Introduction to Artificial Intelligence B Contest Report

Chantal van Duin (s1004516) and Nayeong Kim (s1006313)

April 2018

1 Choice of Agent

For our agent we tried multiple things. We started of with a simple reflex agent which uses a slightly updated version of our better evaluation function of assignment 4. However, since this did not result in pacman choosing the best move every single time, we wanted to try another kind of agent.

We then thought of using a multiple alpha beta agent would be the best option. Since it abandons moves resulting in worse scores compared to the previously calculated moves and prunes mini-max branches which will not be able to influence which move pacman is going to make next. The stated above properties of a multiple alpha beta make it faster than an A-star agent, making it more suitable to use for the contest.

In addition, we wanted to use a multiple alpha beta agent instead of a normal alpha beta agent as there are multiple ghosts that pacman needs to take in consideration in the contest. However, while we observed that the use of the multiple alpha beta agent in combination with a simplified evaluation function did result in higher scores, it also took too much computation time to return a move to qualify for the contest.

In the end we choose to use a simple reflex agent in combination with an elaborate evaluate function, resulting in better scores than our first simple reflex agent but with less computation time than our multiple alpha beta agent.

```
def move(self, gstate: gamestate.Gamestate) -> util.Move:
    moves = gstate.legal_moves_vector(gstate.agents[self.id])
    scores = {move: self.evaluate(gstate.copy, move) for move in moves}
    max_score = max(scores.values())
    max_moves = [move for move in moves if scores[move] == max_score]
    return random.choice(max_moves)
```

The above-mentioned code is the code that agent uses to decide what move it is going to make based on current game state of pacman. The agent does this by evaluating the score for each move that pacman can make - by calling the evaluation function for each move -, putting those scores into a dictionary

and selecting the move that gives the highest score of all scores put into the dictionary.

2 Evaluation Function

By cause of using a simple reflex agent, a more extensive evaluation function is needed. We want pacman to be able to do the below-mentioned actions while taking below-mentioned things in consideration:

- be able to take the move that results in winning the game and avoiding the move that results in losing the game
- eating entire food supply in level in the least amount of moves possible
- collecting pellets along the way and eating nearby scared ghosts when a pellet has been collected by pacman
- avoiding ghosts when they come too close to pacman
- avoid getting stuck in one place of level where there are a lot of ghosts in the surroundings of the small remaining amount of food

2.1 Win Or Lose

Pacman wins the game when all the food in level is gone and loses the game when it gets eaten by a ghost. We want to give the biggest value possible when a move results in winning the game and the lowest value possible when it gets eaten by a ghost, this is done by below stated code:

```
if gstate.win:
    return float('inf')
elif gstate.loss:
    return float('-inf')
```

2.2 Eating Food Supply

We want to stimulate pacman in eating the food supply in the least amount of moves possible. We do this in two ways :

- By increasing the score when pacman has eaten one dot of food. We can increase the score by creating a minus score for each dot of food that is still left in the level. By eating a dot of food the total minus score of all the food dots decreasing resulting in a higher score, stimulating pacman to eat the food dots faster since it results in a higher score and thus a faster victory.

```
dotsNotEaten = 0
dotList = []

for dot in gstate.dots.list():
    dotList.append((dot, dotDist))


for dotPos, distance in dotList:
    dotsNotEaten -= 800

return score + dotsNotEaten + pelletsNotEaten
```

- By increasing the score when pacman gets closer to a food dot. If score increases for when pacman gets closer to dots of food and when it is on food dot then pacman is encouraged to get closer to the food dots which results in the dots of food getting eaten faster.

```
dotsPresent = 0

 for dot in gstate.dots.list():
    dotDist = self.distancer.get_distance(gstate.pacman, dot)
    dotList.append((dot, dotDist))

for dotPos, distance in dotList:
    score += (1 / distance) * 600
    if gstate.pacman == dotPos:
        score += 300

return score + dotsNotEaten + pelletsNotEaten
```

2.3 Pellet Collecting And Eating Ghosts

Pacman gets higher scores when it collects pellets and use the pellets to eat scared ghosts with. So to increase the chance that pacman will eat a pellet, we increase score when it has eaten a pellet and when it gets close to or is on a pellet. We do this in similar fashion as increasing the score for when pacman eats food dots. Since we want that pacman prioritizes eating dots of food more than collecting pellets, we give the collecting of pellets a lesser score than the eating of food dots.

```
pelletsNotEaten = 0
pelletList = []

for pellet in gstate.pellets.list():
```

```
    pelletList.append((pellet, self.distancer.get_distance(gstate.pacman, pellet)))

for pelletPos, pelletDist in pelletList:
    score += (1 / pelletDist) * 180
    pelletsNotEaten -= 150
    if gstate.pacman == pelletPos:
         score += 600
```

Since we want pacman to go after ghosts when they are scared as a result of collecting a pellet, we need to increase the score when pacman gets close to a scared ghost. This is done by using the timer in gstate.timers with the timer indicating how much time the ghost is still scared for pacman. Because we do not want pacman to still be too close when the timer runs out, we only want to chase ghosts when there is enough time on the timer (timer ¿= 4). We chase ghosts by increasing the score when it eats ghosts, when pacman is close enough to eat a ghost before time is up and when it is really close to the ghosts.

```
ghostList = []

  for ghost in gstate.ghosts:
        ghostDist = self.distancer.get_distance(gstate.pacman, ghost)
        ghostList.append(ghostDist)
        for timer in gstate.timers:
        if timer >= 4:
            if ghostDist == 0:
                score += 6000
            elif ghostDist < 2:
                score += 300
            elif ghostDist < timer:
                score += (1 / ghostDist) * 300
        elif timer <= 4 and ghostDist != 0:

return score + dotsNotEaten + pelletsNotEaten
```

