* Title - Assignment on State space formulation and PEAS representation for various AI applications.

* PEAS Document -

PEAS stands for Performance measure, Environment, Actuators, and Sensors, which is a model used to describe intelligent systems.

- AI Application - Smart Invigilator

1) Performance measure -

The smart invigilator's goal is to ensure the integrity of exams and enforce the rules.

2) Environment -

The environment includes the exam room, the students, and any other relevant variables.

3) Actuators -

The smart invigilator can perform the actions to achieve its goal, such as monitoring students, detecting cheating and alerting authorities.

4) Sensors -

The sensors are the inputs the smart invigilator receives from the environment, such as audio, visual, and other data sources, which it uses to make decisions and take actions.

* Conclusion - Thus we studied the implementation of PEAS representation for various AI applications.

Experiment No. 3

Program Code –

```
n = int(input("Enter the number of nodes in graph: "))
graph = {}
for i in range(n):
  key = input("Enter key for node: ")
  value = list(map(str, input("Enter values separated by space: ").split()))
  graph[key] = value
print("Graph: ", graph)
visited = []
queue = []
visited.append('0')
queue.append('0')
while queue:
    s = queue.pop(0)
    print (s, end = " ")
    for frontier in graph[s]:
        if frontier not in visited:
            visited.append(frontier)
            queue.append(frontier)
```

Output –

```
PS N:\Academics\Study Material\Degree (B.E.) in Computer Engineering\6th Sem\Artificial Intelligence (AI)\Practicals> & "C:/Program Files/Python311/python.exe" "n:/Academics/Study Material/Degree (B.E.) in Computer Engineering/6th Sem/Artificial Intelligence (AI)/Practicals/Expt3/BFS.py"
Enter the number of nodes in graph: 5
Enter key for node: 0
Enter values separated by space: 1 2
Enter key for node: 1
Enter values separated by space: 0 2 3
Enter key for node: 2
Enter values separated by space: 0 1 4
Enter key for node: 3
Enter values separated by space: 1 4
Enter key for node: 4
Enter values separated by space: 2 3
Graph:  {'0': ['1', '2'], '1': ['0', '2', '3'], '2': ['0', '1', '4'], '3': ['1', '4'], '4': ['2', '3']}
0 1 2 3 4
PS N:\Academics\Study Material\Degree (B.E.) in Computer Engineering\6th Sem\Artificial Intelligence (AI)\Practicals>
```

Experiment No. 4

Program Code –

```
n = int(input("Enter the number of vertices: "))
m = int(input("Enter the number of edges: "))
graph = [None] * n
visited = [False] * n
for i in range(n):
    graph[i] = []
for i in range(m):
    u, v = [int(x) for x in input("Enter the edge (u v): ").split()]
    graph[u].append(v)
start = int(input("Enter the starting vertex: "))
print("Depth First Traversal:")
def DFS(graph, visited, vertex):
    visited[vertex] = True
    print(vertex, end=' ')
    for i in graph[vertex]:
        if not visited[i]:
            DFS(graph, visited, i)
DFS(graph, visited, start)
```

Output –

```
PS N:\Academics\Study Material\Degree (B.E.) in Computer Engineering\6th Sem\Artificial Intelligence (AI)\Practicals> & "C:/Program Files/Python311/python.exe" "n:/Academics/Study Material/Degree (B.E.) in Computer Engineering/6th Sem/Artificial Intelligence (AI)/Practicals/Expt4/DFS.py"
Enter the number of vertices: 8
Enter the number of edges: 10
Enter the edge (u v): 0 1
Enter the edge (u v): 0 2
Enter the edge (u v): 0 3
Enter the edge (u v): 1 3
Enter the edge (u v): 2 4
Enter the edge (u v): 3 5
Enter the edge (u v): 3 6
Enter the edge (u v): 4 7
Enter the edge (u v): 4 5
Enter the edge (u v): 5 2
Enter the starting vertex: 0
Depth First Traversal:
0 1 3 5 2 4 7 6
PS N:\Academics\Study Material\Degree (B.E.) in Computer Engineering\6th Sem\Artificial Intelligence (AI)\Practicals>
```

