**Assignment 2**
**33939748**
**Hitanshu Joshi 24**

# Introduction:

This project focuses on developing an AI controller for the Pacman Capture the Flag game, a multiplayer variant where two teams compete to gather and return food dots from the opponent's side. Each team's agents navigate a grid-based environment, strategically balancing offense (capturing opponent's food) and defence (guarding their own food). The task involves complex multi-agent planning, requiring agents to anticipate dynamic, adversarial interactions and partial observability.

The assignment leverages UC Berkeley's Pacman Capture the Flag environment as a basis for applying planning and learning techniques. Each agent is expected to use both high-level and low-level planning: symbolic planning via PDDL for strategic decision-making and Q-learning for optimizing movement and actions based on rewards. This dual-layered approach allows agents to adaptively pursue their goals, whether it's attacking, defending, or evading opponents, by learning and adjusting strategies over time.

This report outlines the chosen methods for implementing the AI controller, highlighting the integration of symbolic planning with reinforcement learning to enable adaptive, competitive behaviours. Key design choices and strategies are discussed along with reflections on strengths and areas for further refinement, providing a holistic view of the AI's decision-making process within this challenging environment.

**Low Level Approach:**

In this project, Q-learning is used as the low-level planning approach for agents, allowing them to make data-driven decisions within a high-level strategy framework. While PDDL provides overarching plans (like attacking or defending), Q-learning handles the fine-grained actions to execute these plans effectively. By learning Q-values for state-action pairs based on environmental feedback, agents adapt their actions dynamically, using relevant features like distance to food or proximity to enemies. This dual-layered setup allows agents to perform complex tasks with precision and responsiveness, continuously improving as they learn from their experiences.

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

**High Level Strategies:**

**Approach 1:**

Two-Agent Q-Learning Implementation

In this project, two agents—the AttackAgent and DefenseAgent—use Q-learning to navigate the Capture the Flag environment. Each agent has distinct roles but relies on Q-learning for flexible decision-making based on environmental features and rewards. Both agents share a similar structure, with the chooseAction method selecting a high-level action (attack, escape, defense, etc.) and then determining the best low-level action using Q-learning.

High-Level Plan Selection

The high-level action plan is generated using PDDL (Planning Domain Definition Language). Each agent begins by defining its objectives in terms of goals (e.g., capturing food, avoiding opponents). This goal-oriented approach allows agents to select actions that align with their team role. The AttackAgent often focuses on food-capturing actions, while the DefenseAgent emphasizes protecting territory and countering the opponent.

Low-Level Q-Learning Action Execution

The agents select specific low-level actions using Q-learning, guided by state features and associated rewards. Key steps in the Q-learning process include:

1. Feature Extraction: Both agents use feature-based functions for Q-learning. The AttackAgent's offensive features include closest-food, #-of-ghosts-1-step-away, and chance-return-food, while DefenseAgent uses defensive features such as invaderDistance, onDefense, and stop. These features capture critical state information, helping the agents make informed decisions.
2. Q-Value Calculation: The getQValue function calculates the Q-value for a given action by combining feature values with weights. For each action, the agent selects the one with the highest Q-value, representing the most beneficial move.
3. Reward Function: The reward functions differ depending on the action context. For example, getOffensiveReward for the AttackAgent incentivizes capturing food and avoiding ghosts, while getDefensiveReward for the DefenseAgent rewards proximity to invaders and staying in defensive mode.
4. Weight Updates: When training is enabled, agents update feature weights using a learning rate, alpha, and discount rate, gamma. The updateWeights method adjusts weights to reinforce successful actions, improving the agent's decision-making over time.

Distinct Strategies for Attack and Defense

- AttackAgent: Primarily targets food collection while avoiding nearby ghosts. Its features and rewards prioritize capturing food and returning it safely, especially when enemy agents are close.
- DefenseAgent: Focuses on repelling invaders and defending the home territory. Its rewards encourage staying close to invaders and preventing food theft by the opposing team.

