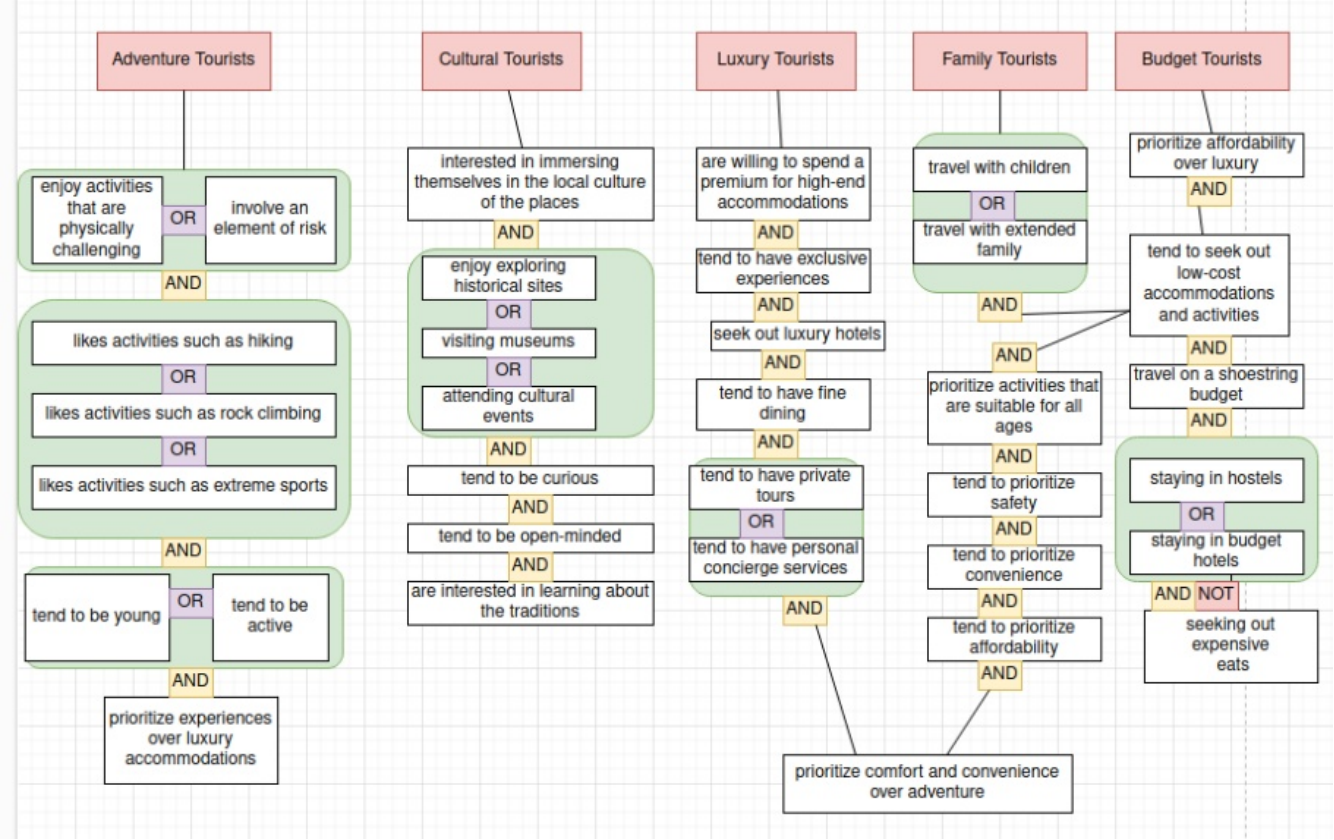
Tema: *Expert Systems*

A efectuat: st. gr. SI-221M Postu Ivan

A verificat: Gavrilița Mihail

Chișinău – 2023

Task 1 - Define 5 types of tourists that visit Luna-City. Draw the Goal Tree representing these types of tourists.



