CS 4365 Pacman CTF Project Report

Fawaz Ahmed

May 9, 2022

Introduction

This project was for the class CS 4365 Artificial Intelligence with Professor Yu Chung Vincent Ng. The game that was used is called Pacman CTF (capture the flag). In this game there are two teams of two agents. Each team has half of the map as their territory. In the team's territory there are food or "pellets" that needs to be protected. The enemy team's territory contains food as well, however the agents need to collect these. The goal of the game is to collect the more pellets than the enemy team within the given time limit. There are also special capsules that spawn in the enemy territory that can be eaten. When eaten the agents can eat the enemy agents for a limited amount of time. Once the time limit is over the agents can no longer eat the enemy and they instead can be eaten. The way to control these agents is using an AI created by us inside of the myTeam.py file.

My strategy was to code two different AI, the first would be centered towards defense and the second would be towards offense. The defensive agent will not leave its territory and would defend and chase any enemy Pacman invaders. The defensive agent will attempt to eat the invader. In the case where the invader eats a capsule, the defensive agent will get as close as possible without touching the invader. The second AI is the offensive agent, and it will leave its territory and go to the nearest pellet. Once it eats the pellet it will come back to its territory to gain a point. The process repeats until the game ends, it gets eaten, or gets stuck.

Theoretical Foundation

The technique that was used was building a reflex agent. These agents do not remember past actions/moves, rather they look at future moves "successors" and pick the one that returns the greatest value. The value is calculated by looking at the game state and using algorithms to collect the different feature points/inputs, multiplying them with different weight values, and adding them together to get the sum. This process is depicted in figure 1 bellow. For example, an action that takes an agent closer to a pellet would be rated higher than an action that ends up killing the agent.

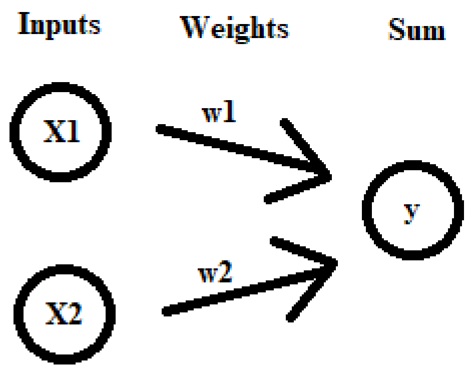


Figure 1. How a sum is calculated given features/inputs and its weights

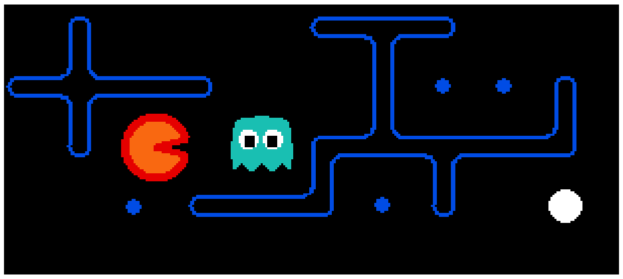


Figure 2. Example of a successor that is not preferred

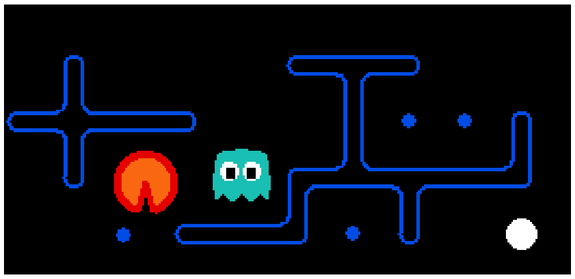


Figure 3. Example of a successor that is preferred

For example, in figure 2 above, the Pacman is going towards the ghost and away from the pellet. This successor would be evaluated quite low compared to the second figure (figure 3) that shows the Pacman going away from the ghost and towards the pellet. A way to calculate this is by having weights for each input. There are two types of weights, a positive and negative weight. A positive weight indicates an input is preferred, for example, could be the distance between the Pacman and the closest enemy ghost which would be a positive weight. The greater the distance the greater the output. A negative weight would indicate an input we want to decrease. For example, the distance between a Pacman and the nearest pellet. Here we want to keep decreasing the distance between the Pacman and the pellet so a negative weight would be used.

This type of technique is essentially what I used to create the defensive and offensive agent AIs for my team. I will explain each agent and their weights in greater detail in the next section. Each weight will have its own section explain its purpose and how it is calculated

Final Agent Description

Overview

I have implemented two different agents, the first being a defensive orientated agent. This agent focusses on only defending food and preventing enemy agents from entering my territory. The second agent is offensive oriented and has the job of getting pellets to increase my points. The reason why I chose two different types of agents was so that the code would be simpler to read and write since each agent is only in charge of a single task. A better approach would have been to code agents that are flexible in how they act. For example, an agent that can both attack and defend would be better than an agent that can only defend. This implementation would be something that can be added on into the code in a future date.

Defensive Agent

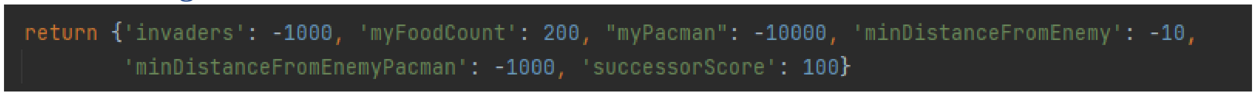

```
return {'invaders': -1000, 'myFoodCount': 200, "myPacman": -10000, 'minDistanceFromEnemy': -10,
        'minDistanceFromEnemyPacman': -1000, 'successorScore': 100}
```