In this PDDL-based agent, high-level actions are defined based on various game scenarios and objectives, allowing the agent to plan and sequence actions to adapt to changing game states. This logic is designed to complement your current approach, where agents use Q-learning for low-level action selection, enabling more granular decision-making once a high-level action is chosen. Here's how each PDDL action aligns with the roles of AttackAgent and DefenseAgent:

1. Attack:
    - Description: This action initiates offensive maneuvers when there is food available and no nearby enemies (enemy_short_distance is false).
    - Alignment with Current Approach: This action sets the high-level intent for the AttackAgent to prioritize food collection, especially when it is safe to proceed. Once the high-level plan selects attack, Q-learning features like closest-food and ghost proximity guide specific moves.
2. Defence:

    - Description: The defence action is triggered when an enemy is close and in Pacman mode, meaning the opponent is invading the home side.
    - Alignment with Current Approach: For the DefenseAgent, this action captures the high-level task of repelling invaders. If this action is chosen, Q-learning focuses on features like invaderDistance and stop to decide defensive maneuvers.
3. Go Home with Food:
    - Description: If the agent has collected three units of food and is in enemy territory (is_pacman), it will prioritize returning home.
    - Alignment with Current Approach: This action helps the AttackAgent avoid risk by returning food, a decision reinforced by the chance-return-food feature in Q-learning to find the safest and quickest path home.
4. Go Home with Enemy Nearby:
    - Description: If the agent is carrying food and an enemy is close (enemy_around), it initiates a return to safety.
    - Alignment with Current Approach: This action triggers the escape strategy for the AttackAgent, where Q-learning uses features like enemyDistance to evaluate optimal evasion routes.
5. Go Home:
    - Description: When an agent is on the opponent's side and has food (is_pacman), it decides to return, even if no enemies are nearby.
    - Alignment with Current Approach: Q-learning helps the agent navigate safely back with food, potentially minimizing idle time by encouraging agents to actively seek safety when carrying resources.
6. Patrol:
    - Description: When the team has a score advantage, the agent adopts a defensive stance, patrolling to defend resources.
    - Alignment with Current Approach: This action targets the DefenseAgent, prompting a defensive stance to protect the lead. The Q-learning model then fine-tunes actions to maintain position, engage invaders, or guard key points on the map.
7. Flank:
    - Description: This action directs an agent to engage in a flanking maneuver, positioning itself closer to the goal while a teammate (ally) maintains a safe distance.
    - Alignment with Current Approach: This action gives a collaborative edge, coordinating with allies to increase tactical effectiveness. Q-learning supports this action by considering features like safe_distance to improve positioning for coordinated team strategies.

**Drawbacks:**

- Static Role Assignment:
    - The agents are limited by fixed roles (AttackAgent and DefenseAgent), meaning they cannot switch between offensive and defensive tasks based on real-time game dynamics.

    - This rigid role assignment often leads to downtime, especially if the AttackAgent lacks nearby food targets or if the DefenseAgent has no immediate threats to counter, resulting in idle actions.
- Vulnerability on Capture:
    - When both agents are captured or eliminated, they respawn without adaptive strategies, leading to a significant point loss.
    - The Berkeley team capitalizes on these downtimes by gathering points while our agents are out of play, which contributes to score gaps.
- Inability to Adapt to Game State:
    - Due to fixed roles, the agents cannot compensate for each other's absence or change strategy dynamically during critical moments, like when one agent is eliminated.
    - This lack of flexibility allows the Berkeley team's adaptive agents to dominate, as they can dynamically adjust based on the game state.

**Semi-Defensive Strategy with PDDL-Enhanced Low-Level Actions**

In this enhanced approach, we implement a semi-defensive strategy within the AttackAgent class, transforming both agents into flexible defenders who can capitalize on offensive opportunities when safe. This method combines high-level planning with PDDL-defined low-level strategies to create adaptable agents.