Task 2 - Implement the rules from the defined tree in Task 1 in your code (use the IF, AND, OR and THEN rules which are already implemented).

```
from production import IF, AND, THEN, OR, NOT

TOURIST_RULES = (
    IF(
        AND(
            OR(
                "(?x) enjoy activities that are physically challenging",
                "(?x) involve an element of risk",
            ),
            OR(
                "(?x) likes activities such as hiking",
                "(?x) likes activities such as rock climbing",
                "(?x) likes activities such as extreme sports",
            ),
            OR(
                "(?x) tend to be young",
                "(?x) tend to be active",
            ),
            "(?x) prioritize experiences over luxury accommodations",
```

```
    ),
    THEN("(?x) areAdventureTourists"),
),
IF(
    AND(
        "(?x) interested in immersing themselves in the local culture of the places",
        OR(
            "(?x) enjoy exploring historical sites",
            "(?x) visiting museums",
            "(?x) attending cultural events",
        ),
        "(?x) tend to be curious",
        "(?x) tend to be open-minded",
        "(?x) are interested in learning about the traditions",
    ),
    THEN("(?x) areCulturalTourists"),
),
IF(
    AND(
        "(?x) are willing to spend a premium for high-end accommodations",
        "(?x) tend to have exclusive experiences",
        "(?x) seek out luxury hotels",
        "(?x) tend to have fine dining",
        OR(
            "(?x) tend to have private tours",
            "(?x) tend to have personal concierge services",
        ),
        "(?x) prioritize comfort and convenience over adventure",
    ),
    THEN("(?x) areLuxuryTourists"),
),
IF(
    AND(
        OR(
            "(?x) travel with children",
            "(?x) travel with extended family",
        ),
        "(?x) tend to seek out low-cost accommodations and activities",
        "(?x) prioritize activities that are suitable for all ages",
        "(?x) tend to prioritize safety",
        "(?x) tend to prioritize convenience",
        "(?x) tend to prioritize affordability",
        "(?x) prioritize comfort and convenience over adventure",
    ),
    THEN("(?x) areFamilyTourists"),
),
IF(
    AND(
```

```
            "(?x) prioritize affordability over luxury",
            "(?x) tend to seek out low-cost accommodations and activities",
            "(?x) travel on a shoestring budget",
            OR(
                "(?x) staying in hostels",
                "(?x) staying in budget hotels",
            ),
            NOT("(?x) seeking out expensive eats"),
        ),
        THEN("(?x) areBudgetTourists"),
    ),
)
```

Task 3 - If you are using the provided code, check how the Forward Chaining algorithm works and illustrate an example. If you are implementing your own code, implement the Forward Chaining algorithm yourself.

```
from rules import TOURIST_RULES
from production import forward_chain

print(
    forward_chain(
        TOURIST_RULES,
        {
            "Ion prioritize affordability over luxury",
            "Ion tend to seek out low-cost accommodations and activities",
            "Ion travel on a shoestring budget",
            "Ion staying in hostels",
            "Ion staying in budget hotels",
        },
        False,
        False,
    )
)

print(
    forward_chain(
        TOURIST_RULES,
        {
            "Jimmy interested in immersing themselves in the local culture of the places",
            "Jimmy enjoy exploring historical sites",
            "Jimmy tend to be curious",
            "Jimmy tend to be open-minded",
            "Jimmy are interested in learning about the traditions",
        },
        False,
```

```
        False,
    )
)

print(
    forward_chain(
        TOURIST_RULES,
        {
            "Rick interested in immersing themselves in the local culture of the places",
            "Rick enjoy exploring historical sites",
            "Rick tend to be curious",
            "Rick tend to be open-minded",
        },
        False,
        False,
    )
)
```