Figure 4. Weights of the defensive agent

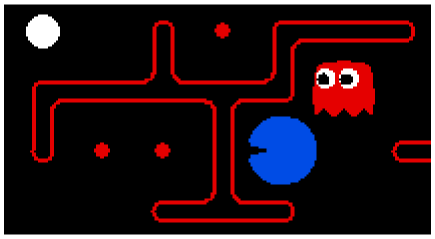
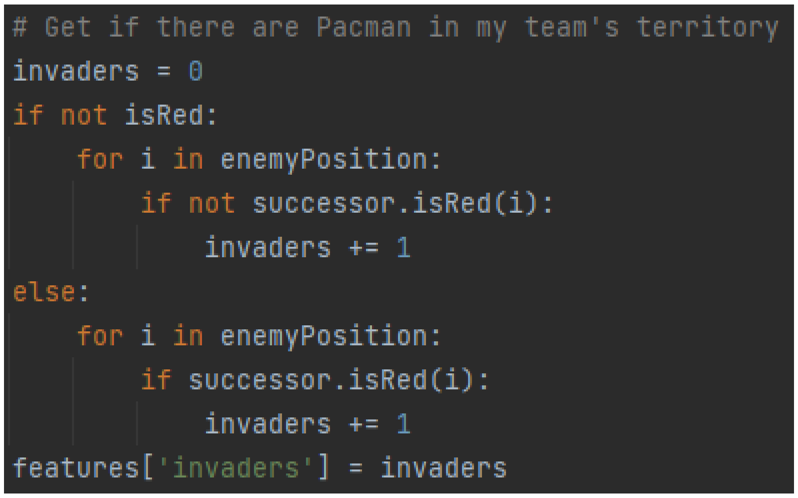


Figure 5. Defensive agent (red ghost) chasing enemy Pacman (blue Pacman)

Invaders

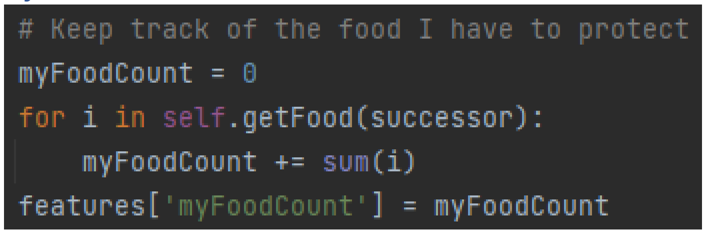


```
# Get if there are Pacman in my team's territory
invaders = 0
if not isRed:
    for i in enemyPosition:
        if not successor.isRed(i):
            invaders += 1
else:
    for i in enemyPosition:
        if successor.isRed(i):
            invaders += 1
features['invaders'] = invaders
```

Figure 6. Code to calculate invaders

This feature is calculated by counting the number of invaders (enemy agents in my territory). Since the weight of this feature is negative, we want to decrease it. The way to do this is by eating the enemy agent.

myFoodCount

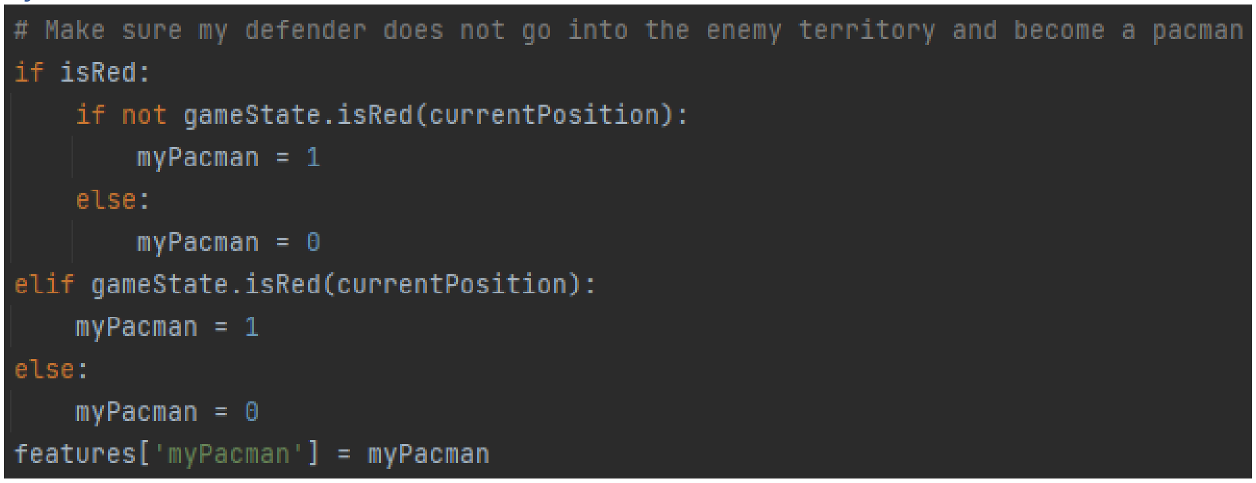


```
# Keep track of the food I have to protect
myFoodCount = 0
for i in self.getFood(successor):
    myFoodCount += sum(i)
features['myFoodCount'] = myFoodCount
```

Figure 7. Code to calculate myFoodCount

The weight of this feature is positive, so we want to try our best to increase or to not decrease this number. This feature is used to dissuade the defensive agent from letting the enemy Pacman from eating the team's pellets.

