Project Report

Introduction

The Pac-Man Capture is a game where the goal is to steal all enemy pellets and bring them back to the side of friendly territory. There are rules to prevent and enhance the prevention of the friendly pellets being stolen and the capture of enemy pellets. To understand the full rules, please refer to the documentation regarding the rules on the course website.

My solution uses A-star search to select the next move. However, the underlying power in my solution is the way the goal is selected and the cost function of each successor. The goal and cost function leverage a lot of information of the game state such as food, pellets, enemies, etc. to ultimately give a score or ratio to each potential goal or successor. Then the goal is selected based on the best ratio and the successor is selected based on the least cost function. These will be explained in further detail below.

Theoretical Foundation

The algorithm that was selected was A-star search. The way it works is that it starts from a node. The node is put into a queue and the successors of that node are generated. Then for each successor, it will calculate the cost of g of the successor and the heuristic (h cost) and add them. This will be called the cost function. If the successor is the goal, then the algorithm will stop, otherwise, there should be two lists one for the open nodes that need to be explored and a closed list that tracks the nodes that were already explored. If the successor is in the open list and it has a higher cost function, then this successor will be skipped. If the successor is in the closed list and it has a higher cost function, then this successor will be skipped. Otherwise, the successor will be added to the open list. Finally, the parent node that generated the successors, that will be added to the closed list. The cycle will repeat with each node in the open list until the goal is found or the open list is empty.

The reason for the selection of A-star search is because I needed an algorithm that understood the cost of each potential successor. A-star search is considered to have "brains" in that it knows which successor is more costly from the beginning to the current successor and the current successor to the goal. I knew that I wanted to consider a lot of information and give it to A-star search such as food distance, enemy distance, exit distance, etc., and A-star search allowed for that to be tweaked easily without having to change the actual algorithm. This allowed me to focus more on tweaking the cost function and giving it a smarter "brain" than having to worry about the algorithm itself.

Another reason for the selection of the A-star search is because it is not constrained to a unidirectional search. This means it can backtrack and visit nodes that it has already visited. This is especially important because, in a maze where a Pac-Man might get stuck, it may need to go back from where it started. The disadvantage of the A-star search is that it requires a lot of memory and time to process the successors and queue. In a game where the maze may be small or may be large plus the time limit, execution time is important. A-star search algorithm is not very efficient. However, despite the limitations, I concluded the benefits outweighed the limitations.

Find Agent Description

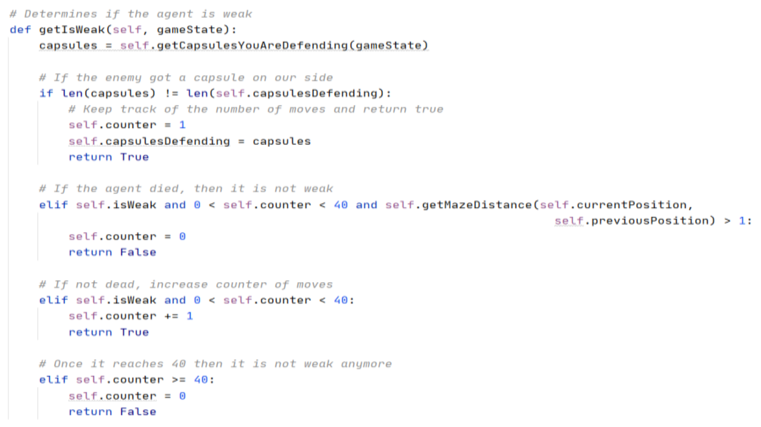
Description

The way the code is divided is by two agents. One offensive agent and another defensive agent. The reason for this is because both have different goals and should consider different information. One is to defend and the other is to take the pallets. Both of the agents have a base agent class that is used to get useful information that is needed. In addition, both agents have valuable information about the enemy based on the enemy base agent.

Friendly & Enemy Base Agent

I knew that in order to effectively choose a goal, the heuristic, and the g cost in the A-star search algorithm a lot of information will need to be gathered for the friendly agents and the enemy agents. For this reason, I created a friendly and enemy base agent classes to gather useful information to be later used in the offensive agent and defensive agent.