Experiment No. 5

Program Code –

```
from queue import PriorityQueue


def greedy_best_first_search(graph, start, goal, weights):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    came_from[start] = None

    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            break

        for neighbor in graph[current]:
            if neighbor not in came_from:
                priority = weights[neighbor]
                frontier.put(neighbor, priority)
                came_from[neighbor] = current
    return came_from
graph = {}
n = int(input("Enter the number of nodes: "))

for i in range(n):
    node = input("Enter the node: ")
    neighbors = input("Enter the neighbors separated by spaces: ")
    graph[node] = neighbors.split()

weights = {}
for node in graph:
```

```
    weight = int(input("Enter the weight of node {}: ".format(node)))
    weights[node] = weight

start = input("Enter the start node: ")
goal = input("Enter the goal node: ")

came_from = greedy_best_first_search(graph, start, goal, weights)

path = []
current = goal
while current != start:
    path.append(current)
    current = came_from[current]
path.append(start)
path.reverse()

print("Shortest path:", path)
```

Output –

```
Enter the number of nodes: 6
Enter the node: P
Enter the neighbors separated by spaces: Q R S
Enter the node: Q
Enter the neighbors separated by spaces: P X
Enter the node: R
Enter the neighbors separated by spaces: P X Y
Enter the node: S
Enter the neighbors separated by spaces: P
Enter the node: X
Enter the neighbors separated by spaces: Q R Y
Enter the node: Y
Enter the neighbors separated by spaces: R X
Enter the weight of node P: 6
Enter the weight of node Q: 20
Enter the weight of node R: 4
Enter the weight of node S: 11
Enter the weight of node X: 9
Enter the weight of node Y: 3
Enter the start node: P
Enter the goal node: Y
Shortest path: ['P', 'R', 'Y']
```

Experiment No. 6

Program Code –

```
import random


print("***** 4-Queen Problem using Hill Climbing Algorithm *****")
def get_board_cost(board):
    cost = 0
    for i in range(len(board)):
        for j in range(i+1, len(board)):
            if board[i] == board[j] or abs(board[i]-board[j]) == abs(i-j):
                cost += 1
    return cost


def hill_climb():
    board = [random.randint(0, 3) for _ in range(4)]
    print("Board: ", board)
    cost = get_board_cost(board)
    while cost > 0:
        new_board = list(board)
        for i in range(len(board)):
            for j in range(4):
                if j != board[i]:
                    new_board[i] = j
                    new_cost = get_board_cost(new_board)
                    if new_cost < cost:
                        board = list(new_board)
                        cost = new_cost
                        break
        if board == new_board:
            break
    return board
print("Solution: ", hill_climb())
```

Output –

```
***** 4-Queen Problem using Hill Climbing Algorithm *****
Board:  [0, 1, 3, 2]
Solution:  [1, 3, 0, 2]
```

* Aim- Write case study on first order logic.

* Theory -

- First-Order logic -
  - First-order logic is a way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
  - FOL is sufficiently expressive to represent the natural language statements in concise way.
  - First-order logic is also known as Predicate logic or First-order predicate logic.

- Basic Elements of First-order logic -
  1) Constant - 1, 2, A, John, Mumbai, cat....
  2) Variables - x, y, z, a, b, ...
  3) Predicates - Brother, Father, >....
  4) Function - sqrt, LeftLegOf, ...
  5) Connectives - $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$
  6) Equality - ==
  7) Quantifier - $\forall$, $\exists$

- Atomic sentences -
  - Atomic sentences are the most basic sentences of first-order logic
  - These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
  - Example - Chinky is a cat : $\Rightarrow$ cat(Chinky)

- Complex Sentences -
  - Complex sentences are made by combining atomic sentences using connectives.