myPacman

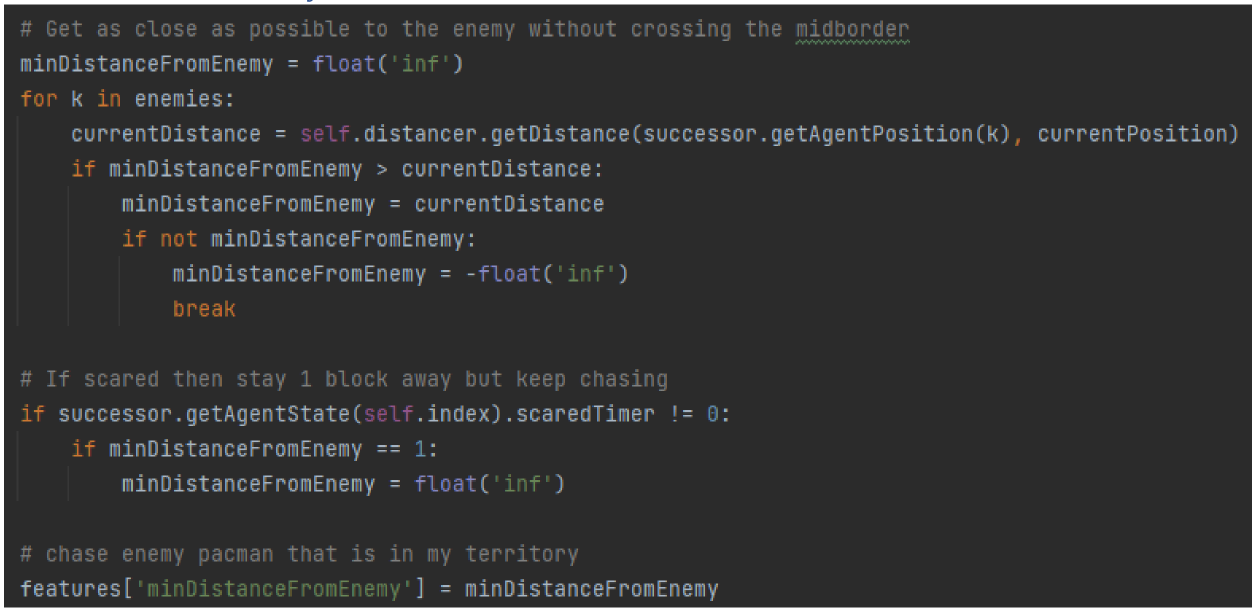


```
# Make sure my defender does not go into the enemy territory and become a pacman
if isRed:
    if not gameState.isRed(currentPosition):
        myPacman = 1
    else:
        myPacman = 0
elif gameState.isRed(currentPosition):
    myPacman = 1
else:
    myPacman = 0
features['myPacman'] = myPacman
```

Figure 8. Code to calculate if agent is a Pacman or not

This feature prevents the defensive agent from going into the enemy territory. This is done so that this agent doesn't accidently chase enemy agents into their territory which would kill it. Since the weight of this feature is very low the agent has an action that forces it to go into the enemy territory.

minDistanceFromEnemy



```
# Get as close as possible to the enemy without crossing the midborder
minDistanceFromEnemy = float('inf')
for k in enemies:
    currentDistance = self.distancer.getDistance(successor.getAgentPosition(k), currentPosition)
    if minDistanceFromEnemy > currentDistance:
        minDistanceFromEnemy = currentDistance
        if not minDistanceFromEnemy:
            minDistanceFromEnemy = -float('inf')
            break

# If scared then stay 1 block away but keep chasing
if successor.getAgentState(self.index).scaredTimer != 0:
    if minDistanceFromEnemy == 1:
        minDistanceFromEnemy = float('inf')

# chase enemy pacman that is in my territory
features['minDistanceFromEnemy'] = minDistanceFromEnemy
```

Figure 9. Code to calculate the minimum distance from an enemy ghost

This code calculates the distance from the enemy ghost and uses it to choose the closest route to it. This is done to prevent the enemy ghost from entering my team's territory. This type of action is called body blocking and is often used in video games to prevent the enemy from making progress.

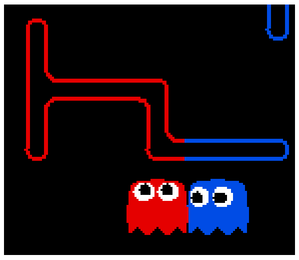
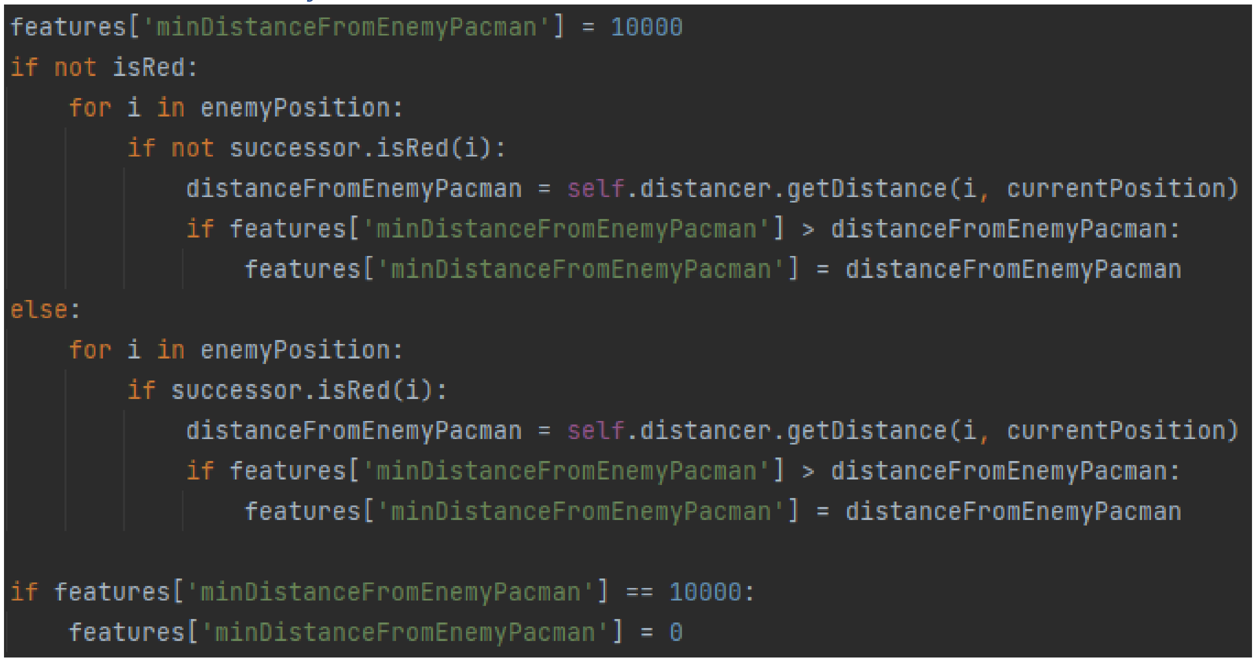


