CODE IMPLEMENTATION

Search Algorithms:

```
class PriorityQueue:
    """Define a PriorityQueue data structure that will be used"""
    def  __init__(self):
        self.Heap = []
        self.Count = 0
        self.len = 0

    def push(self, item, priority):
        entry = (priority, self.Count, item)
        heapq.heappush(self.Heap, entry)
        self.Count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.Heap)
        return item

    def isEmpty(self):
        return len(self.Heap) == 0
```

Depth – First Search:

```
def depthFirstSearch(gameState):
    """Implement depthFirstSearch approach"""
    beginBox = PosOfBoxes(gameState)
    beginPlayer = PosOfPlayer(gameState)
    startState = (beginPlayer, beginBox)
    frontier = collections.deque([[startState]])
    exploredSet = set()
    actions = [[0]]
    temp = []
    while frontier:
        node = frontier.pop()
        node_action = actions.pop()
        if isEndState(node[-1][-1]):
            temp += node_action[1:]
            print(f'Cost of dfs: {cost_dfs_bfs(temp)}')
            break
        if node[-1] not in exploredSet:
            exploredSet.add(node[-1])
            for action in legalActions(node[-1][0], node[-1][1]):
                newPosPlayer, newPosBox = updateState(node[-1][0], 
node[-1][1], action)
                if isFailed(newPosBox):
                    continue
                frontier.append(node + [(newPosPlayer, newPosBox)])
                actions.append(node_action + [action[-1]])
    # print('Length of the solution using dfs: %i' %len(temp))
    return temp
```

Breadth – First Search:

```
def breadthFirstSearch(gameState):
    """Implement breadthFirstSearch approach"""
    beginBox = PosOfBoxes(gameState) ### Initialize the coordinates of
box on the screen
    beginPlayer = PosOfPlayer(gameState) ### Initialize the
coordinates of player on the screen
    startState = (beginPlayer, beginBox) ### Start state is a
combination(tuple) of beginBox and beginPlayer
    frontier = collections.deque([[startState]]) ### Frontier is
declare as a collections (in this case is a double-ended queue).
    exploredSet = set() ### A set of explored node or closed set.
    actions = collections.deque([[0]]) ### A queue storing actions
    temp = [] ### This is the queue that store the solution.
    while frontier: ### Iterating through the frontier queue.
        node = frontier.popleft()   ### Pop the leftside node from the
frontier queue.
        node_action = actions.popleft() ### Pop the leftside action
from the actions queue.
        if isEndState(node[-1][-1]): ### Check whether the node is the
end state or not.
            temp += node_action[1:]  ### If it is, temp will save the
path from the beginning to the end.
            break; ### Exit the iteration.

        if node[-1] not in exploredSet: ### Check whether the node is
already explored or not.
            exploredSet.add(node[-1]) ### If not, add the node to the
explored set.
            for action in legalActions(node[-1][0], node[-1][1]): ###
Iterating through the set of legal actions.
                newPosPlayer, newPosBox = updateState(node[-1][0],
node[-1][1], action) ### Update the current position of player and box
corresponding to the action.
                if isFailed(newPosBox): ### Check whether the new
position of the box if failed or not
                    continue ### If it is, skip this loop.
                frontier.append(node + [(newPosPlayer, newPosBox)])
### Else, add the new position to the frontier queue.
                actions.append(node_action + [action[-1]]) ### Add new
action to the actions queue.
    # print('Length of the solution using bfs: %i' % len(temp))
    return temp ### Return the solution.
```