- Quantifiers in First-order logic -
  - A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
  - There are two types of quantifier -
    1) Universal Quantifier (for all) -
       It is a symbol of logical representation which specifies that the statement within its range is true for everything or every instance of a particular thing.
       Represented by ∀.
       e.g. All man drink coffee.
       ∀x man(x) → drink (x, coffee)

    2) Existential Quantifier -
       It is a type of quantifier which express that the statement within its scope is true for at least one instance of something.
       Represented by ∃.
       e.g. Some boys are intelligent -
       ∃x : boys(x) ∧ intelligent

- Knowledge engineering -

The process of constructing knowledge-base in first-order logic is called as knowledge-engineering.

- Inference in FOL -

Inference in first-order logic is used to deduce new facts or sentences from existing sentences.

* First-order Logic inference rules for quantifier -

1) Universal generalization -

- It states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as $\forall x P(x)$
- It can be represented as $\frac{P(c)}{\forall x P(x)}$

2) Universal Instantiation -

- It is also called as universal elimination.
- It can be applied multiple times to add new sentences.
- As per UI, we can infer any sentence obtained by substituting a ground term for the variable.
- It can be represented as $\frac{\forall x P(x)}{P(c)}$

3) Existential Instantiation -

- It is also called as Existential elimination.
- It can be applied only once to replace the existential sentence.
- It can be represented as $\frac{\exists x P(x)}{P(c)}$

4) Existential Introduction-

- It is also known as existential generalization, which is a valid inference rule in first-order logic.
- It states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as $\frac{P(c)}{\exists x P(x)}$

* Generalized Modus Ponens Rule:

- Generalized Modus Ponens can be summarized as, "P implies Q and P is asserted to be true, therefore Q must be True."
- According to Modus Ponens, for atomic sentences p_i, p_i', q. Where there is a substitution θ such that $SUBST(\theta, p_i') = SUBST(\theta, p_i)$, it can be represented as:

$$\frac{p_1', p_2', \ldots p_n', (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{SUBST(\theta, q)}$$

- Unification-

- Unification is a process of making two different logical atomic expressions identical by finding a substitution.
- It takes two literals as input and makes them identical using substitution.

* Conditions for unification -

1) Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
2) Number of arguments in both expressions must be identical.
3) Unification will fail, if there are two similar variables present in the same expression.

- Forward chaining in AI -
  - It is also known as forward reasoning method when using an inference engine.
  - Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules in the forward direction to extract more data until a goal is reached.
  - Forward chaining algorithm starts with known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

- Backward chaining in AI -
  - It is also known as backward reasoning method when using an inference engine.
  - A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

- Resolution in FOL-
  - Resolution is a theorem proving technique that proceeds by building refutation proofs.
  - Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements.
  - Unification is a key concept in proofs by resolutions.
  - Resolution is a single inference rule which can efficiently operate on the conjunctive normal form or clausal form.

[Example]

- Sentences -

1) If a student is good in sports, then they must belong to the B4 batch.
2) All TEComp students who are absent during sports belong to the B2 batch.
3) Vikas loves workouts and is a TEComp student.
4) All TEComp students are good in studies.
5) If a student loves playing cricket, then they must love workouts.
6) If a student is an athelete, then they must love playing cricket and workouts.
7) If a student loves playing cricket, then they are ~~unlikely to be~~ good in sports.
8) If a student is absent during sports, then they are unlikely to be good in sports.
9) If a student is good in studies, then they are likelly to be ~~good in studies~~. a TEComp student.

