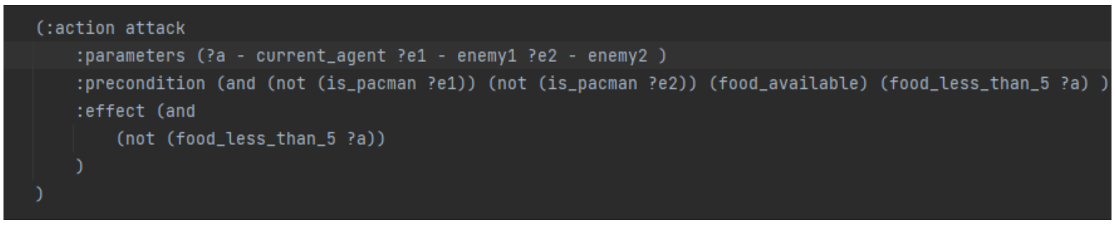
PART 1 Modifications of PDDL definition

For the modification part of the upper-level planner, 1. For the original four actions, some preconditions and effects are modified. 2. An additional go_home2 action is added to handle some abnormal situations.

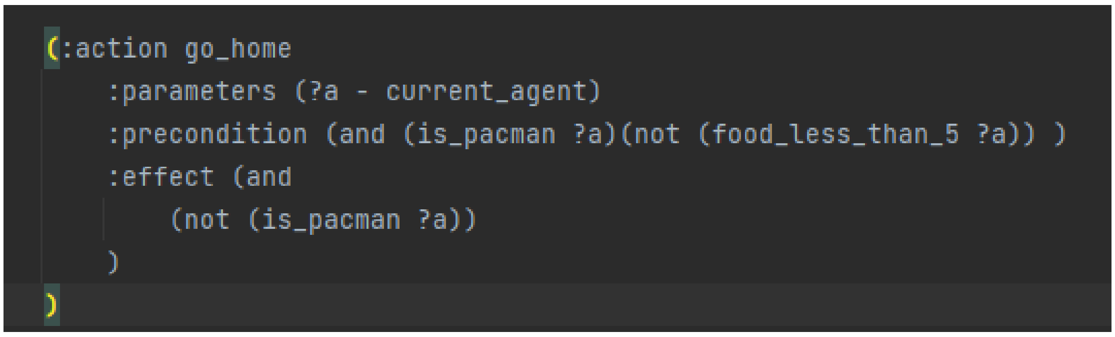
Modification to the action attack is :



```
(:action attack
    :parameters (?a - current_agent ?e1 - enemy1 ?e2 - enemy2 )
    :precondition (and (not (is_pacman ?e1)) (not (is_pacman ?e2)) (food_available) (food_less_than_5 ?a) )
    :effect (and
        (not (food_less_than_5 ?a))
    )
)
```

This means that as long as there are 5 fruits in the backpack, the attack state will end.

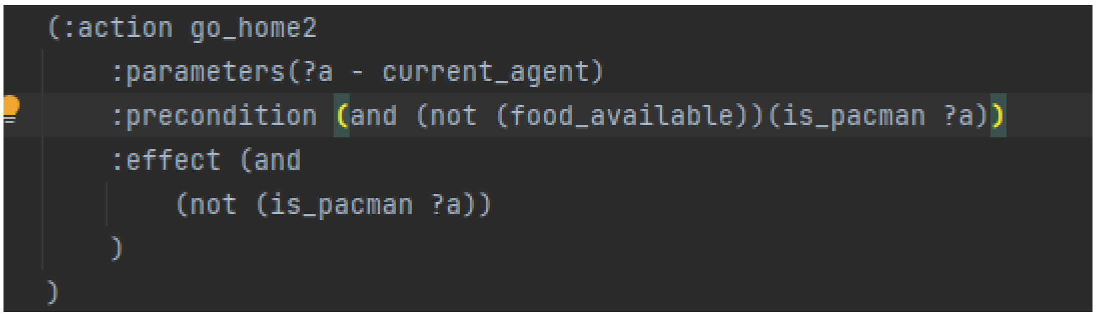
Modification to the action go_home is:



```
(:action go_home
    :parameters (?a - current_agent)
    :precondition (and (is_pacman ?a)(not (food_less_than_5 ?a)) )
    :effect (and
        (not (is_pacman ?a))
    )
)
```

The change here is to connect with the above definition of attack action.