Figure 10. Defensive agent (red) is body blocking the enemy ghost (blue)

For example, in figure 10 the defensive agent (red) is body blocking the enemy ghost (blue) from entering and eating the pellets.

minDistanceFromEnemyPacman



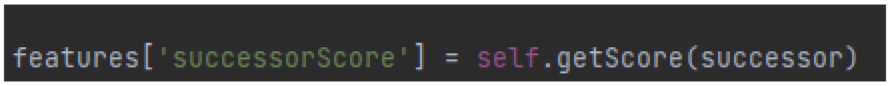
```
features['minDistanceFromEnemyPacman'] = 10000
if not isRed:
    for i in enemyPosition:
        if not successor.isRed(i):
            distanceFromEnemyPacman = self.distancer.getDistance(i, currentPosition)
            if features['minDistanceFromEnemyPacman'] > distanceFromEnemyPacman:
                features['minDistanceFromEnemyPacman'] = distanceFromEnemyPacman
else:
    for i in enemyPosition:
        if successor.isRed(i):
            distanceFromEnemyPacman = self.distancer.getDistance(i, currentPosition)
            if features['minDistanceFromEnemyPacman'] > distanceFromEnemyPacman:
                features['minDistanceFromEnemyPacman'] = distanceFromEnemyPacman

if features['minDistanceFromEnemyPacman'] == 10000:
    features['minDistanceFromEnemyPacman'] = 0
```

Figure 11. Code to calculate the minimum distance from an enemy Pacman

This code is used to reduce the agent's distance from an enemy Pacman. It takes calculates the minimum distance from the agent's current position and the enemy Pacman's current position. This action is shown in figure 5. If there is no enemy Pacman in my territory, then it returns 0 which doesn't affect the sum (i.e., no changes need to be made)

successorScore

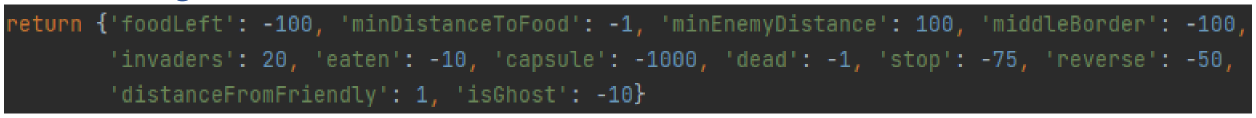


```
features['successorScore'] = self.getScore(successor)
```

Figure 12. Code to get the successor score

This simple code is used to return the successor score so that successors that have greater score are preferred.

Offensive Agent



```
return {'foodLeft': -100, 'minDistanceToFood': -1, 'minEnemyDistance': 100, 'middleBorder': -100,
        'invaders': 20, 'eaten': -10, 'capsule': -1000, 'dead': -1, 'stop': -75, 'reverse': -50,
        'distanceFromFriendly': 1, 'isGhost': -10}
```

Figure 13. Weights of the Offensive agent

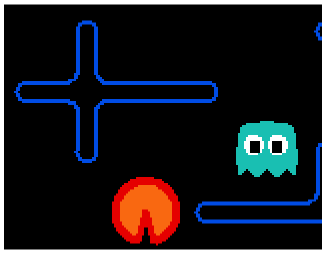
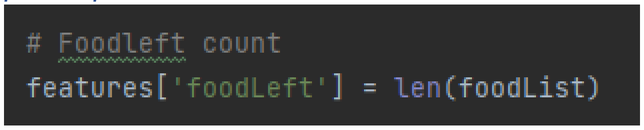


Figure 14. Offensive agent (orange Pacman) running away from an enemy Pacman (green-blue ghost)

foodLeft

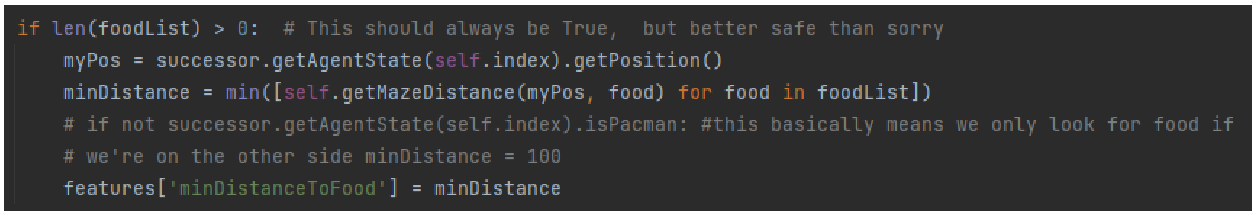


```
# Foodleft count
features['foodLeft'] = len(foodList)
```

Figure 15. Code to get the number of food left

Get good left and since it is a negative weight, we want to reduce the number of foods that needs to be eaten.

