# FAI Experiments Code

- **Program – Propositional Logic**

```
def evaluate_expression(p, q, expression):
    if expression == "p and q":
        return p and q
    elif expression == "p or q":
        return p or q
    elif expression == "not p":
        return not p
    elif expression == "not q":
        return not q
    else:
        return None

def truth_table(expression):
    print("p | q | Result")
    print("--|---|-------")
    for p in [True, False]:
        for q in [True, False]:
            result = evaluate_expression(p, q, expression)
            print(f"{int(p)} | {int(q)} | {int(result)}")

expression = "p and q"
truth_table(expression)
```

**Output –**

```
p | q | Result
| | 1 | 1 | 1
1 | 0 | 0
0 | 1 | 0
0 | 0 | 0
```

**Program – Predicate Logic**

```
def is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(x ** 0.5) + 1):
        if x % i == 0:
            return False
    return True
```

```
def universal_quantification(predicate, domain):
    return all(predicate(x) for x in domain)

def existential_quantification(predicate, domain):
    return any(predicate(x) for x in domain)

domain = range(1, 21)

print("All numbers are prime:", universal_quantification(is_prime, domain))
print("There exists a prime number:", existential_quantification(is_prime, domain))
```

**Output –**

```
All numbers are prime: False
There exists a prime number: True
```

- **Best First Search (BFS)**

```
from queue import PriorityQueue

v = 14
graph = [[] for i in range(v)]

def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True
    while pq.empty() == False:
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
            break
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
```

```
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

**Output –**

0 1 3 2 8 9

- **Depth First Search (DFS)**

**Program –**

```
graph = {
'5': ['3', '7'],
'3': ['2', '4'],
'7': ['8'],
'2': [],
'4': ['8'],
'8': []
}
visited = set()
def dfs(visited, graph, node):
if node not in visited:
print(node)
visited.add(node)
for neighbour in graph[node]:
dfs(visited, graph, neighbour)
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

**Output –**

Following is the Depth-First Search
5
3
2
4
8
7

- **Breadth First Search (BFS)**

**Program –**

```
graph = {
'5': ['3', '7'],
'3': ['2', '4'],
'7': ['8'],
'2': [],
'4': ['8'],
'8': []
}
visited = []
queue = []
def bfs(visited, graph, node):
visited.append(node)
queue.append(node)
while queue:
m = queue.pop(0)
print(m, end=" ")
for neighbour in graph[m]:
if neighbour not in visited:
visited.append(neighbour)
queue.append(neighbour)
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

**Output –**

Following is the Breadth-First Search
5 3 7 2 4 8

- **Tic Tac Toe game using heuristics**

**Program – Without heuristic function**

```
class TicTacToe:
def __init__(self):
self.board = [' '] * 9
self.current_player = 'X'
self.winning_combinations = [
[0, 1, 2], [3, 4, 5], [6, 7, 8], # Rows
[0, 3, 6], [1, 4, 7], [2, 5, 8], # Columns
[0, 4, 8], [2, 4, 6] # Diagonals
]
def print_board(self):
print(" 0 | 1 | 2 ")
print(" -----------")
print(" 3 | 4 | 5 ")
print(" -----------")
print(" 6 | 7 | 8 ")
print("Current Board:")
for i in range(0, 9, 3):
print(f" {self.board[i]} | {self.board[i+1]} | {self.board[i+2]} ")
if i < 6:
print("-----------")
def is_winner(self, player):
for combo in self.winning_combinations:
if all(self.board[i] == player for i in combo):
return True
return False
def is_board_full(self):
return ' ' not in self.board
def is_valid_move(self, move):
return 0 <= move < 9 and self.board[move] == ' '
def make_move(self, move, player):
```

```
self.board[move] = player
def switch_player(self):
self.current_player = 'O' if self.current_player == 'X' else 'X'
def play(self):
while not self.is_game_over():
self.print_board()
move = int(input(f"{self.current_player}'s turn. Enter your move (0-8): "))
if self.is_valid_move(move):
self.make_move(move, self.current_player)
if self.is_winner(self.current_player):
self.print_board()
print(f"{self.current_player} wins!")
break
elif self.is_board_full():
self.print_board()
print("It's a draw!")
break
else:
self.switch_player()
else:
print("Invalid move. Try again.")
def is_game_over(self):
return self.is_winner('X') or self.is_winner('O') or self.is_board_full()
if __name__ == "__main__":
game = TicTacToe()
game.play()
```

**Output –**