Addition of the action of go_home2:



```
(:action go_home2
    :parameters(?a - current_agent)
    :precondition (and (not (food_available))(is_pacman ?a))
    :effect (and
        (not (is_pacman ?a))
    )
)
```

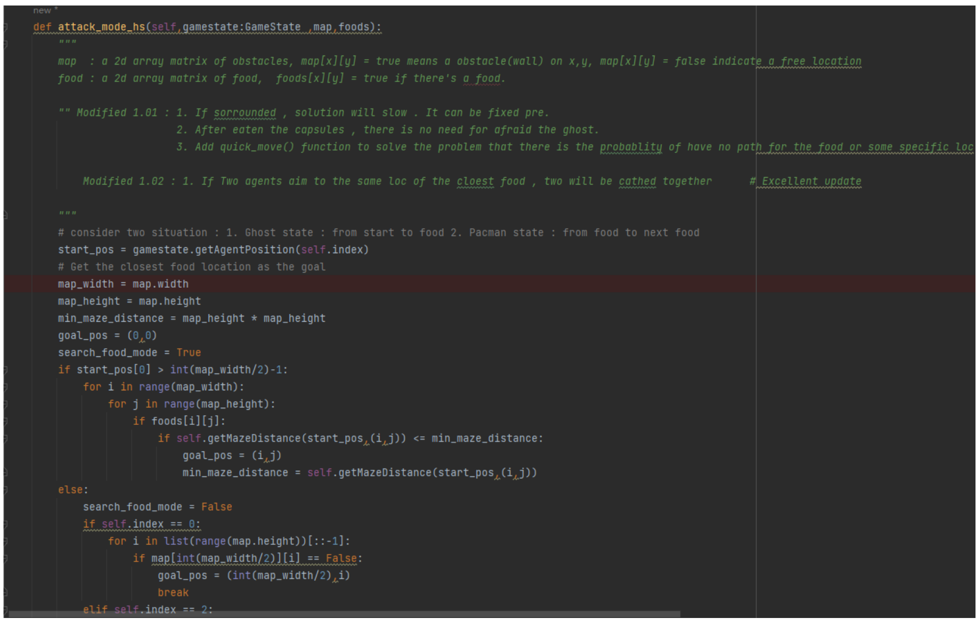
The increase here is to deal with the situation when the fruit is eaten up (optimistic strategy). One of the conditions in the precondition is inconsistent with the go_home action, and the rest are the same.

In addition, due to the changes of these PDDL actions, it is necessary to change the function goalScoring() in the visible file for the settings of positive goal and negative goal in myTeam.py.

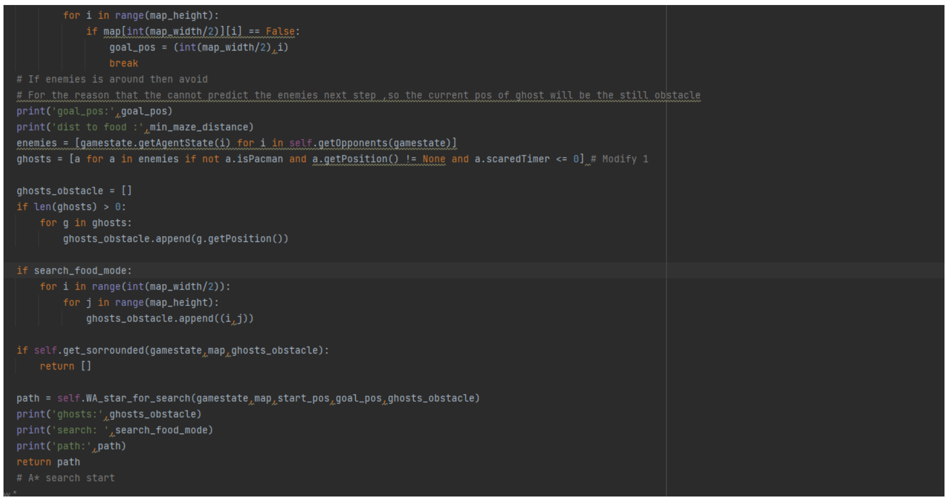
PART 2 Settings of the lower planner

1. ATTACK MODE

   This function accepts a parameter list consisting of the current game state, obstacle map, and food location map (hereinafter referred to as gamestate, map, food). In terms of policy setting, two situations are considered. First, when moving in one’s own area, the two agents decide to move to the upper and lower edges of the junction of the two areas according to their own index to enter the opponent’s area (the purpose of this setting is to prevent The opponent's homogenization strategy makes the opponent's two agents closer, and can only follow one of our agents when defending). Then, when the agent enters the enemy area, set the goal pos to the food closest to itself. In these two processes, the pathfinding algorithm used is the A star algorithm. In addition, the additional obstacle settings can be known from the attack_mode_hs() function (including the judgment of the opponent's position and the choice of some own areas).