minDistanceToFood

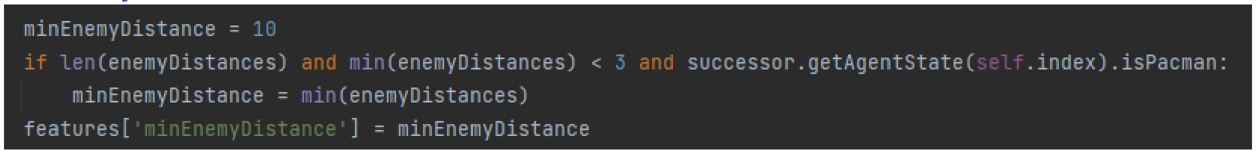


```
if len(foodList) > 0:  # This should always be True,  but better safe than sorry
    myPos = successor.getAgentState(self.index).getPosition()
    minDistance = min([self.getMazeDistance(myPos, food) for food in foodList])
    # if not successor.getAgentState(self.index).isPacman: #this basically means we only look for food if
    # we're on the other side minDistance = 100
    features['minDistanceToFood'] = minDistance
```

Figure 16. Code to get the distance to the closest food

Here we loop through all the food coordinates and get the distance to the closest pellet. Since the weight is negative, we want to reduce this distance as much as possible. This would mean we would eventually eat the pellet.

minEnemyDistance

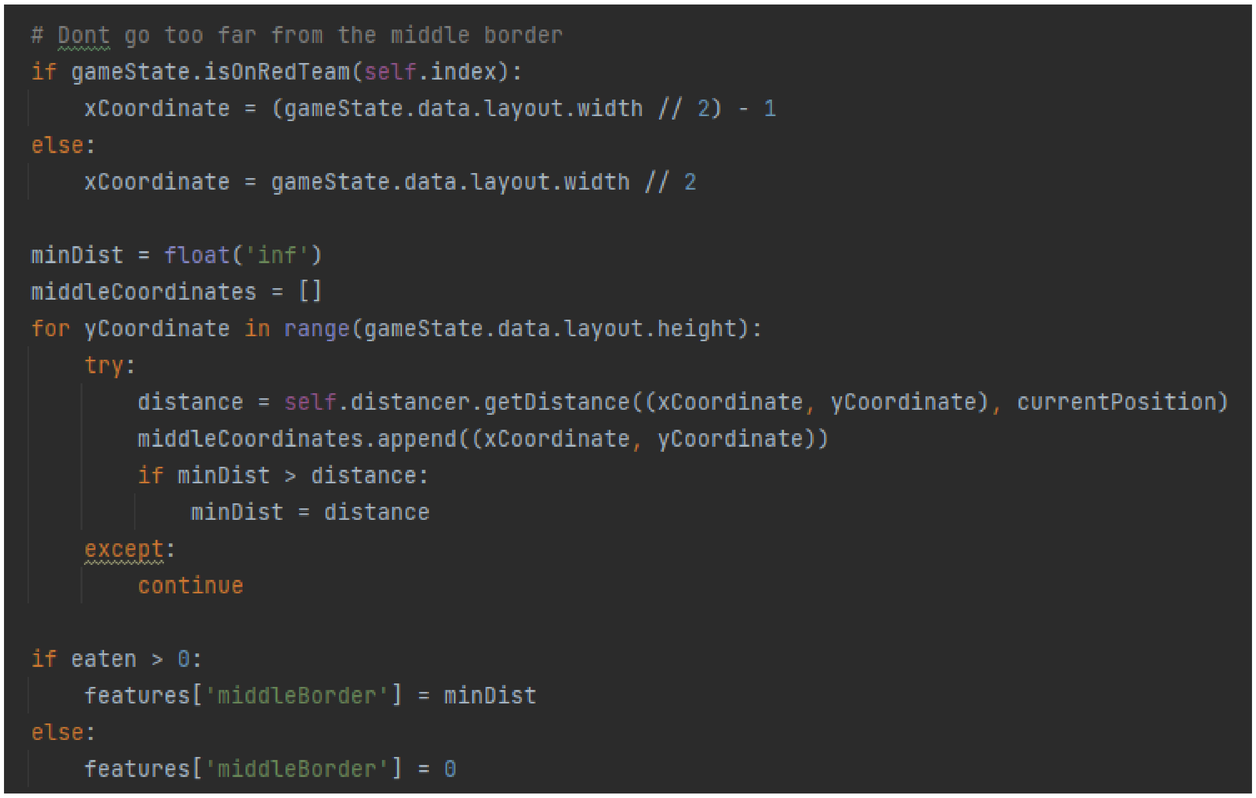


```
minEnemyDistance = 10
if len(enemyDistances) and min(enemyDistances) < 3 and successor.getAgentState(self.index).isPacman:
    minEnemyDistance = min(enemyDistances)
features['minEnemyDistance'] = minEnemyDistance
```

Figure 17. Code to get the distance from the closest enemy

This code just doesn't get the minimum distance from the enemy. I first initialized minEnemyDistance as 10 so that we only care about this feature if and only if there is an enemy that is closer than 10 blocks away from the agent. This is done so that the agent doesn't get scared of an enemy ghost even though it is across the map. Only when the enemy ghost is close is when the agent should start to run away as depicted in figure 14.

middleBorder



```python
# Dont go too far from the middle border
if gameState.isOnRedTeam(self.index):
    xCoordinate = (gameState.data.layout.width // 2) - 1
else:
    xCoordinate = gameState.data.layout.width // 2

minDist = float('inf')
middleCoordinates = []
for yCoordinate in range(gameState.data.layout.height):
    try:
        distance = self.distancer.getDistance((xCoordinate, yCoordinate), currentPosition)
        middleCoordinates.append((xCoordinate, yCoordinate))
        if minDist > distance:
            minDist = distance
    except:
        continue

if eaten > 0:
    features['middleBorder'] = minDist
else:
    features['middleBorder'] = 0
```

Figure 18. Code to get the distance to the closest valid middle border coordinate

