Assignment2

Wang Youan (3035237236)

April 3, 2016

1 Reflex Agent

The evaluation function I implemented is mainly based on two factors,

1. **The distance between ghosts and Pacman**. The evaluation function give higher score, while Pacman is further away from ghosts. The Manhattan Distance between ghost and Pacman is greater than 5 is regarded as a safe distance, and returns 5 point bonus each ghost. while less than 2 is a dangerous, immediately return minimum score. When ghost is eatable, this value will times -1.
2. **The distance between Pacman and food**. Use Breadth first search (GSA) to find the nearest food. As the path cost is always one, BFS acts just the same as UCS, and can find the optimal solution. Besides, the process of generating points will produce many duplicate position which have been visited before, so I choose GSA to improve the speed.

Also this function will give one bonus if the action is not STOP.

Normally, this evaluation function acts very good in both "testClassic" (average score is 561, and winning rate is 100%) and "mediumClassic" (average score is 1503.171, 1584,winning rate is 99.8%, 93.5% for one ghost and two ghosts respectively).

2 Minimax Agent

I first generate all the possible next actions that the agent can do, then use "evaluate_function" to get the score of current action, then find the highest score, and return the action related to that score

I use recurse to achieve the evaluate_actions, which needs 3 inputs, agent_index, depth, game_state. This single method can act as both min-value and max-value based on current agent index.

The problem of such agent is the speed of get action. In my Mac, for a "smallClassic" layout, when depth is set to 4, it usually takes about 3 seconds to do one action. Besides, as the evaluation function is not so good, often, Pacman will remains stay if both ghosts and food are a little far away from it.

3 Alpha-Beta Prune Agent

The technique I used in this question is slightly different from that in section 2,

1. **Evaluate-action function add alpha and beta parameters**. These two new parameters are used to prune those unused leaves.
2. **Use three functions to finish this recurse**. This makes the code much easier to read.

This agent acts faster than Minimax Agent. If usually takes about 1.5 seconds to determine the action. However, in the same cases of that to Minimax Agent, this also choose to stay.

4 Expectimax Agent

It is true that usually ghost just random walks, not an adversary who makes optimal decisions and sometimes Minimax agent choose wrong action. That's why we need Expectimax Agent, which will calculate the possibility of every enemy's action and choose the action with highest expect score.

The code of this agent looks much similar of that in section 2, the only difference is that instead of return min-value, it will return the average value of all actions.

This gives ”trappedClassic” about 49.2% winning rate (compares to 0% of Minimax Agent)

5 Better Evaluation Function

I use similar method as in section 1, based on the distance of Pacman to ghosts and Pacman to the nearest food.

On my computer, in Question5, average score is 1219.9, win rate is 100% and running time is about 9s, which I think is a considerable good performance.

6 Tic Tac Toe Game

This game is a little different from the traditional Tic-Tac-Toe game, as player only can cross X.

I defined 3 classes to implement this game,

- **GameBoard** Use to represent one board. Main method includes,
  - *__eq__()* check whether two boards are equal. The difficulties in this function is that board with different order may be the same board (e.g. *X123X5678* and *01X3X5678* actually are same board), so we need to rotate the board, and compare whether one board is equal to another, its rotated and symmetric boards.
  - *get_valid_actions()* returns valid actions of current board (if no such action, return an empty list).
  - *evaluate_board()*. Evaluate current game board and give a score, based on [1]. Use a function to generate all the board shape as a lookup table, and will return of value in that table.
- **TicTacToeGame** main game class, store all the boards, players information, judge whether current player's action is a valid move, and check whether game is over.
- **AIPlayer** mainly used to get action based on current board status. As can be found in [1], if current player want to win, he must ensure that after his move, the position of all boards must in a, b^2, bc, c^2. Otherwise, his adversary will have the chance of win. For odd numbers of board, the player go first win, but even just the opposite. So, under the assignment's rule (3 boards, AI go first), there must exit such move that AI must win. So just one step BFS search is enough to find right action.

Besides, I implement *merge_state()* to merge the states of all board together. As for multiple board, just simply add all the board score together may not get the final score. This function also bases on [1].

I did a little more than required. In my game, you can play in any number of boards and choose to be first hand or second hand. Also, you can also watch two AI players play against each other. The difficult lies in optimal move not always exists in some cases. (e.g two boards, the first player has no optimal solution). So I just simply choose an action with least optimal solution number, and wish next player can't find the best choice.

References

[1] Plambeck, Thane E., and Greg Whitehead. *The Secrets of Notakto: Winning at X-only Tic-Tac-Toe.* arXiv preprint arXiv:1301.1672. 2013 Jan 8.