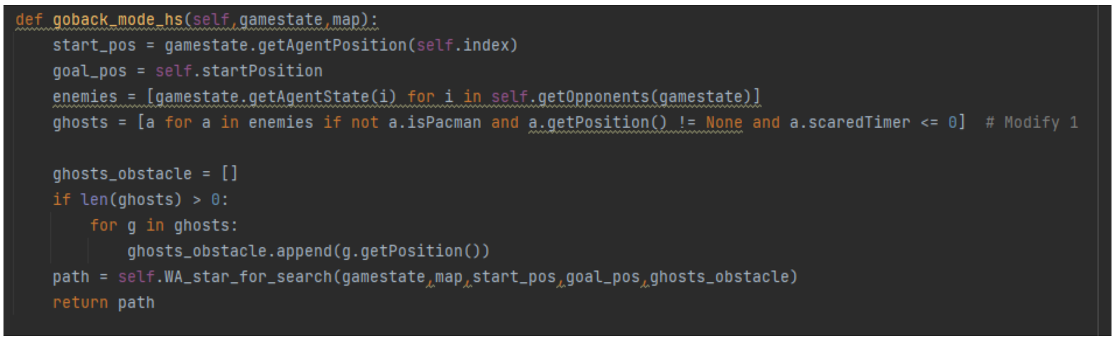
```
new *
def attack_mode_hs(self,gamestate:GameState ,map,foods):
    """
    map  : a 2d array matrix of obstacles, map[x][y] = true means a obstacle(wall) on x,y, map[x][y] = false indicate a free location
    food : a 2d array matrix of food,  foods[x][y] = true if there's a food.

    "" Modified 1.01 : 1. If sorrounded , solution will slow . It can be fixed pre.
                       2. After eaten the capsules , there is no need for afraid the ghost.
                       3. Add quick_move() function to solve the problem that there is the probablity of have no path for the food or some specific loc

       Modified 1.02 : 1. If Two agents aim to the same loc of the cloest food , two will be cathed together      # Excellent update

    """
    # consider two situation : 1. Ghost state : from start to food 2. Pacman state : from food to next food
    start_pos = gamestate.getAgentPosition(self.index)
    # Get the closest food location as the goal
    map_width = map.width
    map_height = map.height
    min_maze_distance = map_height * map_height
    goal_pos = (0,0)
    search_food_mode = True
    if start_pos[0] > int(map_width/2)-1:
        for i in range(map_width):
            for j in range(map_height):
                if foods[i][j]:
                    if self.getMazeDistance(start_pos,(i,j)) <= min_maze_distance:
                        goal_pos = (i,j)
                        min_maze_distance = self.getMazeDistance(start_pos,(i,j))
    else:
        search_food_mode = False
        if self.index == 0:
            for i in list(range(map.height))[::-1]:
                if map[int(map_width/2)][i] == False:
                    goal_pos = (int(map_width/2),i)
                    break
        elif self.index == 2:
```



```
        for i in range(map_height):
            if map[int(map_width/2)][i] == False:
                goal_pos = (int(map_width/2),i)
                break
# If enemies is around then avoid
# For the reason that the cannot predict the enemies next step ,so the current pos of ghost will be the still obstacle
print('goal_pos:',goal_pos)
print('dist to food :',min_maze_distance)
enemies = [gamestate.getAgentState(i) for i in self.getOpponents(gamestate)]
ghosts = [a for a in enemies if not a.isPacman and a.getPosition() != None and a.scaredTimer <= 0] # Modify 1

ghosts_obstacle = []
if len(ghosts) > 0:
    for g in ghosts:
        ghosts_obstacle.append(g.getPosition())

if search_food_mode:
    for i in range(int(map_width/2)):
        for j in range(map_height):
            ghosts_obstacle.append((i,j))

if self.get_sorrounded(gamestate,map,ghosts_obstacle):
    return []

path = self.WA_star_for_search(gamestate,map,start_pos,goal_pos,ghosts_obstacle)
print('ghosts:',ghosts_obstacle)
print('search: ',search_food_mode)
print('path:',path)
return path
# A* search start
```

