Design evaluation function for Pacman Game with Minimax, AlphaBeta, and Expectimax: The short-term goal, when evalFn is NOT all you need!

Quan Hoang Ngoc - 22521178

April 2024

Keywords:
- *Adversarial Games and Modeling: Objectives, Utility, Policy, Game Tree, Value of States*
- *Adversarial Search Algorithms: Minimax, AlphaBeta, Expectimax*
- *Resource Limitation and Evaluation Function*

1 Theoretical Background

- To understand the theories presented below, people need knowledge about some of the above keywords.

1.1 Adversarial Games, Adversarial Search Algorithms, and Evaluation Functions

- *Adversarial Games* (Zero-Sum Games) are types of games that include both oneself and one's opponents. In these games, our objectives and those of our opponents are opposite of each other. *Pacman Game* is an adversarial game where the player controls Pacman and the opponents are ghosts.

- We want to go to an *end state* where we win and the opponents are losers. For computers to understand these, we assign each end state a score that corresponds to the *utility* we receive when the game ends. Of course, the end states that we win and have higher scores. Now, the problem is we need to find a *policy* that recommends a move from each state to *maximize our utility.*

- To recommend a move from each state we must compute the *value of a state* which is the best achievable utility when starting from that state until the game ends for it's successor states, and then select the best action.

- Because we are not sure about opponents' actions, we can not compute the value of states if we don't assume about opponents' behaviors: pessimistic that they always play optimally (Minimax, AlphaBeta) or optimistic that their actions are random (Expectimax).

- Finally, because of resource limitations, we can not search on *all states of the game tree.* Thus, we only can simulate the game to a certain depth and then use an *evaluation function* to estimate the actual value of states.

- Attention in this understanding, the value of the evaluation function for each state is an approximate value for the best achievable utility when starting from that state **until the game ends**.

2 How to design a evaluation function

- An easy way to design an evaluation function is to think out of the box about *some features* that affect the behavior or move decisions of Pacman at each state.

- For example, the score Evaluation Function uses the current score of *simulated states* (non-terminal states) by search algorithms to estimate the value of these states.

- The default better Evaluation Function uses linear combinations of some features such as distance to nearest food, capsule, ghost, and number of the rest foods, and then transforms these features into non-linear and assigns weights for them to balance them. This function has quite a few **disadvantages and unconsidered situations**, but our evaluation function isn't based on improving this evaluation function, we will not discuss more about it.

2.1 Proposal Evaluation Function

- Attention to the theory that we present above, evaluating the "good level" for a simulated state until the game ends is **very difficult** because you must answer the question that the temporary end state helps eat all the rest foods, and capsules easily and escape from the ghosts. How to evaluate this when it is a distant future: many many turns and various situations. Yes, it is a long-term goal.

- This will be easier if you only evaluate for the near future. In other words, our approach is that at each turn of the game, before calling the search algorithms you will define an **incremental short-term goal** and try to achieve it in the best possible way with the hope that it will help you achieve a long-term goal in the distant future. This is similar to greedy ideas!

- Our short-term goal is to try to eat ***K* nearest foods and remaining capsules** in current maps at the ROOT STATE (it is the root of the search tree, the state when search algorithms are called, not simulated states by algorithms). To avoid constantly changing your goals, you must be steadfast in your goals. Thus, we let a parameter ***P*** (patient) and only refine the goals after each ***P*** times algorithms are called.

- To set up for Pacman to perform these, we use a *penalty feature f1* which is the length of the shortest path to the Pacman at TEMPORARY END STATES by algorithms that can eat all goals (K foods and remaining capsules). This is a feature that is complex to compute, we must use both Postman, Dijkstra algorithms and the Caching technique for this feature.

- At the same time, we want stronger penalties that Pacman eats capsules, so we use a *penalty feature f2* which is the total distance from Pacman at temporary end states by algorithms to capsule goals.

- While *f1 and f2* direct the movement of pac-man to the goals, we also use additional two features to decide the behavior of pac-man when it performs these goals. The *bonus feature g1* is the difference between the number of foods at the game state when the short-term goal is defined and the temporary end state by algorithms. Similarly, the *bonus feature g2* is the difference between the number of capsules at these two states.

- We also want Pacman to stay away from the ghosts, so we use *penalty feature f3* which is the total distance from Pacman's position at the temporary end states by algorithms to the position of ghosts (`not frozen`) at the root state (`you also can change it to a temporary end states`). And then, we reverse this feature to help Pacman stay away from the ghosts.

- However, when the ghost is frozen, we use this feature as a bonus score instead of a penalty score. The *bonus feature g3* which is the length of the shortest path to Pacman can eat the ghosts (maximize is 3 ghosts, use Postman, and Dijkstra algorithms). Similarly, we also reverse this feature.

- When the game goes to the *end stage* (`the number of remaining foods is smaller than parameter`

`F`), there may exist multiple actions for which the above criteria all give the same value. Thus, we additionally use the *bonus feature g-score* which is the difference score between the state when the short-term goal is defined and temporary end states by algorithms. At most times of the game, we don't much care about this feature, but when the game goes to the end stage g-score spikes and makes Pacman select the action that ends the game quickly.

- Besides, if a state is the winner it will be a *bonus g-win* score and *pen f-over* if that state is a loser.