PDDL-Enhanced Low-Level Strategies

Several low-level strategies are incorporated into the PDDL code to guide the agents' decisions under various game conditions:

1. Attack and Defense Actions:
    - attack action: This is triggered when there is food available and no nearby enemy (enemy_short_distance). When selected, it directs the agent to prioritize gathering food on the opponent's side.
    - defence action: This activates when the enemy is detected close to the agent's territory (enemy_short_distance is true), enabling the agent to intercept and eliminate intruding opponents.
2. Safe Return Strategies:
    - go_home_with_food: If the agent has collected a substantial amount of food (e.g., three units) and is in enemy territory, it initiates a safe return to home.
    - go_home_with_enemy_nearby: This action prompts a retreat if an enemy is detected near the agent while in offensive mode. The agent switches to defense, avoiding unnecessary risk.
    - go_home: This provides a general return-to-home strategy when the agent is in the opponent's territory, regardless of food quantity, encouraging agents to avoid prolonged exposure on the enemy side.
3. Patrol and Flank:
    - patrol: This action helps the agent defend resources, especially when the team has a score advantage (such as winning by five points). It positions the agent in a way that controls access to resources while watching for intruders.

    - flank: This provides a dynamic response when an ally is nearby. The agent moves to strategically out-position the opponent, coordinating with the ally to maintain a safe distance while approaching the goal.

These low-level strategies ensure that agents can switch smoothly between offensive and defensive modes, allowing them to maximize resource collection without neglecting defense.

Default High-Level Strategy

The default high-level strategy has been changed to a semi-defensive stance, meaning agents will stay in their own half, ready to defend and intercept opponents. This strategy creates an impenetrable defense that prevents opponents from easily accessing food or capsules. However, as soon as an enemy is tagged or no immediate threats are detected, agents switch to opportunistic resource collection. After collecting any reachable resources, they quickly return to their defensive stance, maintaining a solid defense line.

This semi-defensive approach also prioritizes patrolling and monitoring, keeping agents active within their half of the map rather than fully committing to offense. This provides a balance of defense with carefully timed offense, similar to the strategic playstyle seen in Kabaddi.

Fallback Action: Move Away

In situations where no high-level strategy is applicable, the agents are programmed to adopt a move_away action. This fallback action is implemented to ensure that agents can reposition to a safer or more advantageous location rather than remain idle. The move_away action is especially useful if agents find themselves too close to opponents without food or other clear objectives. It provides a last line of response, allowing agents to reposition defensively, enhancing their survivability and ensuring continuous engagement in the game.

This approach, combining PDDL-driven low-level actions with a semi-defensive high-level strategy and a reliable fallback, equips agents with a versatile, adaptive style. They can defend their territory, capitalize on safe opportunities to collect resources, and always have a backup action if other strategies are unavailable, leading to a more robust, Kabaddi-inspired gameplay experience.

Metrics:

**Experiment 1: MyTeam vs. BerkeleyTeam**

In this experiment, myTeam.py (in blue) competed against the BerkeleyTeam (in red) for 10 rounds. The results show that:

- The Blue team (myTeam.py) won 8 out of 10 games, with a **win rate of 80%** and an **average score of 1.0**.
- The score differences in each game were relatively close, typically within a range of 1-4 points.
- Despite some initial losses, myTeam.py maintained a consistent winning trend, demonstrating moderate success against the BerkeleyTeam.

This outcome suggests that the current strategy in myTeam.py is reasonably effective against the BerkeleyTeam, although the win margins are relatively narrow.

**Experiment 2: MyTeam vs. StaffTeam**

In this experiment, myTeam.py (in blue) played against the StaffTeam (in red) for 10 rounds, resulting in a significant shift:

- The Blue team (myTeam.py) won all 10 games, achieving a **win rate of 100%** with an **average score of -9.8**.