Uniform Cost Search:

```
def cost(actions):
    """A cost function"""
    return len([x for x in actions if x.islower()])

def uniformCostSearch(gameState):
```

```
    """Implement uniformCostSearch approach"""
    beginBox = PosOfBoxes(gameState)
    beginPlayer = PosOfPlayer(gameState)
    startState = (beginPlayer, beginBox)
    frontier = PriorityQueue()
    frontier.push([startState], 0)
    exploredSet = set()
    actions = PriorityQueue()
    actions.push([0], 0)
    temp = []
    while not frontier.isEmpty():
        node = frontier.pop()
        node_action = actions.pop()

        if isEndState(node[-1][-1]):
            temp += node_action[1:]
            break;

        if node[-1] not in exploredSet:
            exploredSet.add(node[-1])
            for action in legalActions(node[-1][0], node[-1][1
                newPosPlayer, newPosBox = updateState(node[-1][0],
node[-1][1], action
                if isFailed(newPosBox):
                    continue
                priority =  cost(node_action[1:] + [action[-1]])
                frontier.push(node + [(newPosPlayer, newPosBox)],
priority)
                actions.push(node_action + [action[-1]], priority
    return
```

A* Search:

```
def heuristic(posPlayer, posBox):
    distance = 0
    completes = set(posGoals) & set(posBox)
    sortposBox = list(set(posBox).difference(completes))
    sortposGoals = list(set(posGoals).difference(completes))
    for i in range(len(sortposBox)):
        distance += (abs(sortposBox[i][0] - sortposGoals[i][0])) +
(abs(sortposBox[i][1] - sortposGoals[i][1]))
    return distance

def heuristic_manhattan_weighted(posPlayer, posBox, alpha=0.5):
    distance = 0
    completes = set(posGoals) & set(posBox)
    sortposBox = list(set(posBox).difference(completes))
    sortposGoals = list(set(posGoals).difference(completes))
    for box in sortposBox:
        box_to_player = (abs(box[0] - posPlayer[0]) + abs(box[1] -
posPlayer[1]))
```

```
        goal = next(goal for goal in sortposGoals if goal not in
completes)
        box_to_goal = abs(box[0] - goal[0]) + abs(box[1] - goal[1])
        distance += box_to_player + box_to_goal
        distance += (box_to_player + box_to_goal) * alpha
    return distance

def heuristic_manhattan_plus(posPlayer, posBox):
    distance = 0
    completes = set(posGoals) & set(posBox)
    sortposBox = list(set(posBox).difference(completes))
    sortposGoals = list(set(posGoals).difference(completes))
    for i in range(len(sortposBox)):
        distance += (abs(sortposBox[i][0] - posPlayer[0]) +
abs(sortposBox[i][1] - posPlayer[1]))
        distance += (abs(sortposBox[i][0] - sortposGoals[i][0])) +
(abs(sortposBox[i][1] - sortposGoals[i][1]))
    return distance

def heuristic_euclidean_distance(posBox):
    distance = 0
    completes = set(posGoals) & set(posBox)
    sortposBox = list(set(posBox).difference(completes))
    sortposGoals = list(set(posGoals).difference(completes))
    for i in range(len(sortposBox)):
        distance += math.sqrt((sortposBox[i][0] -
sortposGoals[i][0])**2 + (sortposBox[i][1] - sortposGoals[i][1])**2)
    return distance

def heuristic_chebyshev_distance(posBox):
    distance = 0
    completes = set(posGoals) & set(posBox)
    sortposBox = list(set(posBox).difference(completes))
    sortposGoals = list(set(posGoals).difference(completes))
    for i in range(len(sortposBox)):
        distance += max(abs(sortposBox[i][0] - sortposGoals[i][0]),
abs(sortposBox[i][1] - sortposGoals[i][1]))
    return distance

def aStarSearch(gameState):
    beginBox = PosOfBoxes(gameState)
    beginPlayer = PosOfPlayer(gameState)
    temp = []
    start_state = (beginPlayer, beginBox)
    frontier = PriorityQueue()
    frontier.push([start_state], 0)
    exploredSet = set()
    actions = PriorityQueue()
    cnt_node = 0
    actions.push([0], heuristic(beginPlayer, start_state[1]))
    while len(frontier.Heap) > 0:
        node = frontier.pop()
```

```
        node_action = actions.pop()
        cnt_node += 1
        if isEndState(node[-1][-1]):
            temp += node_action[1:]
            break

        if node[-1] not in exploredSet:
            exploredSet.add(node[-1])

            for action in legalActions(node[-1][0], node[-1][1]):
                newPosPlayer, newPosBox = updateState(node[-1][0],
node[-1][1], action)

                if isFailed(newPosBox):
                    continue

                real_cost = cost(node_action[1:] + [action[-1]])
                f = heuristic(newPosPlayer, newPosBox) + real_cost
                # f = heuristic_euclidean_distance(newPosBox) +
real_cost
                # f = heuristic_chebyshev_distance(newPosBox) +
real_cost
                # f = heuristic_manhattan_plus(newPosPlayer,
newPosBox) + real_cost
                # f = heuristic_manhattan_weighted(newPosPlayer,
newPosBox) + real_cost
                frontier.push(node + [(newPosPlayer, newPosBox)], f)
                actions.push(node_action + [action[-1]], f)
    return temp
```

