INFO8006: Project 2 – Report

Romain Bonhomme – s201892
Charlotte Borbouse – s204283

October 27, 2022

1 Formalization

In this project, the search and ghost problem is implemented. That multi-agent environment, called a game, can easily be formalised as an adversarial search problem. It consists in finding the shortest path performed by Pacman in order to eat all the food dots but with the presence of a ghost in the maze. Ghost can be eaten using capsules in order to increase the score. However, if they eat Pacman, the game is lost. It can be represented by a game tree with a terminal test and a utility function. One agent, Pacman, is trying to reach this goal while the other agent, the ghost, is trying to prevent Pacman to reach this goal. We will define the problem statement.

State space: The initial state s_0, the actions a and the transition model, that are explained below, define the state space S of the problem.

Initial state: The initial state s_0, such as $s_0 \in S$, is composed of Pacman's initial position, the ghost's initial position and direction, the positions of the food dots present in the maze and the initial score. This initial state thus depends on the layout in which we run the game.

Player function: Our players set is $player(s)$ ={Pacman, Ghost}. The function will return Pacman, then Ghost, alternating between them at each move.

Actions: The set of legal actions is $action(s)$ = {North, South, West, East}. For any of the players, the only possible action is moving in any of these four directions. But, if Pacman is against a wall, he can not move into the wall. For the ghost, there is the same condition. Moreover, it also can not make a half-turn unless it has to.

Transition model: Each time Pacman makes a move, the state changes by updating Pacman's position. If it has eaten a dot with that move, the state changes the value of that dot to false. If it was the ghost's turn, we update the ghost's position. The function, $transition(s, a)$, returns the state resulting (s') from the action a applied on the given state s.

Terminal test: The terminal function $terminal(s)$ will be true if there is no food left in the layout, or if the ghost has eaten Pacman, and false differently.

Utility function: As we assume a zero-sum game, the utility function of the problem is defined as

$$utility(\text{state, player}) = \begin{cases} \text{score} & \text{if player = Pacman,} \\ -\text{score} & \text{otherwise.} \end{cases} \tag{1}$$

Where the score is defined in the project's statement. Each player needs to maximize its own utility function to win the game.

2 Minimax

a. With the Minimax algorithm, some sequences of moves can result in the same state, so repeated states appear. It may result in an infinite game tree. However, the completeness of the algorithm can only be guaranteed if the tree is finite. Indeed, Pacman has the possibility to go wherever it wants as long as the move is legal. Consequently, in most mazes, we can found at least one cycle. If we do not strictly exclude cycles, there will always be other states possible to visit and thus other nodes to expand. It explains why the algorithm will never end.

b. From the point of view of Pacman, there is no advantage in going through a cycle. Because the algorithm preserves the set of optimal strategy, as anytime a state will be visited a second time, the utility value of this second visit will always be inferior to the one of the first visit. Actually, the utility value corresponds to the score of the state and by the definition of a Pacman's game, the score associated to a game state decreases with time/moves of Pacman. So, when a state is visited a second time, additional moves must have been required to get from the first visit to the second visit, resulting in a loss of score points.

c. We should avoid these repetitions by memorizing which states we went through, so we would not compute them again, and the game tree would be limited at one occurrence of each states.

3 Heuristic and Cut-Off

3.1 Heuristic

The evaluation function will return a value which depends on 3 factors, $eval(s) = score(s) - 3 * min(distancesFood) + distanceGhost$.

1. **Distance between the closest food dot and Pacman ($min(distancesFood)$)**: Pacman will tend to choose states that bring it closer to the food and thus will achieve its goal faster. The coefficient 3 was chosen by trial and error to make sure that Pacman reaches its goal faster.
2. **Distance between the ghost and Pacman($distanceGhost$)**: Pacman will choose a state where it is further away from the ghost than another where it would be closer.
3. **Score($score(s)$)**: Pacman will choose in priority states that go faster to the goal. It brings a kind of optimality to Pacman's decision

3.2 Cut-Off

For this function, we consider four Cut-Off conditions.

1. **Winning/Losing state**: If the state is a winning state or a losing state the game is over. It would be absurd to expand the successor nodes so the program stops.
2. **Food dots**: If the state is a state where Pacman eats a food dot, it reaches a good step because the goal is eating all food dots so it is not necessary to expand furthermore. Moreover, we have to verify if the ghost is far enough not to eat Pacman.
3. **Quiescent state**: If the distance between the position of the ghost and the position of Pacman is greater than 6 - depth in the tree of state and greater than 1, it stops. This guarantee that Pacman will not be eaten by the ghost in the next moves. A state will be quiescent if we can be sure that in the children nodes, Pacman will not get eaten by the ghost.
4. **Depth**: If the depth of the given state in the tree is equal to 5, it stops expanding. We want to be sure that the number of expanded nodes does not explode as recursions. Because, it can lead to a big use of memory and crash the programm. It allows us to ensure that the recursion is stopped if no other conditions were completed. The value of 5 was chosen to have the best balance between the computation time and the results.