```
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
| |
-----------
| |
-----------
| |
```

```
X's turn. Enter your move (0-8): 1
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
 | X | 
-----------
 |  | 
-----------
 |  | 
O's turn. Enter your move (0-8): 2
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
 | X | O
-----------
 |  | 
-----------
 |  | 
X's turn. Enter your move (0-8): 3
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
 | X | O
-----------
X |  | 
-----------
 |  | 
```

```
O's turn. Enter your move (0-8): 5
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
| X | O
-----------
X | | O
-----------
| |
X's turn. Enter your move (0-8): 6
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
| X | O
-----------
X | | O
-----------
X | |
O's turn. Enter your move (0-8): 8
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
| X | O
-----------
X | | O
-----------
X | | O
O wins!
```

- **Program – With heuristic function**

```
class TicTacToe:
def __init__(self):
self.board = [' '] * 9
self.current_player = 'X'
self.winning_combinations = [
[0, 1, 2], [3, 4, 5], [6, 7, 8], # Rows
[0, 3, 6], [1, 4, 7], [2, 5, 8], # Columns
[0, 4, 8], [2, 4, 6] # Diagonals
]
def print_board(self):
print(" 0 | 1 | 2 ")
print(" -----------")
print(" 3 | 4 | 5 ")
print(" -----------")
print(" 6 | 7 | 8 ")
print("Current Board:")
for i in range(0, 9, 3):
print(f" {self.board[i]} | {self.board[i+1]} | {self.board[i+2]} ")
if i < 6:
print("-----------")
def is_winner(self, player):
for combo in self.winning_combinations:
if all(self.board[i] == player for i in combo):
return True
return False
def is_board_full(self):
return ' ' not in self.board
```

```
def is_game_over(self):
return self.is_winner('X') or self.is_winner('O') or self.is_board_full()
def make_move(self, move, player):
self.board[move] = player
def evaluate_board(self):
if self.is_winner('X'):
return -1 # 'X' wins
elif self.is_winner('O'):
return 1 # 'O' wins
else:
return 0 # Draw
def determine_best_move(self):
for i in range(9):
if self.board[i] == ' ':
return i
def play(self):
while not self.is_game_over():
self.print_board()
if self.current_player == 'X':
move = int(input("Enter your move (0-8): "))
else:
move = self.determine_best_move()
if move in range(9) and self.board[move] == ' ':
self.make_move(move, self.current_player)
if self.is_winner(self.current_player):
self.print_board()
print(f"{self.current_player} wins!")
break
elif self.is_board_full():
```

```
self.print_board()
print("It's a draw!")
break
else:
self.current_player = 'O' if self.current_player == 'X' else 'X'
else:
print("Invalid move. Try again.")
if __name__ == "__main__":
game = TicTacToe()
game.play()
```

**Output –**

```
0 | 1 | 2
-----------
3 | 4 | 5
-----------
6 | 7 | 8
Current Board:
| |
-----------
| |
-----------
| |
Enter your move (0-8): 0
X | |
-----------
| |
```

-----------
| |
Current Board:
X | |
-----------
| |
-----------
O | |
Enter your move (0-8): 1
X | X |
-----------
| |
-----------
O | |
Current Board:
X | X | O
-----------
| |
-----------
O | |
Enter your move (0-8): 4
X | X | O
-----------
| X |
-----------

O | |
Current Board:
X | X | O
-----------
O | X |
-----------
O | |
Enter your move (0-8): 8
X | X | O
-----------
O | X |
-----------
O | | X
X wins!

- **A simple Expert system for a traffic light controller (using class and knowledge base method)**

```
class TrafficLightController:
def __init__(self):
self.knowledge_base = {
'weather': None,
'time_of_day': None,
'traffic_density': None,
'previous_state': None
}
def update_knowledge_base(self, weather, time_of_day, traffic_density):
self.knowledge_base['weather'] = weather
self.knowledge_base['time_of_day'] = time_of_day
self.knowledge_base['traffic_density'] = traffic_density
def decide_traffic_light_state(self):
# Rules for deciding traffic light state based on the knowledge base
if self.knowledge_base['weather'] == 'clear' and self.knowledge_base['time_of_day'] ==
'day':
if self.knowledge_base['traffic_density'] == 'low':
return 'green'
elif self.knowledge_base['traffic_density'] == 'medium':
return 'yellow'
```