2. GO_HOME MODE

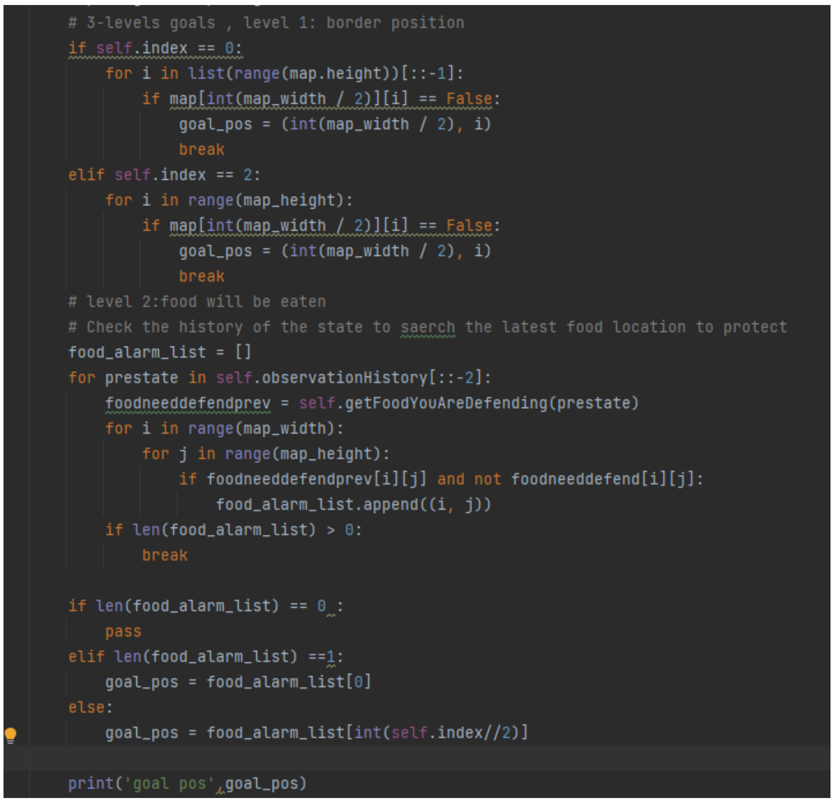
   This function accepts a parameter list composed of gamestate and map. The function is relatively simple, that is, to return to this area, so directly set the target point as the initial birth point, and avoid enemies that are not in a state of fear on the way back.



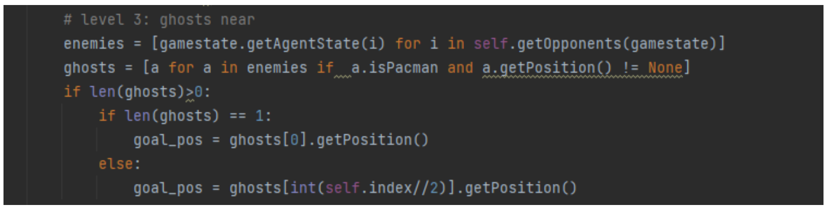
```
def goback_mode_hs(self,gamestate,map):
    start_pos = gamestate.getAgentPosition(self.index)
    goal_pos = self.startPosition
    enemies = [gamestate.getAgentState(i) for i in self.getOpponents(gamestate)]
    ghosts = [a for a in enemies if not a.isPacman and a.getPosition() != None and a.scaredTimer <= 0]  # Modify 1

    ghosts_obstacle = []
    if len(ghosts) > 0:
        for g in ghosts:
            ghosts_obstacle.append(g.getPosition())
    path = self.WA_star_for_search(gamestate,map,start_pos,goal_pos,ghosts_obstacle)
    return path
```