Output:

```
{'Ion tend to seek out low-cost accommodations and activities', 'Ion prioritize affordability over luxury', 'Ion staying in budget hotels', 'Ion staying in hostels', 'Ion travel on a shoestring budget', 'Ion areBudgetTourists'}
['Jimmy prioritize comfort and convenience over adventure', 'Jimmy are willing to spend a premium for high-end accommodations', 'Jimmy are interested in learning about the traditions', 'Jimmy tend to have personal concierge services', 'Jimmy seek out luxury hotels', 'Jimmy tend to prioritize safety', 'Jimmy enjoy exploring historical sites']


('Ion areBudgetTourists', 'Ion prioritize affordability over luxury', 'Ion staying in budget hotels', 'Ion staying in hostels', 'Ion tend to seek out low-cost accommodations and activities', 'Ion travel on a shoestring budget')
('Jimmy are interested in learning about the traditions', 'Jimmy areCulturalTourists', 'Jimmy enjoy exploring historical sites', 'Jimmy interested in immersing themselves in the local culture of the places', 'Jimmy tend to be curious', 'Jimmy tend to be open-minded')
('Rick enjoy exploring historical sites', 'Rick interested in immersing themselves in the local culture of the places', 'Rick tend to be curious', 'Rick tend to be open-minded')
```

Task 4 - Implement the Backward Chaining algorithm for the Goal Tree

```
from rules import TOURIST_RULES
from production import backward_chain, instantiate, match
from production import AND, OR

def backward_chain(rules, hypothesis, matchData = None):
    result = set()

    if matchData is None:
        for condition in rules:
```

```
            for action in condition._action:
                if match(action, hypothesis) != None:
                    result.add(instantiate(action, match(action, hypothesis)))
                    result.update(backward_chain(condition, hypothesis, match(action,
hypothesis)))
    else:
        for rule in rules._conditional:
            if isinstance(rule, AND):
                result.add(backward_chain(rule, hypothesis, matchData))
            elif isinstance(rule, str):
                result.add(instantiate(rule, matchData))
            elif isinstance(rule, OR):
                for r in rule:
                    result.add(instantiate(r, matchData))
    return result

print(backward_chain(TOURIST_RULES, "Ion areBudgetTourists"))
```

Output:

```
{'Ion tend to seek out low-cost accommodations and activities', 'Ion prioritize affordability
over luxury', 'Ion staying in budget hotels', 'Ion staying in hostels', 'Ion travel on a
shoestring budget', 'Ion areBudgetTourists'}
```