```
elif self.knowledge_base['traffic_density'] == 'high':
return 'red'
elif self.knowledge_base['weather'] == 'rainy' or self.knowledge_base['weather'] == 'foggy':
return 'yellow'
elif self.knowledge_base['time_of_day'] == 'night':
return 'red'
else:
return 'green'
# Example usage:
controller = TrafficLightController()
# Update knowledge base with current conditions
controller.update_knowledge_base('clear', 'day', 'low')
# Decide traffic light state
state = controller.decide_traffic_light_state()
print("Traffic Light State:", state)
```

**Output –**

Traffic Light State: green

- **Program – Expert System for diagnosing a medical condition based on symptoms provided by the user**

```
class MedicalDiagnosisSystem:
def __init__(self):
self.knowledge_base = {
'fever': None,
'cough': None,
'fatigue': None,
'headache': None
```

```
}
def update_symptoms(self, fever, cough, fatigue, headache):
self.knowledge_base['fever'] = fever
self.knowledge_base['cough'] = cough
self.knowledge_base['fatigue'] = fatigue
self.knowledge_base['headache'] = headache
def diagnose_condition(self):
# Check for symptoms and diagnose the condition
if self.knowledge_base['fever'] and self.knowledge_base['cough'] and
self.knowledge_base['headache']:
return "You may have the flu."
elif self.knowledge_base['fever'] and self.knowledge_base['fatigue']:
return "You may have a viral infection."
elif self.knowledge_base['cough'] and self.knowledge_base['fatigue']:
return "You may have a respiratory infection."
elif self.knowledge_base['headache']:
return "You may have a migraine."
else:
return "Unable to diagnose the condition based on provided symptoms."
# Example usage:
diagnosis_system = MedicalDiagnosisSystem()
# Get symptoms from the user
fever = input("Do you have a fever? (yes/no): ").lower() == 'yes'
cough = input("Do you have a cough? (yes/no): ").lower() == 'yes'
fatigue = input("Do you experience fatigue? (yes/no): ").lower() == 'yes'
headache = input("Do you have a headache? (yes/no): ").lower() == 'yes'
# Update symptoms in the system
diagnosis_system.update_symptoms(fever, cough, fatigue, headache)
# Diagnose the condition
```

```
diagnosis = diagnosis_system.diagnose_condition()
print("Diagnosis:", diagnosis)
```

**Output –**

```
Do you have a fever? (yes/no): yes
Do you have a cough? (yes/no): yes
Do you experience fatigue? (yes/no): no
Do you have a headache? (yes/no): yes
Diagnosis: You may have the flu.
```

- **Develop a simple expert system - for Car Troubleshooting.**

```
class ExpertSystem:
  def __init__(self):
    self.rules = {
      "dead_battery": ["car_won't_start", "lights_dim"],
      "bad_fuel_pump": ["car_won't_start", "no_fuel_pressure"],
      "faulty_starter_motor": ["car_won't_start", "clicking_sound"],
      "bad_alternator": ["car_won't_start", "dead_battery", "alternator_not_charging"],
      "clogged_air_filter": ["car_running_rough", "poor_gas_mileage"],
      "faulty_oxygen_sensor": ["car_running_rough", "check_engine_light_on"],
      "bad_spark_plugs": ["car_running_rough", "engine_misfiring"],
      "low_transmission_fluid": ["car_slipping_gears", "transmission_leaking_fluid"],
      "bad_transmission_solenoids": ["car_slipping_gears", "transmission_not_shifting"],
      "bad_engine_mounts": ["car_vibrating", "engine_noise"],
      "worn_out_brake_pads": ["car_squealing", "brake_pedal_spongy"]
    }

  def diagnose(self, symptoms):
    possible_problems = []
    for problem, symptoms_list in self.rules.items():
      if all(symptom in symptoms for symptom in symptoms_list):
        possible_problems.append(problem)
    return possible_problems

  def get_subsets(self, lst):
    return [lst[i:j] for i in range(len(lst)) for j in range(i + 1, len(lst) + 1)]

def main():
  expert_system = ExpertSystem()
  print("Car Troubleshooting Expert System")
  print("----------------------------------")
  symptoms = input("Enter symptoms (comma separated): ").split(',')
  symptoms = [symptom.strip() for symptom in symptoms]
```