2.4 Avoiding Ghosts

We want pacman to avoid ghosts when they are at a close enough distance that the chance is big that they can eat pacman.

```
ghostList = []

for ghost in gstate.ghosts:
    ghostDist = self.distancer.get_distance(gstate.pacman, ghost)
    ghostList.append(ghostDist)
        for timer in gstate.timers:
            if timer >= 4:
                    # code for chasing scared ghosts, see section 2.3
```

```
            elif timer <= 4 and ghostDist != 0:
                if dotsPresent < self.smallFood: #code for avoiding stuck
                    in one place,see section 2.4
                else:
                    if ghostDist < 3:
                        score -= (1 / ghostDist) * 2000
                    else:
                        score -= (1 / ghostDist) * 400

 return score + dotsNotEaten + pelletsNotEaten
```

2.5 Avoid Stuck In One Place

For the reason that pacman gets a higher or lower score when it gets close to food dots or ghosts, it sometimes gets stuck in one place. This has to with the fact that there is is a small amount of food left in the level and that the ghosts are surrounding that small food supply. Pacman wants to move closer to get food but is too scared of ghosts to do so.

To solve this we make pacman less scared of ghosts when there is less amount of food in the level. We do this by calculating in the beginning of level how much food there is and what is considered a small amount of food in that level. This we calculate in the prepare function since we need to calculate it when starting a level.

```
def prepare(self, gstate):
    self.dotsList = gstate.dots.list()
    self.smallFood = 1/20 * len(self.dotsList)
```

In the evaluation function we look how much food there is left for each move and if the amount of food in level reaches the small amount of food calculated in beginning of level then we increase the reward -higher score - when pacman eats a food dot and make pacman less afraid of ghost by decreasing the punishment -lower score - when it gets closer to ghosts. In addition we decrease the score when pacman decides to stop moving.

```
dotsPresent = 0
ghostList = []

for dot in gstate.dots.list():
    dotsPresent += 1

 for dotPos, distance in dotList:
    if dotsPresent < self.smallFood:
        score += 1000
    if move is move.stop:
        score -= 700
```

```
for ghost in gstate.ghosts:
    ghostDist = self.distancer.get_distance(gstate.pacman, ghost)
    ghostList.append(ghostDist)
        for timer in gstate.timers:
            if timer >= 4:
                    # code for chasing scared ghosts, see section 2.3
            elif timer <= 4 and ghostDist != 0:
                if dotsPresent < self.smallFood:
                    if ghostDist < 3:
                        score -= (1 / ghostDist) * 1000
                    else:
                        score -= (1 / ghostDist) * 100
                else:
                # code for avoiding ghost, see section 2.4

 return score + dotsNotEaten + pelletsNotEaten
```

2.6 Additional Code

To be able to calculate distance between pacman and the ghosts, dots and pellets with the

```
self.distancer_get.distance,
```

we need to initialize the distancer in the prepare function. Furthermore to be able to use gstate in the contestAgent class we need to use super().prepare(gstate) in the prepare function and to be able to use move we need to use

```
gstate.apply_move(0,move)
```

in the evaluate function.

```
def prepare(self, gstate):
    super().prepare(gstate)
    self.distancer = distancer.Distancer(gstate)
    self.distancer.precompute_distances()

def evaluate(self, gstate: gamestate.Gamestate, move: util.Move):
     gstate.apply_move(0, move)
```

3 Getting Closed In By Ghosts

We wanted to make a function that reduced the risk that pacman would not get closed in by two ghost and thus gets eaten. Although we wrote a function to do so, we did not see it back in the results. We wanted to include it our report even though it may not have worked the way it was supposed to do,it is however not in our handed in code.

We used a boolean function closedIn which was called for every move in the evaluation function. If closedIn equaled true then we would return an extremely low value in evaluation function so that pacman would not choose that move and as a consequence pacman would avoid getting closed in by ghosts. The closedIn function results true if pacman has not enough time to get to a crossroad before a ghost will be at his position. We calculate this time by making use of a crossroad representation.

```
def closedIn(self, gstate: gamestate.Gamestate):
    isClosedIn = True
    for pos, move, cost in self.successors(gstate, gstate.pacman):
        crossroadDist = self.distancer.get_distance(gstate.pacman, pos)
        ghostDists = {ghost: self.distancer.get_distance(pos, ghost) for
                    ghost in gstate.ghosts}
            minGhostDist = min(ghostDists.values())
            if crossroadDist > minGhostDist:
                 isClosedIn = False
    return isClosedIn

def successors(self, gstate, state):
 """
Returns a list of successors, which are (position, moves, cost) tuples.
 """
    successors = []
    for firstMove in self.legal_moves(gstate, state):  # for all initial moves
        position = state + firstMove.vector
        path = [firstMove]

        # Keep moving while we can only move onwards or turn back (no crossroads
        or dead end)
        next_moves = [move for move in self.legal_moves(gstate, position) if
                        move != path[-1].opposite]
        while len(next_moves) == 1:
            position = position + next_moves[0].vector
            path.append(next_moves[0])
            next_moves = [move for move in self.legal_moves(gstate, position)
                        if move != path[-1].opposite]
            successors.append((position, path, len(path)))
    return successors

def legal_moves(self, gstate: gamestate.Gamestate, position):
    moves = []
    for move in util.Move.no_stop:
        new_vector = position + move.vector
        if not gstate.walls[new_vector]:
            moves.append(move)
```

```
return moves
```