Adversarial Search Algorithms:

Minimax:

```
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        def minimax(state):
            bestValue, bestAction = None, None
            print(state.getLegalActions(0))
            value = []
            for action in state.getLegalActions(0):
                succ  = minValue(state.generateSuccessor(0, action),
1, 1)
                value.append(succ)
                if bestValue is None:
                    bestValue = succ
                    bestAction = action
                else:
                    if succ > bestValue:
                        bestValue = succ
                        bestAction = action
```

```
            print(value)
            return bestAction

        def minValue(state, agentIdx, depth):
            if agentIdx == state.getNumAgents():
                return maxValue(state, 0, depth + 1)
            value = None
            for action in state.getLegalActions(agentIdx):
                succ = minValue(state.generateSuccessor(agentIdx,
action), agentIdx + 1, depth)
                if value is None:
                    value = succ
                else:
                    value = min(value, succ)

            if value is not None:
                return value
            else:
                return self.evaluationFunction(state)

        def maxValue(state, agentIdx, depth):
            if depth > self.depth:
                return self.evaluationFunction(state)
            value = None
            for action in state.getLegalActions(agentIdx):
                succ = minValue(state.generateSuccessor(agentIdx,
action), agentIdx + 1, depth)
                if value is None:
                    value = succ
                else:
                    value = max(value, succ)
            if value is not None:
                return value
            else:
                return self.evaluationFunction(state)
        action = minimax(gameState)
        return action
```

Alpha – Beta:

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        def value(state):
            bestValue, bestAction = None, None
            alpha = float("-inf")
            beta = float("inf")
            value = []
            for action in state.getLegalActions(0):
                # value =
max(value,minValue(state.generateSuccessor(0, action), 1, 1))
                succ = minValue(state.generateSuccessor(0, action), 1,
1, alpha, beta)
```

```
                value.append(succ)
                if bestValue is None:
                    bestValue = succ
                    bestAction = action
                else:
                    if succ > bestValue:
                        bestValue = succ
                        bestAction = action
            print(value)
            return bestAction

        def minValue(state, agentIdx, depth, alpha, beta):
            if agentIdx == state.getNumAgents():
                return maxValue(state, 0, depth + 1, alpha, beta)
            value = None
            for action in state.getLegalActions(agentIdx):
                succ = minValue(state.generateSuccessor(agentIdx,
action), agentIdx + 1, depth, alpha, beta)
                if value is None:
                    value = succ
                    if value <= alpha:
                        return value
                    beta = min(beta, value)
                else:
                    value = min(value, succ)

            if value is not None:
                return value
            else:
                return self.evaluationFunction(state)

        def maxValue(state, agentIdx, depth, alpha, beta):
            if depth > self.depth:
                return self.evaluationFunction(state)
            value = None
            for action in state.getLegalActions(agentIdx):
                succ = minValue(state.generateSuccessor(agentIdx,
action), agentIdx + 1, depth, alpha, beta)
                if value is None:
                    value = succ
                else:
                    value = max(value, succ)
                    if value >= beta: return value
                    alpha = max(alpha, value)

            if value is not None:
                return value
            else:
                return self.evaluationFunction(state)

        action = value(gameState)
        return action
```

Expectimax:

```
class ExpectimaxAgent(MultiAgentSearchAgent):
   def getAction(self, gameState):
        def value(state):
            bestValue, bestAction = None, None
            value = []
            for action in state.getLegalActions(0):
                # value =
max(value,minValue(state.generateSuccessor(0, action), 1, 1))
                succ = expValue(state.generateSuccessor(0, action), 1,
1)
                value.append(succ)
                if bestValue is None:
                    bestValue = succ
                    bestAction = action
                else:
                    if succ > bestValue:
                        bestValue = succ
                        bestAction = action
            print(value)
            return bestAction

        def expValue(state, agentIdx, depth):
            if agentIdx == state.getNumAgents():
                return maxValue(state, 0, depth + 1)
            value = None
            p = 1 / (len(state.getLegalActions(agentIdx)) + 1e-6)
            # p = 0 if len(state.getLegalActions(agentIdx)) == 0 else
1 / (len(state.getLegalActions(agentIdx)))
            for action in state.getLegalActions(agentIdx):
                succ = expValue(state.generateSuccessor(agentIdx,
action), agentIdx + 1, depth)
                if value is None:
                    value = succ
                else:
                    value += p * succ
            if value is not None:
                return value
            else:
                return self.evaluationFunction(state)

        def maxValue(state, agentIdx, depth):
            if depth > self.depth:
                return self.evaluationFunction(state)
            value = None
            for action in state.getLegalActions(agentIdx):
                succ = expValue(state.generateSuccessor(agentIdx,
action), agentIdx + 1, depth)
                if value is None:
                    value = succ
                else:
```

```
                value = max(value, succ)

        if value is not None:
            return value
        else:
            return self.evaluationFunction(state)

    action = value(gameState)
    return action
```

MDP:

Policy Evaluation:

```
def policy_evaluation(env, policy, max_iters=500, gamma=0.9):
    # Initialize the values of all states to be 0
    v_values = np.zeros(env.observation_space.n)

    for i in range(max_iters):
        prev_v_values = np.copy(v_values)

        # Update the value of each state
        for state in range(env.observation_space.n):
            action = policy[state]

            # Compute the q-value of the action
            q_value = 0
            for prob, next_state, reward, done in
env.P[state][action]:
                q_value += prob * (reward + gamma *
prev_v_values[next_state])

            v_values[state] = q_value # update v-value

        # Check convergence
        if np.all(np.isclose(v_values, prev_v_values)):
            print(f'Converged at {i}-th iteration.')
            break

    return v_values
```

Value Iteration:

```
def value_iteration(env, max_iters=500, gamma=0.9):
    # initialize
    v_values = np.zeros(env.observation_space.n)

    for i in range(max_iters):
        prev_v_values = np.copy(v_values)

        # update the v-value for each state
        for state in range(env.observation_space.n):
```

```
            q_values = []

            # compute the q-value for each action that we can perform
at the state
            for action in range(env.action_space.n):
                q_value = 0
                # loop through each possible outcome
                for prob, next_state, reward, done in
env.P[state][action]:
                    q_value += prob * (reward + gamma *
prev_v_values[next_state])

                q_values.append(q_value)

            # select the max q-values
            best_action = np.argmax(q_values)
            v_values[state] = q_values[best_action]

        # check convergence
        if np.all(np.isclose(v_values, prev_v_values)):
            print(f'Converged at {i}-th iteration.')
            break

    return v_values
```

Policy Extraction:

```
def policy_extraction(env, v_values, gamma=0.9):
    # initialize
    policy = np.zeros(env.observation_space.n, dtype=np.int32)

