Day-5 Experiments

1. Write a Prolog program to implement Monkey Banana Problem.

Program:

```
holding/1 as dynamic
:- dynamic holding/1.

% Facts
at(monkey, ground).
at(banana, ceiling).
at(chair, ground).
at(stick, ground).
holding(none).

% Actions
move(Object) :- at(monkey, ground), at(Object, ground), write('Monkey moves to '),
write(Object), nl.
pick(Object) :- move(Object), holding(none), retract(holding(none)), assert(holding(Object)),
write('Monkey picks up the '), write(Object), nl.
climb :- holding(stick), write('Monkey climbs on chair'), nl.
knock :- climb, write('Monkey knocks the bananas down'), nl.

% Goal
get_bananas :- pick(stick), climb, knock.
```

Sample output:

```
?-
|
|    get_bananas.
Monkey moves to stick
Monkey picks up the stick
Monkey climbs on chair
Monkey climbs on chair
Monkey knocks the bananas down
true.
```

2. Write a Prolog Program for fruit and its color using Back Tracking.

Program:

```
% Facts about fruits and their colors
fruit_color(apple, red).
fruit_color(banana, yellow).
fruit_color(grape, green).
fruit_color(grape, purple).
fruit_color(orange, orange).
fruit_color(cherry, red).
fruit_color(blueberry, blue).

% Rule to find fruits of a specific color
find_fruits_by_color(Color) :-
    fruit_color(Fruit, Color),
    write(Fruit), write(' is '), write(Color), nl,
    fail. % Forces backtracking to find all fruits of the given color.

% Rule to find all colors of a specific fruit
```

```
find_colors_by_fruit(Fruit) :-
    fruit_color(Fruit, Color),
    write(Fruit), write(' is '), write(Color), nl,
    fail. % Forces backtracking to find all colors of the given fruit.

% A rule to stop backtracking gracefully
find_fruits_by_color(_).
find_colors_by_fruit(_).
```

Sample output:

```
% c:/Users/Admin/Desktop/AIProject/bac
?-
|     find_fruits_by_color(red).
apple is red
cherry is red
true.

?-
```

3. Write a Prolog Program to implement Best First Search algorithm.

Program:

```
% Define edges and their weights (or distances)
edge(a, b, 1).
edge(a, c, 4).
edge(b, d, 3).
edge(c, d, 1).
edge(b, e, 5).
edge(d, e, 2).

% Define heuristic values for each node
heuristic(a, 7).
heuristic(b, 6).
heuristic(c, 2).
heuristic(d, 1).
heuristic(e, 0).

% Best First Search Algorithm
best_first_search(Start, Goal, Path, Cost) :-
    bfs_helper([[Start, 0]], Goal, [], Path, Cost).

% BFS Helper Function
bfs_helper([[Goal, Cost] | _], Goal, Visited, Path, Cost) :-
    reverse([Goal | Visited], Path).

bfs_helper([[Node, NodeCost] | Rest], Goal, Visited, Path, Cost) :-
    findall([NextNode, NewCost],
            (edge(Node, NextNode, StepCost),
             \+ member(NextNode, Visited),
             heuristic(NextNode, H),
             NewCost is NodeCost + StepCost + H),
            Neighbors),
    append(Rest, Neighbors, NewQueue),
    sort(2, @=<, NewQueue, SortedQueue),
    bfs_helper(SortedQueue, Goal, [Node | Visited], Path, Cost).
```

Sample output:

```
?-
% c:/Users/Admin/Desktop/AIProject/bfs.pl
?-
|    best_first_search(a, e, Path, Cost).
Path = [a, c, b, d, e],
Cost = 10
```