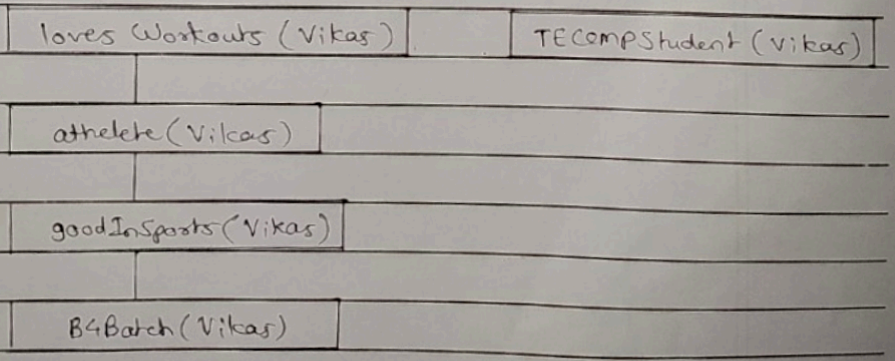
10) If a student is a TEComp student, then they are likely to be good in studies.

- First-order logic for each statement -

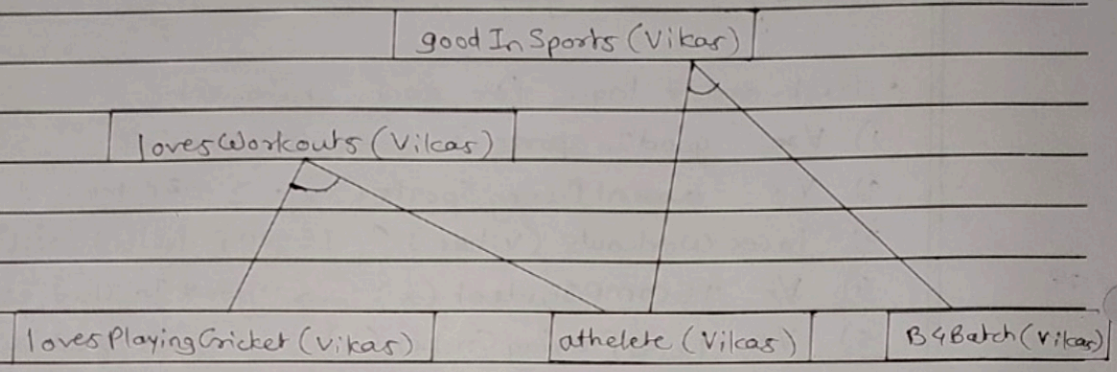
1) $\forall x$ goodInSports(x) $\rightarrow$ B4Batch(x)
2) $\forall x$ absentDuringSports(x) $\rightarrow$ B2Batch(x)
3) lovesWorkouts(Vikas) $\wedge$ TECompStudent(Vikas)
4) $\forall x$ TECompStudent(x) $\rightarrow$ goodInStudies(x)
5) $\forall x$ lovesPlayingCricket(x) $\rightarrow$ lovesWorkout(x)
6) $\forall x$ athelete(x) $\rightarrow$ lovesPlayingCricket(x) $\wedge$ lovesWorkouts(x)
7) $\forall x$ goodInSports(x) $\rightarrow$ lovesPlayingCricket(x)
8) $\forall x$ absentDuringSports(x) $\rightarrow \neg$ goodInsports(x)
9) friends(Ajay, Vikas) $\wedge$ lostCricketMatch(Vikas, Ajay)
10) goodBatsman(Ajay)