    # loop through each state in the environment
    for state in range(env.observation_space.n):
        q_values = []
        # loop through each action
        for action in range(env.action_space.n):
            q_value = 0
            # loop each possible outcome
            for prob, next_state, reward, done in
env.P[state][action]:
                q_value += prob * (reward + gamma *
v_values[next_state])

            q_values.append(q_value)

        # select the best action
        best_action = np.argmax(q_values)
        policy[state] = best_action

    return policy
```

Policy Iteration:

```
def policy_iteration(env, max_iters = 500, max_pe_iters = 500, gamma =
0.9):
    # Initialize
    policy = np.random.randint(env.action_space.n, size =
(env.observation_space.n))
    # policy = np.zeros(env.observation_space.n, dtype=np.int32)
    v_values = np.zeros(env.observation_space.n)

    for iteration in range(max_iters):
        #Policy Evaluation
        for i in range(max_pe_iters):
            prev_v_values = np.copy(v_values)
            #Update the value for each state
            for state in range(env.observation_space.n):
                action = policy[state]

                # compute the q-value for each action that we can
perform at the state
                q_value = 0
                for prob, next_state, reward, done in
env.P[state][action]:
                    q_value += prob * (reward + gamma *
prev_v_values[next_state])

                v_values[state] = q_value # update v-value

            #Check convergence
            if np.all(np.isclose(v_values, prev_v_values)):
                break
        # Policy Improvement
        prev_policy = np.copy(policy)
        for state in range(env.observation_space.n):
            q_values = []
            for action in range(env.action_space.n):
                q_value = 0
                for prob, next_state, reward, done in
env.P[state][action]:
                    q_value += prob * (reward + gamma *
v_values[next_state])

                q_values.append(q_value)
            # Choose the best action
            best_action = np.argmax(q_values)
            policy[state] = best_action
        # Check convergence
        if np.all(prev_policy == policy):
            print(f'Converged at {iteration}-th iteration.')
            break
    return policy, v_values
```

Reinforcement Learning:

Q – Learning:

```
def q_learning(env, num_episodes, num_steps_per_episode,
learning_rate, gamma, max_epsilon, min_epsilon, epsilon_decay_rate):
    q_table = np.zeros((env.observation_space.n, env.action_space.n))
    rewards_all = []
    for episode in range(num_episodes):
        state, _ = env.reset()

        reward_episode = 0.0
        done = False
        epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-
epsilon_decay_rate*episode)
        for step in range(num_steps_per_episode):
            exploration = random.uniform(0,1)
            if exploration < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q_table[state, :])

            next_state, reward, done, info, _ = env.step(action)
            q_table[state, action] = q_table[state, action] * (1 -
learning_rate) + learning_rate * (reward + gamma *
np.max(q_table[next_state,:]))

            reward_episode += reward
            state = next_state

            if done:
                break
        rewards_all.append(reward_episode)
        print(f'Episode {episode} finished')
    return q_table, rewards_all
```

SARSA:

```
def SARSA(env, num_episodes, num_steps_per_episode, learning_rate,
gamma, max_epsilon, min_epsilon, epsilon_decay_rate):

    q_table = np.zeros((env.observation_space.n, env.action_space.n))
    rewards_all = []

    for episode in range(num_episodes):
        state, _ = env.reset()
        reward_episode = 0.0
        done = False
        epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-
epsilon_decay_rate*episode)
        if random.uniform(0,1) < epsilon:
            action = env.action_space.sample()
        else:
```

```
            action = np.argmax(q_table[state, :])

        for step in range(num_steps_per_episode):
            next_state, reward, done, info, _ = env.step(action)
            if random.uniform(0,1) < epsilon:
                next_action = env.action_space.sample()
            else:
                next_action = np.argmax(q_table[next_state, :])

            q_table[state, action] = q_table[state, action]  +
learning_rate * (reward + gamma * q_table[next_state, next_action] -
q_table[state, action])

            action = next_action
            state = next_state
            reward_episode += reward

            if done:
                break

        rewards_all.append(reward_episode)
        print(f'Episode {episode} finished')

    return q_table, rewards_all
```

Deep Reinforcement Learning:

