U23AI021

Lab assignment 04
Artificial Intelligence

Harshil Andhariya
u23ai021@coed.svnit.ac.in

Q.1

```
from collections import deque

def forward_chaining(KB, query):
    inferred = set()
    agenda = deque()
    rules = []


    for statement in KB:
        if '→' in statement:
            premise, conclusion = statement.split('→')
            premise = tuple(premise.strip().split('∧'))
            rules.append((premise, conclusion.strip()))
        else:
            inferred.add(statement.strip())
            agenda.append(statement.strip())

    while agenda:
        fact = agenda.popleft()
        if fact == query:
            return True

        new_facts = []
        for premise, conclusion in rules:
            if all(p in inferred for p in premise):
                if conclusion not in inferred:
                    new_facts.append(conclusion)

        for new_fact in new_facts:
            inferred.add(new_fact)
            agenda.append(new_fact)

    return False


KB1 = ["P → Q", "L ∧ M → P", "A ∧ B → L", "A", "B", "M"]
```

```
query1 = "Q"
print("Query Q is derived:", forward_chaining(KB1, query1))

KB2 = ["A → B", "B → C", "C → D", "E", "D ∧ E → F"]
query2 = "F"
print("Query F is derived:", forward_chaining(KB2, query2))
```

```
Query Q is derived: False
Query F is derived: False
```

Q2.

```
def backward_chaining(kb, goal, inferred=None):
    if inferred is None:
        inferred = set()
    if goal in kb["facts"]:
        return True
    for premises, conclusion in kb["rules"]:
        if conclusion == goal and goal not in inferred:
            inferred.add(goal)
            if all(backward_chaining(kb, premise, inferred)
for premise in premises):
                return True
    return False

kb3 = {
    "facts": {"A", "B"},
    "rules": [
        ({"P"}, "Q"),
        ({"R"}, "Q"),
        ({"A"}, "P"),
```

```
        ({"B"}, "R"),
    ]
}
goal3 = "Q"
print("Backward Chaining Result (2a):",
backward_chaining(kb3, goal3))

kb4 = {
    "facts": {"A", "E"},
    "rules": [
        ({"A"}, "B"),
        ({"B", "C"}, "D"),
        ({"E"}, "C"),
    ]
}
goal4 = "D"
print("Backward Chaining Result (2b):",
backward_chaining(kb4, goal4))
```

```
Backward Chaining Result (2a): True
Backward Chaining Result (2b): True
```

Q3.

```
from itertools import combinations

def resolve(clause1, clause2):
    """Attempt to resolve two clauses. If resolvable, return
new clause; otherwise, return None."""
    new_clause = set()
    found = False
    for literal in clause1:
        if -literal in clause2:
            found = True
        else:
```

```
            new_clause.add(literal)
    for literal in clause2:
        if -literal not in clause1:
            new_clause.add(literal)
    return new_clause if found else None

def resolution(kb, conclusion):
    """Resolution refutation method"""
    clauses = [set(clause) for clause in kb]
    negated_goal = {-conclusion}
    clauses.append(negated_goal)

    while True:
        new_clauses = set()
        for (c1, c2) in combinations(clauses, 2):
            resolvent = resolve(c1, c2)
            if resolvent is not None:
                if not resolvent:
                    return True
                new_clauses.add(frozenset(resolvent))

        if new_clauses.issubset(set(map(frozenset,
clauses))):
            return False
        clauses.extend(list(new_clauses))

kb5 = [
    {1, 2},
    {-1, 3},
    {-2, 4},
    {-3, 4},
]
goal5 = 4
print("Resolution Result (3a):", resolution(kb5, goal5))

kb6 = [
    {-1, 2},
    {-2, 3},
```

```
    {-4, -3},
    {1},
]
goal6 = 4
print("Resolution Result (3b):", resolution(kb6, goal6))
```

```
Resolution Result (3a): True
Resolution Result (3b): False
```