4. Write the Prolog program for Medical Diagnosis.

Program:

```
% Facts: Symptoms associated with diseases
symptom(flu, fever).
symptom(flu, headache).
symptom(flu, chills).
symptom(flu, sore_throat).

symptom(cold, sneezing).
symptom(cold, runny_nose).
symptom(cold, sore_throat).

symptom(covid19, fever).
symptom(covid19, cough).
symptom(covid19, difficulty_breathing).
symptom(covid19, loss_of_taste_or_smell).

symptom(malaria, fever).
symptom(malaria, chills).
symptom(malaria, sweating).
symptom(malaria, headache).

% Rules: Diagnosing a disease based on symptoms
diagnose(Disease) :-
    write('What symptoms are you experiencing?'), nl,
    findall(Symptom, symptom(Disease, Symptom), Symptoms),
    ask_symptoms(Symptoms, PresentSymptoms),
    length(PresentSymptoms, Count),
    Count > 0, % Ensure at least one symptom matches
    write('You might have '), write(Disease), write('.'), nl.

ask_symptoms([], []).
ask_symptoms([Symptom | Rest], [Symptom | PresentSymptoms]) :-
    write('Do you have '), write(Symptom), write('? (yes/no): '),
    read(Response),
    Response = yes,
    ask_symptoms(Rest, PresentSymptoms).
ask_symptoms([_ | Rest], PresentSymptoms) :-
    ask_symptoms(Rest, PresentSymptoms).

% Rule to display all possible diseases for a given symptom
possible_diseases(Symptom) :-
    findall(Disease, symptom(Disease, Symptom), Diseases),
    write('Possible diseases with symptom '), write(Symptom), write(': '), write(Diseases), nl.
```

Sample output:

```
What symptoms are you experiencing?
Do you have fever? (yes/no): yes
Do you have cough? (yes/no): yes
Do you have difficulty_breathing? (yes/no): no
Do you have loss_of_taste_or_smell? (yes/no): yes
You might have covid19.
```

5. Write a Prolog Program for forward Chaining. Incorporate required queries.

Program:

```
forward_chain :-
    write('Starting Forward Chaining...'), nl,
    forward_step.

forward_step :-
    find_rule_to_fire(Rule, Conditions),
    apply_rule(Rule, Conditions),
    forward_step.
forward_step :-
    write('No more rules to apply.'), nl.

find_rule_to_fire(Rule, Conditions) :-
    rule(Rule, Conditions),
    \+ fact(Rule), % Rule conclusion is not yet a known fact
    all_conditions_met(Conditions).

all_conditions_met([]).
all_conditions_met([Condition | Rest]) :-
    fact(Condition),
    all_conditions_met(Rest).

apply_rule(Rule, Conditions) :-
    assertz(fact(Rule)), % Add the derived fact
    write('Derived fact: '), write(Rule), write(' from conditions: '), write(Conditions), nl.

% Query to list all known facts
list_facts :-
    write('Known facts:'), nl,
    findall(Fact, fact(Fact), Facts),
    write(Facts), nl.
```

6. Write a Prolog Program for backward Chaining. Incorporate required queries.

Program:

```
% Facts
fact(sunny).
fact(raining).
fact(watered_garden).

% Rules
```

```
rule(wet_ground, [raining]).
rule(flowers_grow, [sunny, watered_garden]).
rule(grass_grows, [sunny, wet_ground]).

% Backward chaining implementation
backward_chain(Goal) :-
    fact(Goal), % Base case: Goal is a known fact
    write(Goal), write(' is a known fact.'), nl.

backward_chain(Goal) :-
    rule(Goal, Conditions), % Check if the goal is the result of a rule
    check_conditions(Conditions),
    write(Goal), write(' is derived using backward chaining.'), nl.

check_conditions([]). % All conditions are satisfied if the list is empty
check_conditions([Condition | Rest]) :-
    backward_chain(Condition), % Recursively verify each condition
    check_conditions(Rest).

% Query to verify if a goal is true
verify(Goal) :-
    ( backward_chain(Goal) ->
        write('The goal '), write(Goal), write(' is true.'), nl
    ;   write('The goal '), write(Goal), write(' cannot be proved.'), nl
    ).
```

Sample output:

```
% C:/Users/Admin/Desktop/AIProject/backwardchain
creep
?-  verify(grass_grows).
sunny is a known fact.
raining is a known fact.
wet_ground is derived using backward chaining.
grass_grows is derived using backward chaining.
The goal grass_grows is true.
true.
```