The most obvious information I needed to gather was to determine if the agents are weak either the friendly agents or the enemy agents. In addition, I needed to know if they are Pac-Man, again the friendly agents and the enemy agents. To determine if a Pac-Man is weak I had the agents keep track of their capsules defending. If at any time it had changed that meant one was captured by the enemy, thus making it weak. Then the counter started to keep track of the amount of time, but it also considered if it died. The code is shown in figure 1. To determine if the agent is a Pac-Man I simply check their location and compared it to the half of the maze.



```python
# Determines if the agent is weak
def getIsWeak(self, gameState):
    capsules = self.getCapsulesYouAreDefending(gameState)

    # If the enemy got a capsule on our side
    if len(capsules) != len(self.capsulesDefending):
        # Keep track of the number of moves and return true
        self.counter = 1
        self.capsulesDefending = capsules
        return True

    # If the agent died, then it is not weak
    elif self.isWeak and 0 < self.counter < 40 and self.getMazeDistance(self.currentPosition,
                                                                       self.previousPosition) > 1:
        self.counter = 0
        return False

    # If not dead, increase counter of moves
    elif self.isWeak and 0 < self.counter < 40:
        self.counter += 1
        return True

    # Once it reaches 40 then it is not weak anymore
    elif self.counter >= 40:
        self.counter = 0
        return False
```

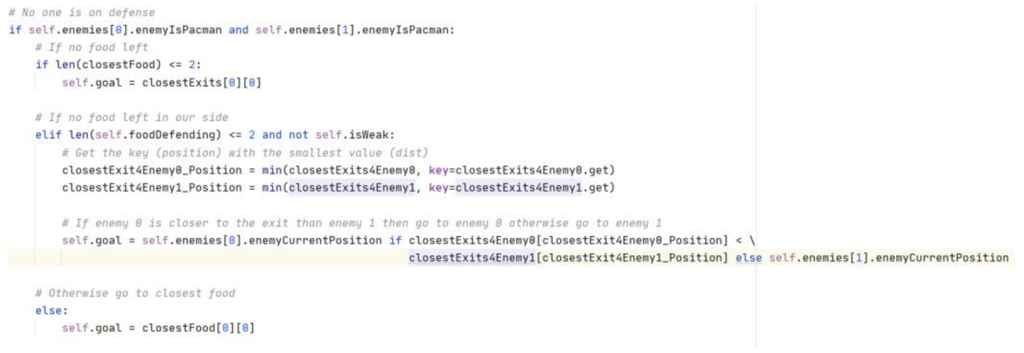
Figure 1: Is Pacman Weak

Other information that was gathered were the walls, current position, previous position, is weak, is Pac-Man, food attacking, food defending, capsules attacking, capsules defending, goal, food carrying, half maze, closest escape, max height, counter, escape routes for the friendly agent. The enemy agent had similar attributes as the friendly agent except for walls, max height, closest scape, and escape routes. The reason for the walls, food attacking, and food defending attributes is because the format the game state had, it was hard to use, the current and previous position was needed to check if the Pac-Man died, is weak and is Pac-Man was explained earlier, food carrying was needed to help determine the goal, half-maze was used to determine the border, closest escape was needed to find the easiest way out, and max-height was needed to find all the escape routes along the y axis. All of these attributes were updated once in each turn with each agent in one function.

This approach was definitely needed however the implementation has severe disadvantages. There is a lot of repetition and a lot of double information. For example, each agent, Offensive Agent and Defensive Agent have the same information of the enemy. However, because they do not communicate with each other, they need that. So, in total, we have 4 agents' information, two in the offensive agent and another in the defensive agent. This is a waste of computation time and memory.

