MINI PROJECT - 1

chakradhar reddi vitta

934595987

```
import numpy as np
import random
import matplotlib.pyplot as plt
```

PART - A

1)

```
import numpy as np

# Initialize state space
states = []
for x in range(4):
    for y in range(4):
        is_fire = 1 if (x, y) in [(0, 1), (0, 2)] else 0
        is_water = 1 if (x, y) in [(2, 1), (2, 2)] else 0
        states.append((x, y, is_water, is_fire))

# Define possible actions and transitions
actions = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1), 'R': (0, 1)}
transition_slips = {'U': ['L', 'R'], 'D': ['L', 'R'], 'L': ['U', 'D'], 'R': ['U', 'D']}

# Define rewards
rewards = {}
for state in states:
    x, y, is_water, is_fire = state
    if (x, y) == (3, 3):
        rewards[state] = 100
    elif is_fire:
        rewards[state] = -10
    elif is_water:
        rewards[state] = -5
    else:
        rewards[state] = -1

    print(rewards)
```

```
{(0, 0, 0, 0): -1}
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10}
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10}
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1}
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1}
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1}
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1}
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
{(0, 0, 0, 0): -1, (0, 1, 0, 1): -10, (0, 2, 0, 1): -10, (0, 3, 0, 0): -1, (1, 0, 0, 0): -1, (1, 1, 0, 0): -1, (1, 2, 0, 0): -1, (1, 3,
```

2)

```
# Function to compute next position
def get_next_position(x, y, action):
    dx, dy = actions[action]
    new_x, new_y = x + dx, y + dy
    return (new_x, new_y) if 0 <= new_x < 4 and 0 <= new_y < 4 else (x, y)

# Function to determine water/fire presence
def check_features(x, y):
    return (1 if (x, y) in [(2, 1), (2, 2)] else 0, 1 if (x, y) in [(0, 1), (0, 2)] else 0)
```

```
# Transition model
def transition_model(state, action):
    if state[:2] == (3, 3):
        return {state: 1.0}

    x, y, water, fire = state
    main_x, main_y = get_next_position(x, y, action)
    left_x, left_y = get_next_position(x, y, transition_slips[action][0])
    right_x, right_y = get_next_position(x, y, transition_slips[action][1])

    return {
        (main_x, main_y, *check_features(main_x, main_y)): 0.8,
        (left_x, left_y, *check_features(left_x, left_y)): 0.1,
        (right_x, right_y, *check_features(right_x, right_y)): 0.1
    }



# Value Iteration
def value_iteration(discount_factor, tolerance=1e-6):
    values = {s: 0 for s in states}
    policy = {s: None for s in states}

    while True:
        max_diff = 0
        for s in states:
            if s[:2] == (3, 3):
                values[s] = rewards[s]
                policy[s] = None
                continue

            action_values = {}
            for a in actions:
                action_values[a] = rewards[s] + sum(prob * discount_factor * values[next_s]
                                                    for next_s, prob in transition_model(s, a).items())
            best_action = max(action_values, key=action_values.get)
            max_diff = max(max_diff, abs(values[s] - action_values[best_action]))
            values[s], policy[s] = action_values[best_action], best_action

        if max_diff < tolerance:
            break

    return policy, values

policy, values = value_iteration(discount_factor=0.95)
policy, values = value_iteration(discount_factor=0.3)

print("Policy for gamma = 0.3:")
for s in states:
    print(f"State {s[:2]}: {policy[s]}")

print("\nObjective values for gamma = 0.3:")
for state, value in values.items():
    print(f"State {state}: {value:.6f}")

policy, values = value_iteration(discount_factor=0.95)
print("\nPolicy for gamma = 0.95:")
for s in states:
    print(f"State {s[:2]}: {policy[s]}")

print("\nObjective values for gamma = 0.95:")
for state, value in values.items():
    print(f"State {state}: {value:.6f}")
```

```
State (3, 0): R
State (3, 1): R
```

```
State (2, 1, 1, 0): -3.856861
State (2, 2, 1, 0): 1.410669
State (2, 3, 0, 0): 23.754969
State (3, 0, 0, 0): 0.103937
State (3, 1, 0, 0): 4.727306
State (3, 2, 0, 0): 23.754969
State (3, 3, 0, 0): 100.000000

Policy for gamma = 0.95:
State (0, 0): D
State (0, 1): D
State (0, 2): R
State (0, 3): D
State (1, 0): D
State (1, 1): R
State (1, 2): R
State (1, 3): D
State (2, 0): D
State (2, 1): D
State (2, 2): R
State (2, 3): D
State (3, 0): R
State (3, 1): R
State (3, 2): R
State (3, 3): None

Objective values for gamma = 0.95:
State (0, 0, 0, 0): 59.313860
State (0, 1, 0, 1): 53.486503
State (0, 2, 0, 1): 60.191669
State (0, 3, 0, 0): 75.355755
State (1, 0, 0, 0): 65.260297
State (1, 1, 0, 0): 68.596681
State (1, 2, 0, 0): 75.821960
State (1, 3, 0, 0): 83.524671
State (2, 0, 0, 0): 70.452480
State (2, 1, 1, 0): 72.534464
State (2, 2, 1, 0): 80.263170
State (2, 3, 0, 0): 91.298344
State (3, 0, 0, 0): 76.143053
State (3, 1, 0, 0): 83.179575
State (3, 2, 0, 0): 91.298344
State (3, 3, 0, 0): 100.000000
```