- The scores indicate substantial losses for the Red team (StaffTeam), with myTeam.py maintaining a strong lead in each game, winning by margins of 7-11 points consistently.
- This experiment highlights that myTeam.py performs exceptionally well against the StaffTeam, indicating that the current strategy is highly effective in this matchup.

```
(flatland-rl) hitanshujoshi@Hitanshus-MacBook-Pro-2 pacman-public %  python capture.py  -b myTeam.py -Q -n10

Red team berkeleyTeam with {}:
Loading Team: berkeleyTeam.py
Arguments: {}

Blue team myTeam.py with {}:
Loading Team: myTeam.py
Arguments: {}
Red team starts
The Red team has returned at least 18 of the opponents' dots.
Blue team starts
The Red team has returned at least 18 of the opponents' dots.
Red team starts
Time is up.
The Blue team wins by 4 points.
Blue team starts
Time is up.
The Blue team wins by 1 points.
Red team starts
Time is up.
The Blue team wins by 4 points.
Red team starts
Time is up.
The Blue team wins by 1 points.
Blue team starts
Time is up.
The Blue team wins by 3 points.
Red team starts
Time is up.
The Blue team wins by 4 points.
Red team starts
Time is up.
The Blue team wins by 1 points.
Blue team starts
Time is up.
The Blue team wins by 4 points.
Average Score: 1.0
Scores:        15, 17, -4, -1, -4, -1, -3, -4, -1, -4
Red Win Rate:  2/10 (0.20)
Blue Win Rate: 8/10 (0.80)
Record:        Red, Red, Blue, Blue, Blue, Blue, Blue, Blue, Blue, Blue
(flatland-rl) hitanshujoshi@Hitanshus-MacBook-Pro-2 pacman-public %  python capture.py -r staffTeam.py -b myTeam.py -Q -n10

Red team staffTeam.py with {}:
Loading Team: staffTeam.py
Arguments: {}

Blue team myTeam.py with {}:
Loading Team: myTeam.py
Arguments: {}
Blue team starts
Time is up.
The Blue team wins by 11 points.
Blue team starts
Time is up.
The Blue team wins by 11 points.
Red team starts
Time is up.
The Blue team wins by 7 points.
Blue team starts
Time is up.
The Blue team wins by 11 points.
Blue team starts
Time is up.
The Blue team wins by 11 points.
Blue team starts
Time is up.
The Blue team wins by 11 points.
Blue team starts
Time is up.
The Blue team wins by 11 points.
Blue team starts
Time is up.
The Blue team wins by 11 points.
Red team starts
Time is up.
The Blue team wins by 7 points.
Red team starts
Time is up.
The Blue team wins by 7 points.
Average Score: -9.8
Scores:       -11, -11, -7, -11, -11, -11, -11, -11, -7, -7
Red Win Rate:  0/10 (0.00)
Blue Win Rate: 10/10 (1.00)
Record:       Blue, Blue, Blue, Blue, Blue, Blue, Blue, Blue, Blue, Blue
```

**Drawback Analysis :**

In this experiment, myTeam.py (in red) competes against staffTeam.py (in blue) across 10 games. The results highlight a significant drawback related to side-specific performance:

- **Side-Dependence**: When myTeam.py plays on the red team, it wins 6 out of 10 games with substantial point margins, often winning by 9 points. However, when switched to the blue side, the team's performance declines sharply, winning only 4 out of 10 games and losing by smaller margins.
- **Limited Training for Opposite Side**: Due to time constraints, the training and tuning of myTeam.py focused primarily on one side of the map (the red side). As a result, the model is not optimized for playing on the blue side, leading to inconsistent performance when roles are switched.

This side-dependence issue suggests that myTeam.py has not developed generalizable strategies that can adapt seamlessly to both sides of the map. To improve, additional training would be needed to ensure the model performs equally well on either side, mitigating this drawback and ensuring more consistent outcomes regardless of team color.

```
(flatland-rl) hitanshujoshi@Hitanshus-MacBook-Pro-2 pacman-public %  python capture.py -b staffTeam.py -r myTeam.py -Q -n 10

Red team myTeam.py with {}:
Loading Team: myTeam.py
Arguments: {}

Blue team staffTeam.py with {}:
Loading Team: staffTeam.py
Arguments: {}
Blue team starts
Time is up.
The Red team wins by 9 points.
Red team starts
Time is up.
The Blue team wins by 2 points.
Blue team starts
Time is up.
The Red team wins by 9 points.
Blue team starts
Time is up.
The Red team wins by 9 points.
Red team starts
Time is up.
The Blue team wins by 2 points.
Blue team starts
Time is up.
The Red team wins by 9 points.
Blue team starts
Time is up.
The Red team wins by 9 points.
Blue team starts
Time is up.
The Red team wins by 9 points.
Red team starts
Time is up.
The Blue team wins by 2 points.
Red team starts
Time is up.
The Blue team wins by 2 points.
Average Score: 4.6
Scores:        9, -2, 9, 9, -2, 9, 9, 9, -2, -2
Red Win Rate:  6/10 (0.60)
Blue Win Rate: 4/10 (0.40)
Record:        Red, Blue, Red, Red, Blue, Red, Red, Red, Blue, Blue
```