Offensive Agent

The first task was to select a goal for the offensive agent. If both enemies are Pac-Man that means there is no one on defense, thus if we have collected all the food but two, return to the closest exit, if no one is on defense and there is no food on our side, then chase the enemy closest to the exit. Otherwise, go to the closest food and this can be seen in figure 2. If there is at least one enemy on defense, then we create a ratio. Based on the ratio it will select either the best food (may not be the closest food), the best capsule, or the best exit. The ratio considers how much food does the agent currently has, how close the enemy is, how close is the capsule, how close is the exit, and is the enemy. Based on that information it will choose the best goal. If both are on defense it will do the same and calculate a ratio based on the factors mentioned to get the optimal goal.



```python
# No one is on defense
if self.enemies[0].enemyIsPacman and self.enemies[1].enemyIsPacman:
    # If no food left
    if len(closestFood) <= 2:
        self.goal = closestExits[0][0]

    # If no food left in our side
    elif len(self.foodDefending) <= 2 and not self.isWeak:
        # Get the key (position) with the smallest value (dist)
        closestExit4Enemy0_Position = min(closestExits4Enemy0, key=closestExits4Enemy0.get)
        closestExit4Enemy1_Position = min(closestExits4Enemy1, key=closestExits4Enemy1.get)

        # If enemy 0 is closer to the exit than enemy 1 then go to enemy 0 otherwise go to enemy 1
        self.goal = self.enemies[0].enemyCurrentPosition if closestExits4Enemy0[closestExit4Enemy0_Position] < \
                                                        closestExits4Enemy1[closestExit4Enemy1_Position] else self.enemies[1].enemyCurrentPosition

    # Otherwise go to closest food
    else:
        self.goal = closestFood[0][0]
```

Figure 2: Choosing a goal - Offensive Agent

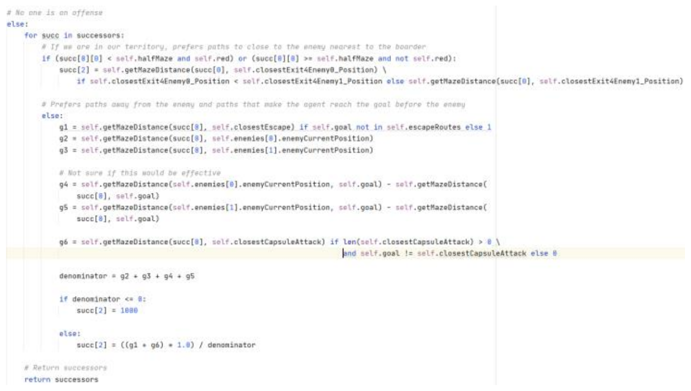
The next task was to create a g function that was attached to each successor. If no one was on defense and the agent was on the friendly side, then it preferred to pass by the enemies if it is not weak. If it is weak, then it prefers to go away from the enemy. Otherwise, it has a constant g function. If at least one enemy is on defense, it prefers passing through the enemy on friendly territory, if not weak, otherwise away from the enemy on friendly territory, and on the enemy side, it prefers path away from the enemy, close to the exit, and close to the capsule. If both are in defense, then it has a constant g cost in friendly territory and it prefers a path away from the enemy, close to the exit, and close to a capsule. The heuristic is just the maze distance between the successor and the goal.

Then the A-star search algorithm is applied, and a move is chosen. The disadvantage of this approach is that the offensive agent does not prefer food or goals that do not lead to dead ends. The agent might choose a goal because it is close, and the enemy is chasing it, but the goal might be at a dead-end and it will eventually die. That is the disadvantage of it.

Defensive Agent

The first task is to select the goal. If both enemies are attacking and the defensive agent is not weak and we have very little food left, then chase the enemy closest to the exit, otherwise chase the closest enemy. If we have one enemy attacking and the defensive agent is not weak then chase the enemy. Otherwise, if no one is on offense or Pac-Man is weak, then take the offensive and calculate a ratio for each exit, food, and capsule and determine which shall be the best goal for that agent.