Prove that Vikas is from B4Batch

* Forward Chaining -



* Backward Chaining -



* Conclusion -

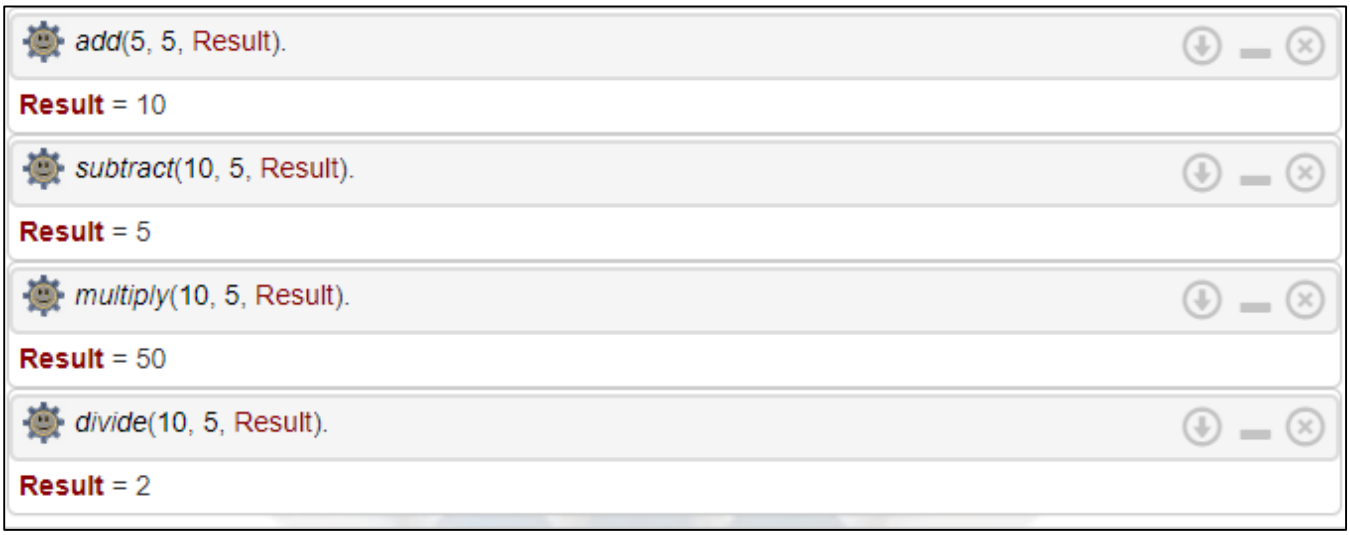
Thus, we studied the First-order logic in AI.

Experiment No. 8

1) Program to perform Arithmetic Operations –

```
add(X, Y, Z) :- Z is X + Y.

subtract(X, Y, Z) :- Z is X - Y.

multiply(X, Y, Z) :- Z is X * Y.

divide(X, Y, Z) :- Z is X / Y.
```

Output –



2) Program to check the number is even, odd or prime –

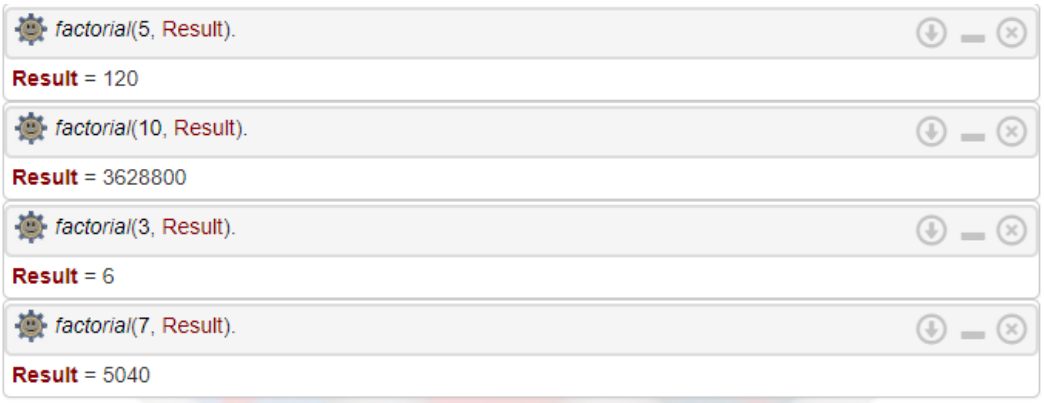
```
is_even(X) :-
    X mod 2 =:= 0.
is_odd(X) :-
    X mod 2 =:= 1.


is_prime(X) :-
    X > 1,
    Upper is floor(sqrt(X)),
    \+ (between(2, Upper, Y), X mod Y =:= 0).
```