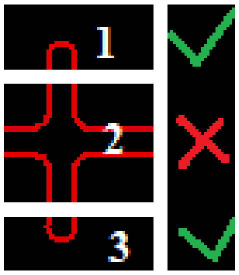
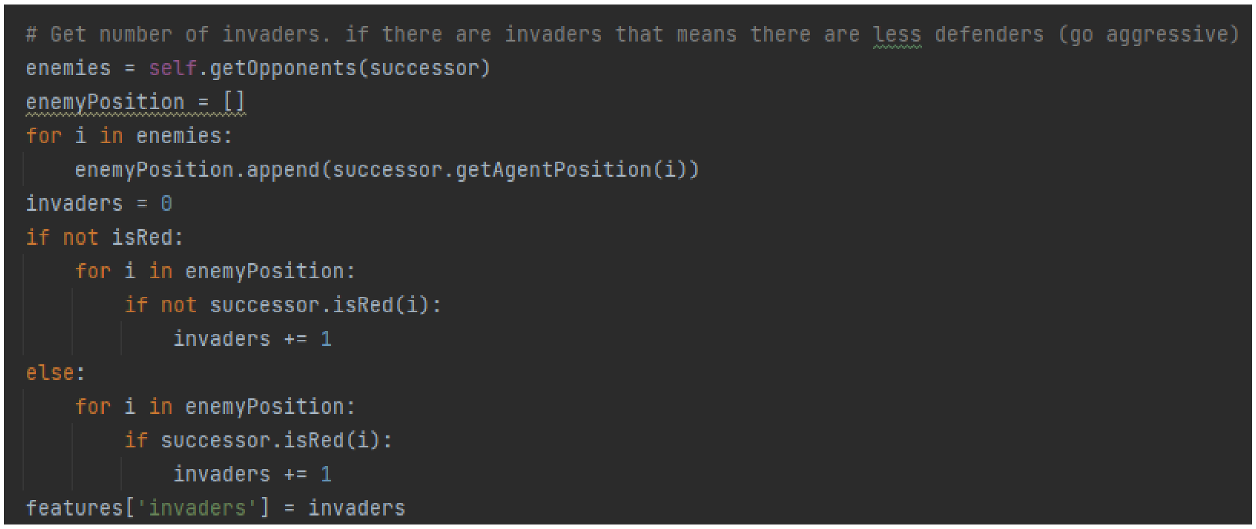


Figure 19. Example of valid and invalid coordinates at the middle border

This code was done so that the agent returned to my territory once it has eaten or is being chased. It gets the distance from each valid coordinate and gets the distance to the closest valid coordinate. The middle coordinates calculated by first accessing the layout data in gameState and dividing by 2. This gives us the X coordinates, however, to get the Y coordinates I had to do a different process. I looped through the height of the layout and got the distance from each coordinate and the agent's current position. The distance function does not work on coordinates that are not valid. For example, in figure TODO the sections 1 and 3 are valid because a Pacman can exit in there. However, in section 2 there is a wall that would block the Pacman and so the distance function would fail. I used a try except to catch is fail and to not save the coordinate if it is not valid. I only save the minimum distance if I have eaten a pellet otherwise the agent would keep returning to my territory and never eat any pellets. Once it eats a pellet the agent will try to get back to my territory as fast as possible.

Invaders

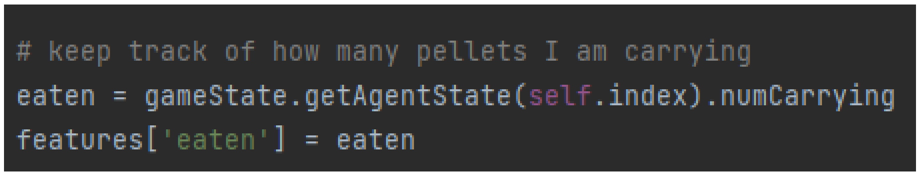


```python
# Get number of invaders. if there are invaders that means there are less defenders (go aggressive)
enemies = self.getOpponents(successor)
enemyPosition = []
for i in enemies:
    enemyPosition.append(successor.getAgentPosition(i))
invaders = 0
if not isRed:
    for i in enemyPosition:
        if not successor.isRed(i):
            invaders += 1
else:
    for i in enemyPosition:
        if successor.isRed(i):
            invaders += 1
features['invaders'] = invaders
```

Figure 20. Code to get the number of invaders

This feature point is used to make it such that the agent will go aggressive when there are enemy agents in my territory. This is because there would be less defenders in the enemy territory and as such my agent should prioritize going into the enemy territory more.

Eaten

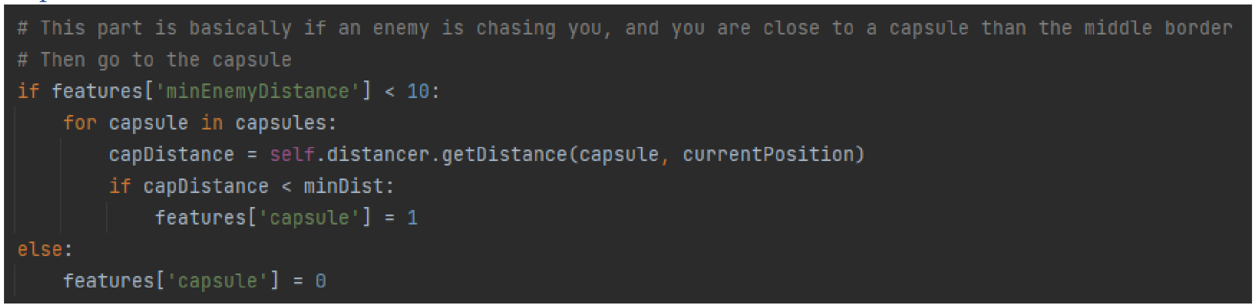


```python
# keep track of how many pellets I am carrying
eaten = gameState.getAgentState(self.index).numCarrying
features['eaten'] = eaten
```

Figure 21. Code to get the number of pellets I have eaten

This feature point is used to keep track of how many pellets I have currently eaten. This feature point discourages eating multiple pellets which is very risky.