3)

```
# Policy Iteration
def policy_iteration(discount_factor, tolerance=1e-6):
    policy = {s: np.random.choice(list(actions.keys())) for s in states}
    values = {s: 0 for s in states}

    while True:
        # Policy Evaluation
        while True:
            max_diff = 0
            new_values = values.copy()
            for s in states:
                if s[:2] == (3, 3):
                    new_values[s] = rewards[s]
                    policy[s] = None
                    continue
                a = policy[s]
                new_values[s] = rewards[s] + sum(prob * discount_factor * values[next_s]
                                                 for next_s, prob in transition_model(s, a).items())
                max_diff = max(max_diff, abs(values[s] - new_values[s]))
            values = new_values
            if max_diff < tolerance:
                break

        # Policy Improvement
        stable = True
        for s in states:
            if s[:2] == (3, 3):
                continue
            old_action = policy[s]
            action_values = {a: rewards[s] + sum(prob * discount_factor * values[next_s]
                                                 for next_s, prob in transition_model(s, a).items()) for a in actions}
            best_action = max(action_values, key=action_values.get)
            policy[s] = best_action
            if old_action != best_action:
```

```
                stable = False
        if stable:
            break

    return policy, values

# Execute Iterations
policy_gamma_03, values_gamma_03 = value_iteration(0.3)
policy_gamma_095, values_gamma_095 = value_iteration(0.95)
policy_pi, values_pi = policy_iteration(0.95)


print("\nPolicy for gamma = 0.95:")
for s in states:
    print(f"State {s[:2]}: {policy_gamma_095[s]}")

print("\nObjective Values:")
for state, value in values.items():
    print(f"State {state}: {value:.6f}")
```

```
Policy for gamma = 0.95:
State (0, 0): D
State (0, 1): D
State (0, 2): R
State (0, 3): D
State (1, 0): D
State (1, 1): R
State (1, 2): R
State (1, 3): D
State (2, 0): D
State (2, 1): D
State (2, 2): R
State (2, 3): D
State (3, 0): R
State (3, 1): R
State (3, 2): R
State (3, 3): None

Objective Values:
State (0, 0, 0, 0): 59.313860
State (0, 1, 0, 1): 53.486503
State (0, 2, 0, 1): 60.191669
State (0, 3, 0, 0): 75.355755
State (1, 0, 0, 0): 65.260297
State (1, 1, 0, 0): 68.596681
State (1, 2, 0, 0): 75.821960
State (1, 3, 0, 0): 83.524671
State (2, 0, 0, 0): 70.452480
State (2, 1, 1, 0): 72.534464
State (2, 2, 1, 0): 80.263170
State (2, 3, 0, 0): 91.298344
State (3, 0, 0, 0): 76.143053
State (3, 1, 0, 0): 83.179575
State (3, 2, 0, 0): 91.298344
State (3, 3, 0, 0): 100.000000
```

```
import numpy as np
import random
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
```

PART B

I am using the output of part a optimal policy.

2A)

```
def generate_episode(strategy):
    trajectory = []
    position = (0, 0, 0, 0)


    while position[:2] != (3, 3):
        if position not in strategy:
```

```
                print(f"Alert: No move found for position {position}. Terminating episode.")
                break

            move = strategy[position]

            if move is None:
                print(f"Reached terminal position {position}. No further moves.")
                break

            transitions = transition_model(position, move)
            if not transitions:
                print(f"Error: No transitions available for position {position} and move {move}.")
                break

            next_positions = list(transitions.keys())
            probabilities = list(transitions.values())

            next_position = random.choices(next_positions, probabilities)[0]

            score = rewards[position]
            trajectory.append((position, move, score))

            position = next_position
        trajectory.append((position, None, rewards[position]))

        return trajectory


    strategy_pi, values_pi = policy_iteration(discount_factor=0.95)
    episode = generate_episode(strategy_pi)
    for i in episode:
      position_value,move_value,score_value = i
      print(f"Position: {position_value[:2]}, Move: {move_value}, Score: {score_value}")
```

```
Position: (0, 0), Move: D, Score: -1
Position: (1, 0), Move: D, Score: -1
Position: (2, 0), Move: D, Score: -1
Position: (3, 0), Move: R, Score: -1
Position: (3, 1), Move: R, Score: -1
Position: (3, 1), Move: R, Score: -1
Position: (3, 2), Move: R, Score: -1
Position: (3, 3), Move: None, Score: 100
```

```
    def dagger_algorithm(cycles=10):
        dataset_X = []
        dataset_Y = []

        learned_strategy = {pos: random.choice(list(actions.keys())) if actions else None for pos in states}

        for cycle in range(1, cycles + 1):
            blend_factor = 0
            mixed_strategy = learned_strategy.copy()

            trajectory = generate_episode(mixed_strategy)

            for pos, action, _ in trajectory:
                if action is not None:
                    dataset_X.append(pos)
                    dataset_Y.append(action)

            if dataset_X:
                classifier = DecisionTreeClassifier()
                classifier.fit(dataset_X, dataset_Y)

                for pos in states:
                    learned_strategy[pos] = classifier.predict([pos])[0] if pos in dataset_X else learned_strategy[pos]

        return learned_strategy


    def evaluate_accuracy(learned_strategy, optimal_strategy):
        if not optimal_strategy:
            return 0.0

        correct_predictions = sum(1 for state in optimal_strategy if learned_strategy.get(state) == optimal_strategy[state])
        return correct_predictions / len(optimal_strategy)
```

