**CSE 537 Project 2 Report**

By Anke Li

03/07/2015

**Question 1:** in this question, I used some states and actions to make return value (or so called score) more informative.

First, I checked if move would lead to win. If so, return a big positive value;

Second, I checked if any ghost is too close to pacman. If so, return a big negative value so pacman will not get too close to ghosts;

Third, I calculated all distances from foods to pacman and then deducted the nearest distance from score. So pacman will prefer move that has closest food available;

Fourth, I encourage pacman to eat food and to not stop by adding or deducting some points when it does so.

In this question, the grader was stuck forever before I added more factors that could affect pacman’s movement. I tried to add more criteria and finally my algorithm could finish ten times of openClassic board.

**Question 2:** the way I deal with multiple ghosts is this:

At first, I tried to separate each ghost to be in one MiniMax tree. For example, if there are 3 ghosts, then I’ll construct three MiniMax trees with each ghost be the one “min” agent in one tree. Then I have to think a way to combine the search results from these three trees so I could determine what the next move for the pacman is.

Then I found that these ghosts are actually not independent to each other. Since we could consider that each agent moves consecutively (pacman-> ghost 1 -> ghost 2 -> … -> ghost n), one ghost’s decision is affected by the move from previous ghosts!

So I have the current idea: constructing this tree so that one ply is: max -> min -> min -> … -> min. All ghosts is trying to find the min and eat pacman. It works perfectly, as you can see in my code.

Answer to the question in project guide: since if pacman just avoid being eaten, the score will continuously go down. In such case, pacman will go towards the food or even commit suicide, since “the constant penalty for living”.

**Question 3:** very similar to question 2, the key of this question is how and when to stop searching. Here is one thing I think is important and hard to catch:

When we search for min value, we maintain beta to record the current minimum value, and when we search for max value, we maintain alpha to record the current maximum value. This is easy to understand. Now here is the trick: we maintain those values for our children! For example, when we search for min value, we maintain beta, and we check alpha came from parent to decide if we should stop searching! Same with max search.

Once I understood this, implement alpha-beta search became easy since all I need to do is adding two values and in each search function, maintaining one and checking another one.

**Traces:** Please find running output in /traces folder

**Comparison:** run following two commands:

1. $ python pacman.py -p MinimaxAgent -a depth=3 -f -l smallClassic –q
2. $ python pacman.py -p AlphaBetaAgent -a depth=3 -f -l smallClassic –q

Using two agents to solve exactly same problem, and they both solved it. But Minimax agent used 32872116 bytes memory, while AlphaBeta agent only used 19223440 bytes memory and is much faster. I have attached two output in my /traces folder. We could see that AlphaBeta agent has less expanded nodes. Difference also exists in running time. Minimax agent used 3.12s to win, while AlphaBeta agent only used 1.58s.