Pacman - Stanford CS 221 Fall 2017-2018

Rohit Apte

2017-10-29

1 Problem 1: Minimax

a **Before you code up Pac-Man as a minimax agent, notice that instead of just one adversary, Pac-Man could have multiple ghosts as adversaries. So we will extend the minimax algorithm from class (which had only one min stage for a single adversary) to the more general case of multiple adversaries. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.**
Specifically, consider the limited depth tree minimax search with evaluation functions taught in class. Suppose there are $n+1$ agents on the board, $a_0, \ldots, a_n$, where a_0 is Pac-Man and the rest are ghosts. Pac-Man acts as a max agent, and the ghosts act as min agents. A single depth consists of all $n+1$ agents making a move, so depth 2 search will involve Pac-Man and each ghost moving two times. In other words, a depth of 2 corresponds to a height of $2(n+1)$ in the minimax game tree. Write the recurrence for $V_{max,min}(s,d)$ in math. You should express your answer in terms of the following functions: `IsEnd(s)`, which tells you if s is an end state; `Utility(s)`, the utility of a state; `Eval(s)`, an evaluation function for the state s; `Player(s)`, which returns the player whose turn it is; `Actions(s)`, which returns the possible actions; and `Succ(s,a)`, which returns the successor state resulting from taking an action at a certain state. You may use any relevant notation introduced in lecture.

Lets revisit the case of one adversary from the class. Minimax would alternate for agent and opponent and calculate the max and min (see image).

For this we know that the formula for depth limited search is given by

$$V_{max,min}(s,d) = \begin{cases} Utility(s) & isEnd(s) \\ Eval(s) & d = 0 \\ \max\limits_{a \in Actions(s)} V_{max,min}(Succ(s,a),d) & player(s) = player \\ \min\limits_{a \in Actions(s)} V_{max,min}(Succ(s,a),d-1) & player(s) = opponent \end{cases}$$

Now if we have $n+1$ agents we need to modify minimax to first do max for agent a_0 (pacman) and then min for each agent $a_1, \ldots, a_n$. For example for 3

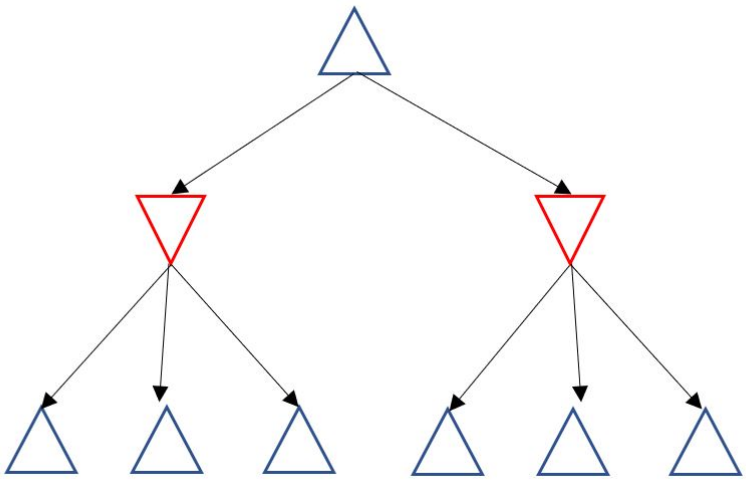

Figure 1: Minimax with 1 adversary

ghosts, see image for what the minimax looks like.

We need to add another parameter called agent to account for which agent we are calculating minimax for. It will denote either the player or one of n opponents (ghosts). nextAgent is simply the next agent to calculate for. If we use indexing then for n ghosts, 0 denotes pacman and $1, \ldots, n$ denotes each ghost. The first agent will be 0 (pacman), the next agent 1 and so on until n. the $n + 1^{th}$ agent is pacman (agent 0).
For n ghosts, recursing through all n of them is considered a depth of 1. So we need to adjust our $d - 1$ to $mod(d)$ where

$$mod(d) = \begin{cases} d & nextAgent < (n+1) \\ d-1 & nextAgent = (n+1) \end{cases}$$

Then we have

$$V_{max,min}(s, agent, d) = \begin{cases} Utility(s) & isEnd(s) \\ Eval(s) & d = 0 \\ \max\limits_{a \in Actions(s)} V_{max,min}(Succ(s,a), nextAgent, d) & player(s) = player \\ \min\limits_{a \in Actions(s)} V_{max,min}(Succ(s,a), nextAgent, mod(d)) & player(s) = opponent \end{cases}$$