```
    possible_problems = expert_system.diagnose(symptoms)
    if possible_problems:
        print("Possible problems:")
        for problem in possible_problems:
            print("-", problem)
    else:
        print("No possible problems found.")

if __name__ == "__main__":
    main()
```

**Output –**

```
Car Troubleshooting Expert System
----------------------------------
Enter symptoms (comma separated): car_won't_start, lights_dim
Possible problems:
- dead_battery
```

- **Develop a simple expert system - for Financial Investment Advisor**

```
class ExpertSystem:
    def __init__(self):
        self.rules = {
            "conservative_investor": ["low_risk_tolerance", "long_term_investment_horizon"],
            "moderate_investor": ["medium_risk_tolerance",
"medium_term_investment_horizon"],
            "aggressive_investor": ["high_risk_tolerance", "long_term_investment_horizon"],
            "balanced_investor": ["medium_risk_tolerance", "long_term_investment_horizon"],
            "income_investor": ["low_risk_tolerance", "short_term_investment_horizon"],
            "growth_investor": ["medium_risk_tolerance", "short_term_investment_horizon"],
            "speculative_investor": ["high_risk_tolerance", "short_term_investment_horizon"]
        }

    def recommend_investment(self, risk_tolerance, investment_horizon):
        possible_investments = []
        for investment, requirements in self.rules.items():
            if (risk_tolerance in requirements) and (investment_horizon in requirements):
                possible_investments.append(investment)
        return possible_investments if possible_investments else ["balanced_investor"]

def main():
    expert_system = ExpertSystem()
    print("Financial Investment Advisor Expert System")
    print("-----------------------------------------")
```

```
    while True:
        risk_tolerance = input("Enter your risk tolerance (low, medium, high): ").lower()
        investment_horizon = input("Enter your investment horizon (short-term, medium-term, long-term): ").lower()

        possible_investments = expert_system.recommend_investment(risk_tolerance, investment_horizon)
        print("Suitable investment plan:")
        print("-" * 30)
        for investment in possible_investments:
            print("-", investment)
        print("-" * 30)
        break

if __name__ == "__main__":
    main()
```

**Output –**

```
Financial Investment Advisor Expert System
----------------------------------------
Enter your risk tolerance (low, medium, high): medium
Enter your investment horizon (short-term, medium-term, long-term): medium-term
Suitable investment plan:
------------------------------
- balanced_investor
------------------------------
```

- **Implementation of computational graph using TensorFlow core.**

```
import tensorflow as tf

# Step 1: Define the graph
# Create TensorFlow constants
a = tf.constant(5.0, name='a')
b = tf.constant(3.0, name='b')

# Perform operations
add = tf.add(a, b, name='add')
sub = tf.subtract(a, b, name='sub')
mul = tf.multiply(a, b, name='mul')
div = tf.divide(a, b, name='div')

# Step 2: Execute the graph
# Using TensorFlow 2.x eager execution, operations are executed immediately
```

```
print("Addition: ", add.numpy())
print("Subtraction: ", sub.numpy())
print("Multiplication: ", mul.numpy())
print("Division: ", div.numpy())

# Step 3: Building a more complex graph with placeholders
# Placeholders are replaced with tf.function in TensorFlow 2.x
@tf.function
def compute(x, y):
    add = tf.add(x, y)
    sub = tf.subtract(x, y)
    mul = tf.multiply(x, y)
    div = tf.divide(x, y)
    return add, sub, mul, div

# Define inputs
x = tf.constant(10.0)
y = tf.constant(2.0)

# Execute the function
results = compute(x, y)
print("Results with tf.function:")
print("Addition: ", results[0].numpy())
print("Subtraction: ", results[1].numpy())
print("Multiplication: ", results[2].numpy())
print("Division: ", results[3].numpy())
```

**Output –**

```
Addition:  8.0
Subtraction:  2.0
Multiplication:  15.0
Division:  1.6666666666666667
Results with tf.function:
Addition:  12.0
Subtraction:  8.0
Multiplication:  20.0
Division:  5.0
```