Task 5 - Implement a system for generating questions from the Goal Tree. Have at least 2 or 3 types of questions (e.g yes / no, multiple choice, etc).

```
import random
from collections import deque
from rules import TOURIST_RULES
from production import (
    IF,
    AND,
    OR,
    NOT,
    instantiate,
)


class QuestionEngine:
    __questions = []
    __questions_dequeue = []
    __tourist_name = None

    def __init__(self, goal_tree, tourist_name):
        self.__tourist_name = tourist_name
        self.__questions = self.__goal_tree_to_questions(goal_tree)
        random.shuffle(self.__questions)
```

```
        self.__questions_dequeue = deque(self.__questions)

    @property
    def questions(self):
        return self.__questions

    def get_questions(self):
        result = []
        while len(self.__questions_dequeue) > 0:
            q = None
            if len(self.__questions_dequeue) > 4 and random.randint(1, 2) == 1:
                q = self.__get_question_type2(4)
            else:
                q = self.__get_question_type1()
            if q == None:
                break
            else:
                result.extend(q)
        return result

    def __get_question_type1(self):
        question = self.__questions_dequeue.pop()
        while True:
            try:
                input_string = input(
                    f"\nIs this statement correct? {question} \nEnter (y) if is true, (n) if
is false or (x) in order to finish: "
                )

                if input_string.lower() == "x":
                    return None
                if input_string.lower() == "y":
                    return [question]
                if input_string.lower() == "n":
                    return []
                raise ValueError()
            except ValueError:
                print("Error: Input is invalid")

    def __get_question_type2(self, count_of_options):
        result = []
        options = []
        question = f"\nSelect one or more correct options which is true related to
{self.__tourist_name} or write (x) in order to finish"

        for i in range(count_of_options):
            options.append(self.__questions_dequeue.pop())
            question += f"\n\t{i+1}) {options[i]}"
```

```
        while True:
            input_string = input(
                f"{question}\nEnter a list of space-separated integers: "
            )

            if input_string.lower() == "x":
                return None

            try:
                int_list = list(set([int(x) - 1 for x in input_string.split()]))
                if len(int_list) < 1:
                    raise ValueError()
                for i in int_list:
                    if i >= len(options) or i < 0:
                        raise ValueError()
                for i in int_list:
                    result.append(options[i])
                break
            except ValueError:
                print("Error: Input is invalid")
        return result

    def __goal_tree_to_questions(self, goal_tree):
        result = []
        for node in goal_tree:
            result.extend(self.__node_tree_to_questions(node))

        for i in range(len(result)):
            result[i] = instantiate(result[i], {"x": self.__tourist_name})

        return result

    def __node_tree_to_questions(self, node):
        result = []
        if isinstance(node, IF):
            for rule in node._conditional:
                result.extend(self.__node_tree_to_questions(rule))
        if isinstance(node, AND) or isinstance(node, OR):
            for rule in node:
                result.extend(self.__node_tree_to_questions(rule))
        if isinstance(node, NOT):
            for rule in node:
                result.extend(self.__node_tree_to_questions(rule))
        if isinstance(node, str):
            result.append(node)
        return result
```

```
question_engine = QuestionEngine(TOURIST_RULES, "Jimmy")

print(question_engine.get_questions())
```

Output:

```
{'Ion tend to seek out low-cost accommodations and activities', 'Ion prioritize affordability over luxury', 'Ion staying in budget hotels', 'Ion staying in hostels', 'Ion travel on a shoestring budget', 'Ion areBudgetTourists'}
['Jimmy prioritize comfort and convenience over adventure', 'Jimmy are willing to spend a premium for high-end accommodations', 'Jimmy are interested in learning about the traditions', 'Jimmy tend to have personal concierge services', 'Jimmy seek out luxury hotels', 'Jimmy tend to prioritize safety', 'Jimmy enjoy exploring historical sites']
```

Task 6 - Wrap up everything in an interactive Expert System that will dynamically ask questions based on the input from the user. Both Forward Chaining and Backward Chaining should be working. + Task 7 Format the output and questions to human readable format.

```
import random
from collections import deque
from rules import TOURIST_RULES
from production import (
    IF,
    AND,
    OR,
    NOT,
    instantiate,
    forward_chain,
    backward_chain
)

class ExpertSystem:
    def start(self):
        tourist_name = input("Enter a tourist name: ")
        question_engine = QuestionEngine(TOURIST_RULES, tourist_name)
        chaining_type = self.__get_chaining_type()
        if chaining_type == "backward":
            facts = self.__get_facts()
            for fact in facts:
                print(backward_chain(TOURIST_RULES, fact))
        if chaining_type == "forward":
            data = set(question_engine.get_questions())
            print(forward_chain(TOURIST_RULES, data, False, False))

    def __get_chaining_type(self):
        while True:
            try:
                input_string = input(
```

```
                    f"Select forward or backward chaining:\nEnter (1) for forward chaining or
(2) for backward chaining: "
                )

                if input_string.lower() == "1":
                    return "forward"
                if input_string.lower() == "2":
                    return "backward"
                raise ValueError()
            except ValueError:
                print("Error: Input is invalid")

    def __get_facts(self):
        result = []
        while True:
            try:
                input_string = input(f"Enter fact or (x) to finish: ")

                if input_string.lower() == "x":
                    return result

                result.append(input_string)
            except ValueError:
                print("Error: Input is invalid")


e = ExpertSystem()
e.start()
```