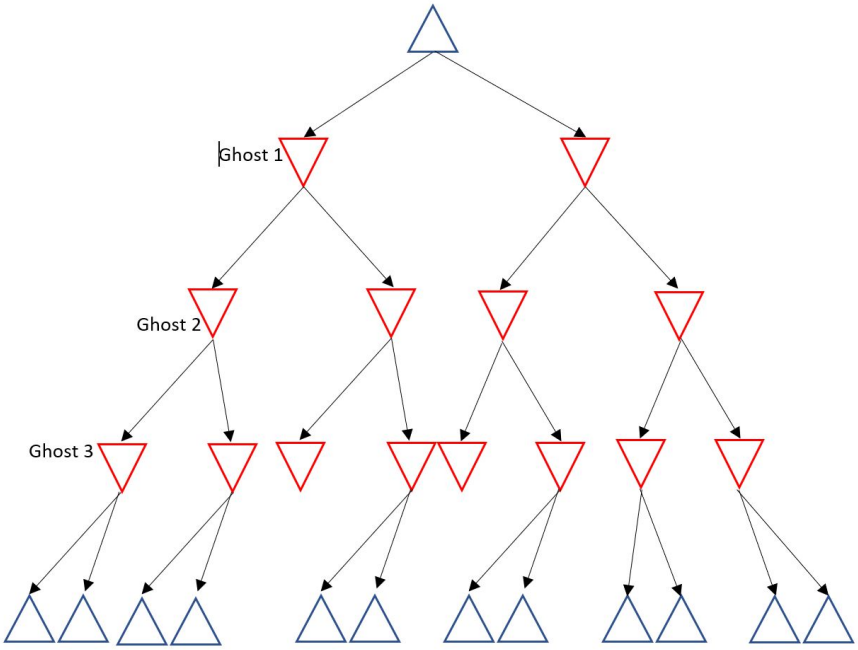

Figure 2: Minimax with 3 adversaries

2 Problem 2: Alpha-beta pruning

3 Problem 3: Expectimax

a Random ghosts are of course not optimal minimax agents, so modeling them with minimax search is not optimal. Instead, write down the recurrence for $V_{max,opp}(s, d)$, which is the maximum expected utility against ghosts that each follow the random policy which chooses a legal move uniformly at random. Your recurrence should resemble that of Problem $1a$ (meaning you should write it in terms of the same functions that were specified in $1a$).

Expectimax is similar to minimax except that the ghost actions are random. So using the same conditions as question 1, if we uniformly sample from the random movements of the ghosts, we take the average move for the opponent. I.e. we have

$$V_{max,exp}(s, agent, d) = \begin{cases} Utility(s) & isEnd(s) \\ Eval(s) & d = 0 \\ \max_{a \in Actions(s)} V_{max,exp}(Succ(s,a), nextAgent, d) & player(s) = player \\ \frac{\sum_{a \in Actions(s)} V_{max,exp}(Succ(s,a), nextAgent, mod(d))}{|Actions(s)|} & player(s) = opponent \end{cases}$$

Note that nextAgent, mod(d) are the same as defined in problem 1.

4 Problem 3: Evaluation Function (Extra credit)

a

b Clearly describe your evaluation function. What is the high-level motivation? Also talk about what else you tried, what worked and what didn't. Please write your thoughts in pacman.pdf (not in code comments).

My evaluation function is as follows

```
score=currentGameState.getScore()
          -2*invDistToHuntingGhost
          +15*scaredTime*invDistToScaredGhost
          -2*remainingFood
          -3*invDistToClosestCapsule
          -1*distToClosestFood
```

We want pacman to have an incentive to do the following

- eat the food pellets
- eat the capsules
- avoid the hunting ghosts
- eat the scared ghosts (based on scared timer)

I used the following features. The magnitude of the weights were achieved by trial and error. The sign of the weights is explained below

- currentScore - from the original evaluation Function. Just used as done in original.
- invDistanceToHuntingGhost - this is 1/distance to the closest active ghost. we take the inverse since the closer the ghost is, the higher the number needs to be. We use a negative weight since the closer the ghost is, the more negative our score should be.
- scaredTime*invDistanceToScaredGhost - pacman should eat the scared ghost if there is sufficient time for him to reach it. 1/distance to scared ghost to signify a smaller distance = higher value. We use a positive weight since the closer a scared ghost is, the better for us. We want pacman to eat scared ghosts!
- remainingFood - the remaining food pellets. Negative weight to signify that more remainingFood = lower score. I.e. pacman should eat the food to improve the score.
- invDistanceToClosestCapsule = 1/distance to the closest capsule. We want pacman to eat capsules he passes by. Again negative weight means pacman can improve the score by eating the capsule (and therefore the next capsule is further away giving a lower negative score)
- invDistanceToClosestFood = 1/distance to closest food. Negative weight implies pacman can improve the score by eating the nearby food.

Pacman sometimes just stays in one place when there is 1 or 2 food pellets left but they are far away. So I also added a condition that says if there are less than 3 food pellets left, increase the reward for going after them (I did this by making the 1/min distance to nearest food to be very large when less than 3 food pellets left).