The next step was to create a g function. If both are on offense and the defensive agent is not weak, then if we have no food left the g function is constant, otherwise, we prefer paths closer to the enemies exit and closer to the capsules. If one enemy agent is attacking, then prefer paths closer to the enemy exit and closer to the capsules. If the defensive agent is weak and enemies are attacking, and the agent is in the friendly territory then it will prefer paths away from the enemies. Finally, if no one is on the offense, if the defensive agent is in friendly territory, it will prefer paths that minimize the distance to the closest enemy just in case it crosses, and once it goes into enemy territory, it prefers a path away from the enemies, closer to an escape, and closer to a capsule as shown in figure 3. The heuristic is just the maze distance between the successor and the goal.



```
# No one is on offense
else:
    for succ in successors:
        # If we are in our territory, prefers paths to close to the enemy nearest to the boarder
        if (succ[0][0] < self.halfMaze and self.red) or (succ[0][0] >= self.halfMaze and not self.red):
            succ[2] = self.getMazeDistance(succ[0], self.closestExit4Enemy0_Position) \
                if self.closestExit4Enemy0_Position < self.closestExit4Enemy1_Position else self.getMazeDistance(succ[0], self.closestExit4Enemy1_Position)

        # Prefers paths away from the enemy and paths that make the agent reach the goal before the enemy
        else:
            g1 = self.getMazeDistance(succ[0], self.closestEscape) if self.goal not in self.escapeRoutes else 1
            g2 = self.getMazeDistance(succ[0], self.enemies[0].enemyCurrentPosition)
            g3 = self.getMazeDistance(succ[0], self.enemies[1].enemyCurrentPosition)

            # Not sure if this would be effective
            g4 = self.getMazeDistance(self.enemies[0].enemyCurrentPosition, self.goal) - self.getMazeDistance(
                succ[0], self.goal)
            g5 = self.getMazeDistance(self.enemies[1].enemyCurrentPosition, self.goal) - self.getMazeDistance(
                succ[0], self.goal)

            g6 = self.getMazeDistance(succ[0], self.closestCapsuleAttack) if len(self.closestCapsuleAttack) > 0 \
                                                                        and self.goal != self.closestCapsuleAttack else 0

            denominator = g2 + g3 + g4 + g5

            if denominator <= 0:
                succ[2] = 1000

            else:
                succ[2] = ((g1 + g6) + 1.0) / denominator

    # Return successors
    return successors
```

Figure 3: G Function - Defensive Agent

Finally, A-star search is applied, and a move is chosen. The only disadvantage of this approach is that the defensive agent may go for a far goal in enemy territory. This may present a problem because if an enemy is getting friendly pallets and the defensive agent is far then it will take a while for it to get back possibly giving crucial time to the enemy. In addition, when it goes back to chase the enemy, it does not consider the enemy may still kill it. Even though it is acting as a “defensive agent” it still needs to be careful and it does not account for it.

Strengths and Weakness

The strengths about these agents are that it doesn't choose one goal but rather it has multiple goals that may be selected. Each goal allows for the game to be won. In addition, the selection of goals considers a lot of information and ultimately tries to choose the best goal. In the cost function, it too considers a lot of information to prefer optimal paths.

The weakness of these agents is that is that there is a lot of repetition wasting time and memory. In addition, there is also crucial information missing. For example, the offensive agent does not consider dead ends and the defensive agent doesn't take into account the optimal path back to the enemy after it's been on attack mode. However, besides the weakness, the agents still perform extremely well.

Observations

According to my observations, my agents did extremely well. In the preliminary tournament, my agents took first place with not a single defeat. However, the major observation that I have made is two things. On the offensive side, my agent does not consider dead ends. This is very important because these are crucial moves wasted allowing for the enemy to catch up. In the defensive agent, when it is in the enemy territory trying to go back and chase an enemy on the friendly territory its only focus is trying to reach the goal, but it does not choose the path to avoid the enemy on the enemy side and again if it dies, it wastes crucial moves that allow for the enemy to get more food. This happens because I did not account for it. Besides that, I believe my agents do everything else fairly well.

Recommendations & Conclusions

In the end, my agents used A-star search for its ability to choose the optimal path. A lot of information was gathered on the friendly agents and the enemy agents. This information was used to choose the best goal for each defensive agent and offensive agent. Using the goal and other information will then give a g cost to each successor and add the heuristic to it. This allowed for the optimal move to be selected.

The best way to improve this program is to resolve the issues described in the observations. The offensive agent would have to consider the dead-end goals and prefer goals that are not in dead-ends. The defensive agent has to try to prefer goals near the border and when it tries to travel to the friendly side to consider the enemy on the enemy side, so it will not die. I believe these are some areas my problem needs improvement.