Capsule

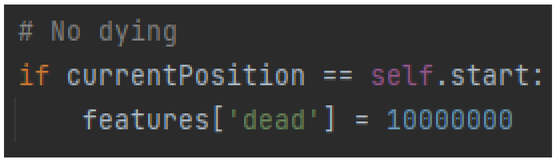


```python
# This part is basically if an enemy is chasing you, and you are close to a capsule than the middle border
# Then go to the capsule
if features['minEnemyDistance'] < 10:
    for capsule in capsules:
        capDistance = self.distancer.getDistance(capsule, currentPosition)
        if capDistance < minDist:
            features['capsule'] = 1
else:
    features['capsule'] = 0
```

Figure 22. Code to return if it is preferred going to a capsule if it is closer to the agent than the middle border

This feature point is used in scenarios where there is no way to get safely back home with out dying. In this case if the agent is getting chased and is closer to a capsule than the middle border it will prioritize getting the capsule than getting back to base.

Dead

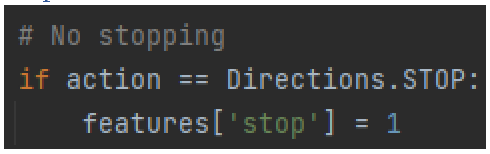


```
# No dying
if currentPosition == self.start:
    features['dead'] = 10000000
```

Figure 23. Code to make sure the agent doesn't die

This is very simple and returns a very large negative value if there is a successor that ends up with the agent dying. This means the agent should prioritize its survival over all else.

Stop

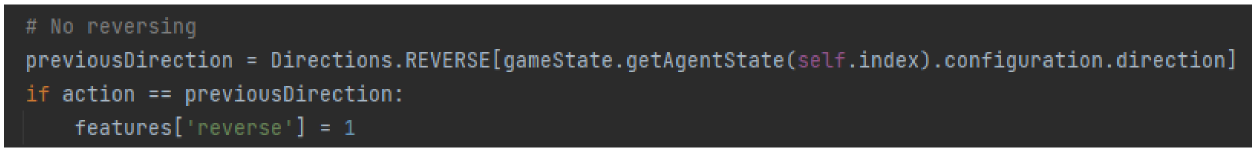


```
# No stopping
if action == Directions.STOP:
    features['stop'] = 1
```

Figure 24. Code to make sure the agent doesn't stop

Since this agent is offensive, it needs to always be on the move. Hence why if the current action is to stop then it is discouraged.

Reverse

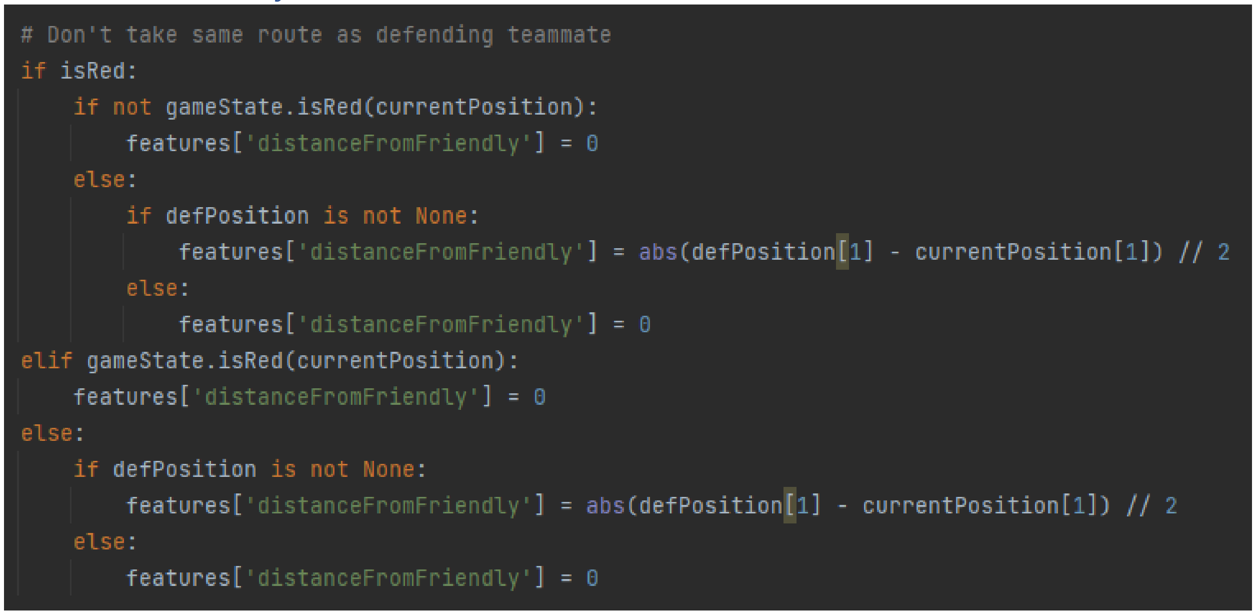


```
# No reversing
previousDirection = Directions.REVERSE[gameState.getAgentState(self.index).configuration.direction]
if action == previousDirection:
    features['reverse'] = 1
```

Figure 25. Code to make sure the agent doesn't reverse

This feature point discourages from reversing or doing the same actions over again. This is really only meant to be effective when it is getting chased so that it doesn't decrease the distance from itself and the enemy agent it is being chased by.