```
class NeuralNetwork(nn.Module):
    def __init__(self, env):
        super(NeuralNetwork, self).__init__()

        self.network = nn.Sequential(
            nn.Linear(env.observation_space.shape[0], 64),
            nn.Tanh(),
            nn.Linear(64, env.action_space.n)
        )

    def forward(self, state):
        return self.network(state)

    def choose_action(self, state):
        state = torch.tensor(state, dtype=torch.float32)
        q_values = self(state.unsqueeze(0)) # pytorch requires inputs
in terms of batch
        best_action = torch.argmax(q_values, dim=1)[0]

        return best_action.detach().item()

def fill_memory(env):
    memory = deque(maxlen=memory_size)
    state = env.reset()
    for _ in range(min_replay_size):
```

```
        action = env.action_space.sample()
        next_state, reward, done, info = env.step(action)
        experience = (state, action, reward, done, next_state)
        memory.append(experience)
        state = next_state
        if done:
            env.reset()

    return memory
```

DQN:

```
def dqn_training(env, max_num_steps, max_epsilon, min_epsilon,
num_epsilon_decay_intervals, gamma, lr):
    q_net = NeuralNetwork(env)
    target_net = NeuralNetwork(env)
    target_net.load_state_dict(q_net.state_dict())
    optimizer = torch.optim.Adam(q_net.parameters(), lr=lr)

    memory = fill_memory(env)
    reward_buffer = deque(maxlen=100) # Rewards of the previous 100
episodes

    reward_per_episode = 0.0
    state = env.reset()
    all_rewards = []
    for step in range(max_num_steps):
        epsilon = np.interp(step, [0, num_epsilon_decay_intervals],
[max_epsilon, min_epsilon])

        random_number = np.random.uniform(0,1)
        if random_number <= epsilon:
            action = env.action_space.sample()
        else:
            action = q_net.choose_action(state)

        next_state, reward, done, info = env.step(action)
        experience = (state, action, reward, done, next_state)
        memory.append(experience)
        reward_per_episode += reward

        state = next_state

        if done:
            state = env.reset()
            reward_buffer.append(reward_per_episode)
            all_rewards.append((step, reward_per_episode))
            reward_per_episode = 0.0


        # Take a batch of experiences from the memory
        experiences = random.sample(memory, batch_size)
```

```
        states = [ex[0] for ex in experiences]
        actions = [ex[1] for ex in experiences]
        rewards = [ex[2] for ex in experiences]
        dones = [ex[3] for ex in experiences]
        next_states = [ex[4] for ex in experiences]

        states = torch.tensor(states, dtype=torch.float32)
        actions = torch.tensor(actions, dtype=torch.int64).unsqueeze(-
1) # (batch_size,) --> (batch_size, 1)
        rewards = torch.tensor(rewards,
dtype=torch.float32).unsqueeze(-1)
        dones = torch.tensor(dones, dtype=torch.float32).unsqueeze(-1)
        next_states = torch.tensor(next_states, dtype=torch.float32)

        # Compute targets using the formulation sample = r + gamma *
max q(s',a')
        target_q_values = target_net(next_states)
        max_target_q_values = target_q_values.max(dim=1,
keepdim=True)[0]
        targets = rewards + gamma * (1-dones) * max_target_q_values

        # Compute loss
        q_values = q_net(states)

        action_q_values = torch.gather(input=q_values, dim=1,
index=actions)
        loss = torch.nn.functional.mse_loss(action_q_values, targets)

        # gradient descent for q-network
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # update target network
        if (step+1) % target_update_frequency == 0:
            target_net.load_state_dict(q_net.state_dict())

        # print training results
        if (step+1) % 1000 == 0:
            average_reward = np.mean(reward_buffer)
            print(f'Episode: {len(all_rewards)} Step: {step+1} Average
reward: {average_reward}')

    return all_rewards, q_net
```