3. DEFENCE MODE

   This function accepts a parameter list consisting of gamestate, map, and an array of foodneeddefence that records the location of your own food at that moment. Under this strategy, there are three priority targets, and the lowest-level target is the area border. Because of the lack of information about the enemy's location, it is necessary to guard the edge area (in this case, the location of the two agents It is not an overlapping edge area, but scattered); the target of the suboptimal level is the food position that has just disappeared, and this position is determined by combining the foodneeddefence state at the time and the foodneeddefence state in the historical gamestate; the target of the optimal level is After sensing the enemy target, it will chase the target.



```
    # 3-levels goals , level 1: border position
    if self.index == 0:
        for i in list(range(map.height))[::-1]:
            if map[int(map_width / 2)][i] == False:
                goal_pos = (int(map_width / 2), i)
                break
    elif self.index == 2:
        for i in range(map_height):
            if map[int(map_width / 2)][i] == False:
                goal_pos = (int(map_width / 2), i)
                break
    # level 2:food will be eaten
    # Check the history of the state to saerch the latest food location to protect
    food_alarm_list = []
    for prestate in self.observationHistory[::-2]:
        foodneeddefendprev = self.getFoodYouAreDefending(prestate)
        for i in range(map_width):
            for j in range(map_height):
                if foodneeddefendprev[i][j] and not foodneeddefend[i][j]:
                    food_alarm_list.append((i, j))
        if len(food_alarm_list) > 0:
            break

    if len(food_alarm_list) == 0 :
        pass
    elif len(food_alarm_list) ==1:
        goal_pos = food_alarm_list[0]
    else:
        goal_pos = food_alarm_list[int(self.index//2)]

    print('goal pos',goal_pos)
```



```
# level 3: ghosts near
enemies = [gamestate.getAgentState(i) for i in self.getOpponents(gamestate)]
ghosts = [a for a in enemies if  a.isPacman and a.getPosition() != None]
if len(ghosts)>0:
    if len(ghosts) == 1:
        goal_pos = ghosts[0].getPosition()
    else:
        goal_pos = ghosts[int(self.index//2)].getPosition()
```

Based on the above, these strategies are all heuristic methods, and the Approximately Q-learning method is not used, because a better reward function setting has not been found to make the weight matrix converge.

PART 3.Performence of the model

Average Score: 5.448979591836735
Scores: 9, 10, 6, 6, 9, 2, 2, 6, 6, 6, 9, 2, 6, 6, 9, 2, 2, 2, 6, 9, 6, 6, 6, 9, 2, 2, 2, 2, 6, 6, 9, 2, 2, 6, 9, 2, 6, 6, 6, 9, 2, 6, 9, 6, 9, 2, 2, 6, 6
Red Win Rate: 49/49 (1.00)
Blue Win Rate: 0/49 (0.00)
Record: Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Red, Re
d, Red, Red, Red, Red, Red

RANDOM1
Average Score: 17.571428571428573
Scores: 13, 13, 21, 21, 21, 21, 13
Red Win Rate: 7/7 (1.00)
Blue Win Rate: 0/7 (0.00)
Record: Red, Red, Red, Red, Red, Red, Red

RANDOM2
Average Score: 13.285714285714286
Scores: 9, 13, 15, 13, 13, 15, 15
Red Win Rate: 7/7 (1.00)
Blue Win Rate: 0/7 (0.00)
Record: Red, Red, Red, Red, Red, Red, Red

RANDOM3
Average Score: 16.714285714285715
Scores: 19, 5, 30, 11, 11, 11, 30
Red Win Rate: 7/7 (1.00)

```
Blue Win Rate: 0/7 (0.00)
Record:          Red, Red, Red, Red, Red, Red, Red

RANDOM4
Average Score: 16.714285714285715
Scores:          16, 16, 16, 16, 16, 21, 16
Red Win Rate:    7/7 (1.00)
Blue Win Rate: 0/7 (0.00)
Record:          Red, Red, Red, Red, Red, Red, Red

RANDOM 5
Scores:          14, 29, -2, 29, -2, 29, 29
Red Win Rate:    5/7 (0.71)
Blue Win Rate: 2/7 (0.29)
Record:          Red, Red, Blue, Red, Blue, Red, Red

RANDOM 6
Average Score: 2.142857142857143
Scores:          -2, 27, -2, -2, -2, -2, -2
Red Win Rate:    1/7 (0.14)
Blue Win Rate: 6/7 (0.86)
Record:          Blue, Red, Blue, Blue, Blue, Blue, Blue

RANDOM7
Average Score: 3.142857142857143
Scores:          3, 9, -1, 2, 3, 3, 3
Red Win Rate:    6/7 (0.86)
Blue Win Rate: 1/7 (0.14)
Record:          Red, Red, Blue, Red, Red, Red, Red
```

Wining rate(default):100%
Wining rate(random):40/49

According to observations, the reason for the failure of some situations is that after the blue team gained an advantage, our side continued to be in a stalemate with the opponent in some states, so that we could not take offensive ways to score points. This is due to the lack of randomness in the algorithm.