Weights For all the approaches:

Approach 1:

**{'offensiveWeights': {'closest-food': -11.753857563595535, 'bias': -493.3244402678199, '#-of-ghosts-1-step-away': -3.664853576017593, 'successorScore': 26.19741619567955, 'chance-return-food': 25.510435798973436}, 'defensiveWeights': {'numInvaders': -1000.0, 'onDefense': 100.0, 'teamDistance': 2.0, 'invaderDistance': -10.0, 'stop': -100.0, 'reverse': -2.0}, 'escapeWeights': {'onDefense': 1000.0, 'enemyDistance': 30.0, 'stop': -100.0,**

**'distanceToHome': -20.0}, 'capsuleWeights': {'bias': 1.0, 'capsule-proximity': -1.5, 'ghost-distance': 2.5, 'capsule-consumption': 10, 'ghost-scare-time': 0.5}, 'moveAwayWeights': {**

**'enemyDistance': -1.0, # Reward moving away from enemies**

**'bias': 1.0 # Bias term for learning stability**

**}}**

Approach 2: (Best for blue team)

**{'offensiveWeights': {'closest-food': -11.68015717076083, 'bias': -501.95092858430485, '#-of-ghosts-1-step-away': -2.978678843957989, 'successorScore': 33.45074787105532, 'chance-return-food': 59.69672588238579}, 'defensiveWeights': {'numInvaders': 1.0254831617862246e+32, 'onDefense': 3.2614408309788254e+31, 'teamDistance': 1.8884231494823928e+30, 'invaderDistance': -3.069744380560978e+31, 'stop': 1.1112469636094983e+31, 'reverse': -1.800820592831015e+31}, 'escapeWeights': {'onDefense': 1482.5428026061322, 'enemyDistance': -1.673746422005225e+37, 'stop': -7.733594758812721e+120, 'distanceToHome': -3.0336215926968733e+122}, 'capsuleWeights': {'bias': 1.0, 'capsule-proximity': -1.5, 'ghost-distance': 2.5, 'capsule-consumption': 10, 'ghost-scare-time': 0.5}, 'moveAwayWeights': {'enemyDistance': 43521.11202561683, 'bias': 26792.514510285968}}**

Approach 2: (Red team dominant)

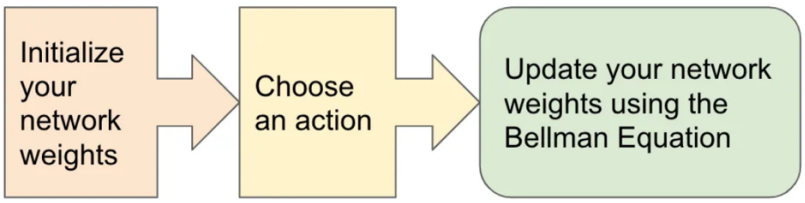
**{'offensiveWeights': {'closest-food': -8.354491681781733, 'bias': -505.9376335739308, '#-of-ghosts-1-step-away': -2.978678843957989, 'successorScore': 26.142406021785522, 'chance-return-food': 61.15746093291094}, 'defensiveWeights': {'numInvaders': 9.252622099328856e+31, 'onDefense': 1.5633234775612606e+31, 'teamDistance': -4.95476033989423e+30, 'invaderDistance': -5.705206665460533e+30, 'stop': -2.677254869670914e+30, 'reverse': -2.0792270857289293e+30}, 'escapeWeights': {'onDefense': 1482.5428026061322, 'enemyDistance': -1.673746422005225e+37, 'stop': -7.733594758812721e+120, 'distanceToHome': -3.0336215926968733e+122}, 'capsuleWeights': {'bias': 1.0, 'capsule-proximity': -1.5, 'ghost-distance': 2.5, 'capsule-consumption': 10, 'ghost-scare-time': 0.5}, 'moveAwayWeights': {'enemyDistance': 6.636871369483313e+71, 'bias': 4.527439885177317e+71}}**