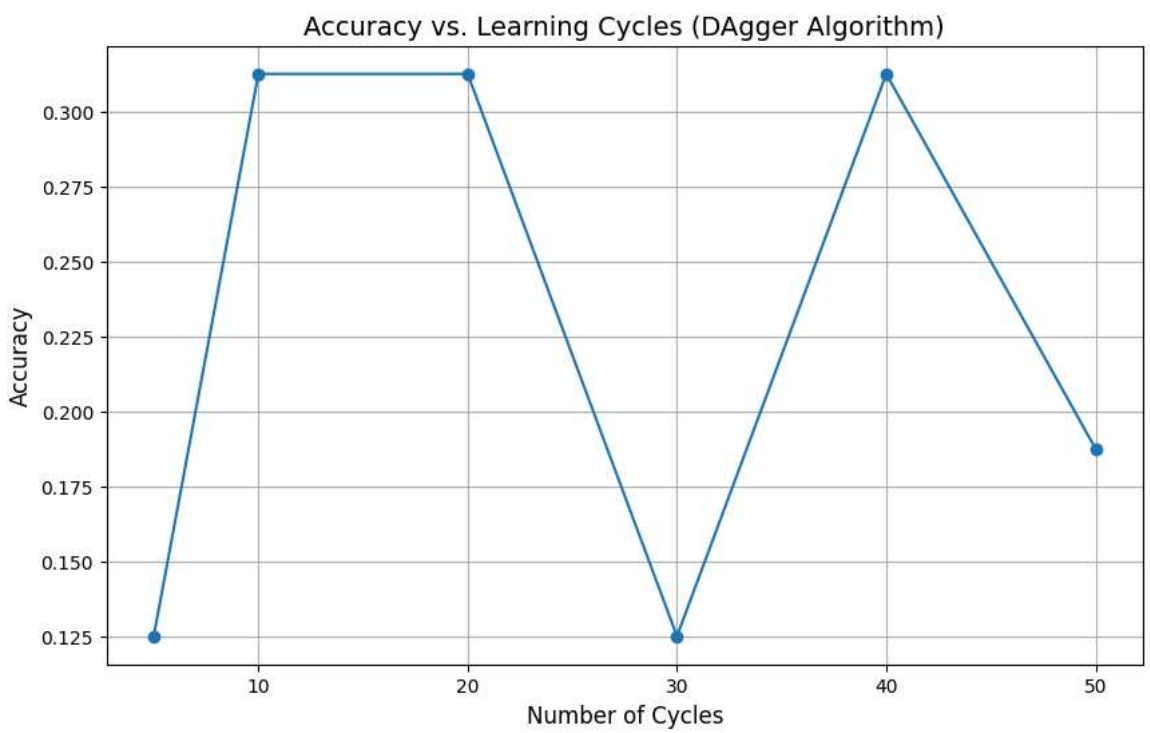
2B)

```
iteration_steps = [5, 10, 20, 30, 40, 50]
accuracy_results = []

optimal_strategy, _ = value_iteration(discount_factor=0.95)

for step in iteration_steps:
    trained_strategy = dagger_algorithm(cycles=step)
    accuracy = evaluate_accuracy(trained_strategy, optimal_strategy)
    accuracy_results.append(accuracy)

plt.figure(figsize=(10, 6))
plt.plot(iteration_steps, accuracy_results, marker='o', linestyle='-')
plt.title('Accuracy vs. Learning Cycles (DAgger Algorithm)', fontsize=14)
plt.xlabel('Number of Cycles', fontsize=12)
plt.ylabel('Accuracy', fontsize=12)
plt.grid(True)
plt.show()
```



Start coding or generate with AI.