distanceFromFriendly



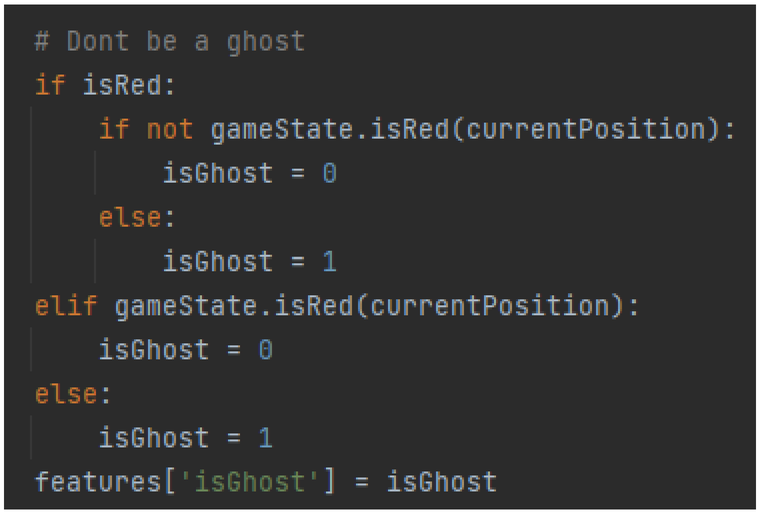
```
# Don't take same route as defending teammate
if isRed:
    if not gameState.isRed(currentPosition):
        features['distanceFromFriendly'] = 0
    else:
        if defPosition is not None:
            features['distanceFromFriendly'] = abs(defPosition[1] - currentPosition[1]) // 2
        else:
            features['distanceFromFriendly'] = 0
elif gameState.isRed(currentPosition):
    features['distanceFromFriendly'] = 0
else:
    if defPosition is not None:
        features['distanceFromFriendly'] = abs(defPosition[1] - currentPosition[1]) // 2
    else:
        features['distanceFromFriendly'] = 0
```

Figure 26. Code to make sure the agent doesn't stay very close to the defensive agent

This code is used to discourage the agent from taking the same route as the defending agent while it is a ghost. This has two reasons. The first is because since the defending agent is going towards the enemy that means there is an enemy where the defending is at. This was done because sometimes the offensive agent would get stuck since there was usually an enemy agent in front of it. Since it cannot go forward without dying it would just stay still. To reduce the chance of this happening we want to go away from the defensive agent.

The second reason is because it is better to come from multiple entrances rather than from one entrance. This is because there might be a scenario where both enemy agents are being blocked by the defensive agent and so by taking another route the offensive agent would have more freedom.

isGhost



```
# Dont be a ghost
if isRed:
    if not gameState.isRed(currentPosition):
        isGhost = 0
    else:
        isGhost = 1
elif gameState.isRed(currentPosition):
    isGhost = 0
else:
    isGhost = 1
features['isGhost'] = isGhost
```

Figure 27. Code to make sure the agent doesn't stay a ghost

This code encourages the agent to be a Pacman rather than a ghost. Hence why it is called offensive.

Observations

Good

My agents ended up in the 8th-9th in the class that had a total of 44 teams. This is a very good score.

I think what made my agents very good was that the defensive agent would always body block the enemy agents and aggressively chase any enemy Pacman. On the other hand, my offensive agent would often eat only one pellet and returning to base to increase the score. This strategy of eating only one pellet is very effective because eating multiple is very risky as it increases the chances of the agent dying and spreading the pellets deeper in the enemy's territory.

There is a scenario that can occur which is unique. If the defensive and offensive agents are together and there is an enemy agent on the other side of the border. Sometimes the offensive agent will bait the enemy agent into chasing it into my territory by going in and out of the enemy territory. This will cause the enemy agent to be eaten by the defensive agent if it gets into my territory for too long. This is probably happening because the enemy agents don't have any feature points or code like my defensive agent that discourage it from becoming a Pacman if it is a defensive oriented agent.

Bad

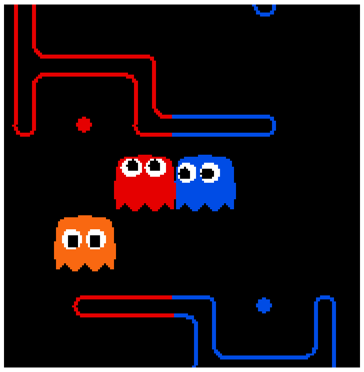


Figure 28. Offensive agent getting stuck

The code I have written is far from perfect as there are many bugs that need to be fixed. For example in figure 28 the offensive agent is stuck going in a circle. I tried to reduce the chances of this happending by adding the feature point distanceFromFriendly that discourages the offensive agent from getting too close to the defensive agent. However, there are still situations where this happens. Because I have code that discourages "stopping" it is instead going in circles which is a step in the right direction but not what I need.

I also don't think my feature point for capsule is correct or optimal as there are situations where the agent will prefer going towards food rather than the capsule even if it means death. This might be because I have kept the weight of the food too high such that the agent prefers going to it over the capsule.

Another issue that I’ve seen occur sometimes, but is quite rare is when the defensive agent is stuck body blocking the enemy agent but there is an invader eating the pellets. I am not sure exactly why this happens. Additionally only when the invader eats a capsule does the defensive agent start chasing the invader. This scenario occurs very rarely so it was hard to debug properly.

Recommendations and Conclusion

There are several ways my code can be improved upon. First it would be better if both agents were flexible in both defense and offense. This is so that if an agent isn’t doing anything or if there is a scenario where an offensive agent is close to an enemy Pacman that the agent can do both things at once. This could be solved by having each agent save a copy of the offensive and defensive classes and having an if else statement that chooses between the two given certain scenarios.

Overall, I did a good job in how my agents perform given that I have gotten pretty high in my class. However, there is a lot more things that can be improved. I did learn a lot from this project that I think will be beneficial for my future.