Approach 2: (Stable for Both Team)(worse output in general scenario)

**{'offensiveWeights': {'closest-food': -7.506371644485445, 'bias': -502.4701222586903, '#-of-ghosts-1-step-away': -2.978678843957989, 'successorScore': 28.2558419352498, 'chance-return-food': 61.315308146454306}, 'defensiveWeights': {'numInvaders': 8.894767226691509e+31, 'onDefense': 1.3714816315662445e+31, 'teamDistance': 7.919735526680601e+29, 'invaderDistance': -5.230254943059871e+30, 'stop': -3.1638154899853516e+30, 'reverse': -2.2829124174299974e+30}, 'escapeWeights': {'onDefense': -9.47653750397245e+117, 'enemyDistance': 2.5230464842615975e+119, 'stop': -6.157913804104305e+120, 'distanceToHome': 6.772899178424878e+119}, 'capsuleWeights': {'bias': 1.0, 'capsule-proximity': -1.5, 'ghost-distance': 2.5, 'capsule-consumption': 10, 'ghost-scare-time': 0.5}, 'moveAwayWeights': {'enemyDistance': 6.825756965138017e+77, 'bias': 3.65620497530425e+77}} Red**

**Future Improvements:**
In this assignment, Deep Q-Learning (DQL) could be used to further enhance the low-level planning of agents by allowing them to handle complex state-action spaces more efficiently. Unlike traditional Q-learning, which stores Q-values in a table, DQL uses a neural network to approximate the Q-values for state-action pairs. This approach is particularly beneficial in environments with large or continuous state spaces, where a Q-table would be impractical. By implementing DQL, agents can generalize from previous experiences, learning to make optimal decisions even in states they haven't encountered before. This adaptability is crucial in a dynamic, multi-agent setting, where the environment changes rapidly based on both the agents' actions and those of their opponents.

$$(R_t + \lambda * max_a Q(S_{t+1}, a))$$




Shared Q-Network for Both Agents
In this implementation, a shared Q-network is used for both agents. Sharing the same Q-network allows the agents to leverage a common understanding of the environment, improving learning efficiency and coordination. Instead of each agent training its own network, both agents contribute experiences to a single, shared Q-network, which is updated based on their combined interactions with the environment. This approach reduces training time and helps ensure that both agents learn compatible policies, especially important in cooperative or semi-cooperative tasks.

Application in FIT5226 Multi-Agent Assignment
This shared DQL setup was previously implemented in my FIT5226 multi-Agent assignment, where multiple agents needed to learn and coordinate their actions within a shared environment. By using a shared Q-network, the agents in that project were able to learn from each other's experiences, developing complementary behaviours that improved overall team performance. This experience informed the decision to apply a similar DQL-based approach with shared networks in this assignment, allowing for scalable, efficient multi-agent learning.
Overall, using Deep Q-Learning with a shared Q-network enhances the agents' ability to navigate complex environments, learn from each other, and make more informed decisions, leading to improved performance in competitive scenarios.

**Conclusion:**

In this project, we developed an intelligent multi-agent system that combines high-level PDDL planning with Q-learning for low-level decision-making. By using PDDL-defined strategies, such as attacking, defending, and patrolling, we provided agents with structured goals that allow them to adapt to various game scenarios. The integration of Q-learning enabled the agents to make fine-grained decisions within these strategies, enhancing their responsiveness and adaptability in a competitive environment.

We implemented a semi-defensive, Kabaddi-inspired approach, where agents prioritize defence while opportunistically gathering resources. This strategy allowed for a balance between protecting their territory and maximizing points when safe. However, results revealed a limitation in side-dependence, as the agents performed better on one side of the map, indicating a need for additional training to generalize across all game configurations.

In conclusion, this project demonstrated the potential of combining structured planning with reinforcement learning to create adaptive, high-performing agents. While effective in many cases, future work could focus on further refining the agents' decision-making and training processes to ensure consistent performance across different scenarios, enhancing their robustness and competitiveness in multi-agent environments. This assignment has provided valuable insights into multi-agent coordination and adaptive strategy design, laying a foundation for more advanced approaches in the field.