Output –

3) Program to calculate factorial of a number –

```
factorial(0, 1).
factorial(N, Result) :-
    N > 0,
    Prev is N - 1,
    factorial(Prev, PrevResult),
    Result is N * PrevResult.
```

Output –



4) Program to find the greatest among three numbers –

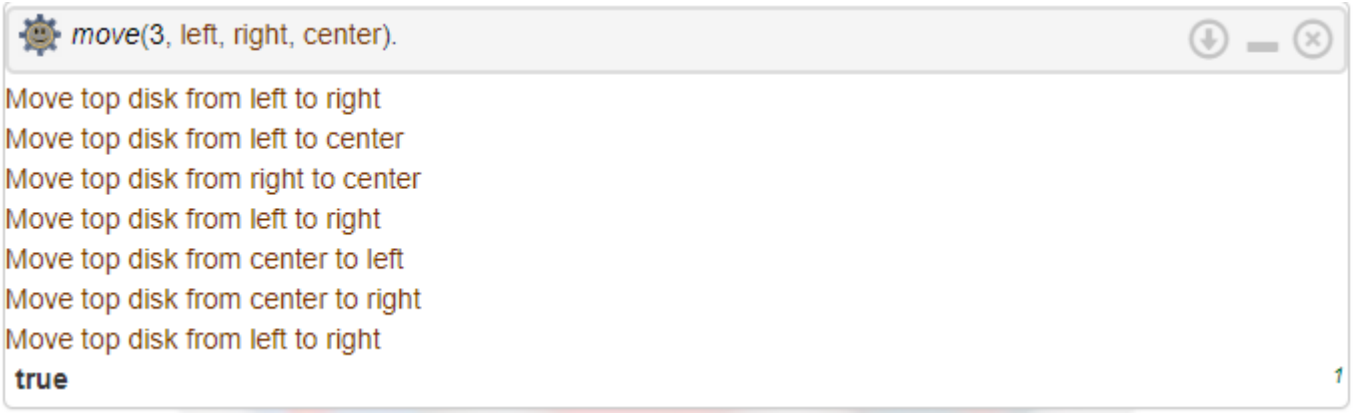
```
greatest(X, Y, Z, G) :-
    X >= Y, X >= Z,
    G is X.
greatest(X, Y, Z, G) :-
    Y >= X, Y >= Z,
    G is Y.
greatest(_, _, Z, G) :-
```

Output –

5) Program to solve the problem of Tower Of Honoi (TOH) –

```
move(1,X,Y,_) :-
   write('Move top disk from '), write(X), write(' to '), write(Y), nl.
move(N,X,Y,Z) :-
   N>1,
   M is N-1,
   move(M,X,Z,Y),
   move(1,X,Y,_),
   move(M,Z,Y,X).
```

Output –


move(3, left, right, center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right
true
1

Experiment No. 9

Program Code –

```
% Facts
father(nitin, niraj).
father(nitin, isha).
father(shrikant, nitin).
father(shrikant, suhas).
father(sitaram, shrikant).
father(suhas, sujal).
father(suhas, sagarika).
mother(nidhi, niraj).
mother(nidhi, isha).
brother(niraj, isha).
brother(suhas, nitin).
sister(isha, niraj).
wife(nidhi, nitin).
wife(suvidha, suhas).

% Rules
parent(X, Y) :- father(X, Y).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
greatgrandparent(X, Y) :- grandparent(X, Z), parent(Z, Y).
siblings(X, Y) :- brother(X, Y), sister(Y, X).
```

Output –


siblings(niraj, isha).
true
1
greatgrandparent(sitaram, niraj).
true
1
greatgrandparent(sitaram, sagarika).
true
1
grandparent(sitaram, suhas).
true
1
grandparent(shrikant, sujal).
true
1
wife(nidhi, nitin).
true
1