- In short with our approach, objectives of simulation by search algorithms is to **prioritize the choice of plans** (short journeys) to realize the short-term goal rather than evaluate the "good level" of a state until the end of the game. We want to choose the good journeys to help achieve the short-term goal and in those journeys, we prioritize journeys that help us eat more food, especially capsules, stay away from ghosts or eat them if possible and quickly end the game when needed.

Notes:
- Root state
- Temporary end state and simulated state by algorithms
- State when the short-term goal is defined
- Short journeys that create by simulation of algorithms

2.2 Tuning the weights

- To simplify the tuning work, we **scale features** to range [0, 1]. Thus, we need to play a few training games (**2 training games**) to collect data for the scaling task. This is **very important** because after features are scaled it will become more stable. You also don't need to tune more for the weights (all weights equal 1).

- However, we use an Adaptive Weights technique, which will increase or decrease the weights when needed. When Pac-Man can eat the ghosts very easily, the weight of g3 increases **3.0 times** make Pacman becomes reckless, and when the game goes to the end stage, the weight of g-score spikes to end the game quickly.

- Normally, the weight of the g-score is **0.25**, we don't want it to affect the evaluation. The weight of f3 equals **1.5** because we want Pacman to care more about the ghosts, and the weight of g1 equals **0.75** because eating food is not as important as others.

- Parameters K, P, and F we choose are **3, 2, and 5**. The constant g-win is **10** and f-over is **20**.

	Weight	Special Weight
f1 (short-term goal)	-1	None
f2 (capsules)	-1	None
f3 (ghosts)	**-1.5**	None
g1 (eating foods)	**0.75**	None
g2 (eating capsules)	1	None
g3 (eating frozen ghosts)	1	**3.0**
g-score	**0.25**	**Spike**
g-win	10	None
g-over	-20	None
K (no. nearest foods)	3	None
P (no. patient)	2	None
F (no. end foods)	5	None

Table 1: Configuration Table

3 Experiments and Evalation

- **map**: name of layout (map)
- **sc**: use scoreEvaluationFunction
- **def**: use default betterEvaluationFunction
- **my**: use the proposal evaluation function
- **Exp**: use the Expectimax algorithm
- **Min**: use the Minimax algorithm
- **Alp**: use the AlphaBeta algorithm
- The first value corresponding with achieve **score** after the game ends, and the second value equal 1 if **win** the game

	map	scExp	defExp	myExp	scMin	defMin	myMin	scAlp	defAlp	myAlp
0	capsule	(-430, 0)	(-430, 0)	(1058, 1)	(-430, 0)	(-440, 0)	(865, 1)	(-430, 0)	(-440, 0)	(865, 1)
1	capsule	(368, 0)	(-459, 0)	(869, 1)	(-506, 0)	(-459, 0)	(868, 1)	(-506, 0)	(-459, 0)	(868, 1)
2	capsule	(109, 0)	(-112, 0)	(849, 1)	(-548, 0)	(-112, 0)	(1048, 1)	(-548, 0)	(-112, 0)	(1048, 1)
3	capsule	(-404, 0)	(-289, 0)	(1453, 1)	(-333, 0)	(-302, 0)	(181, 0)	(-333, 0)	(-302, 0)	(181, 0)
4	capsule	(-307, 0)	(-533, 0)	(1258, 1)	(-496, 0)	(-533, 0)	(1816, 1)	(-496, 0)	(-533, 0)	(1816, 1)
5	small	(1221, 1)	(982, 1)	(1374, 1)	(-204, 0)	(950, 1)	(1574, 1)	(-204, 0)	(950, 1)	(1574, 1)
6	small	(1249, 1)	(985, 1)	(1567, 1)	(230, 0)	(1178, 1)	(1160, 1)	(230, 0)	(1178, 1)	(1160, 1)
7	small	(1070, 1)	(952, 1)	(1576, 1)	(989, 1)	(933, 1)	(1575, 1)	(989, 1)	(933, 1)	(1575, 1)
8	small	(-4, 0)	(954, 1)	(1572, 1)	(-4, 0)	(964, 1)	(1561, 1)	(-4, 0)	(964, 1)	(1561, 1)
9	small	(1329, 1)	(945, 1)	(1728, 1)	(1329, 1)	(-343, 0)	(1762, 1)	(1329, 1)	(-343, 0)	(1762, 1)
10	medium	(-560, 0)	(1252, 1)	(1739, 1)	(-558, 0)	(1254, 1)	(1739, 1)	(-558, 0)	(1254, 1)	(1739, 1)
11	medium	(1520, 1)	(1734, 1)	(1892, 1)	(1520, 1)	(-675, 0)	(1926, 1)	(1520, 1)	(-675, 0)	(1926, 1)
12	medium	(-1076, 0)	(1521, 1)	(1929, 1)	(-1074, 0)	(1451, 1)	(1726, 1)	(-1074, 0)	(1451, 1)	(1726, 1)
13	medium	(1512, 1)	(1274, 1)	(1707, 1)	(1512, 1)	(1274, 1)	(1904, 1)	(1512, 1)	(1274, 1)	(1904, 1)
14	medium	(1373, 1)	(1701, 1)	(1914, 1)	(1373, 1)	(1309, 1)	(1914, 1)	(1373, 1)	(1309, 1)	(1914, 1)
15	minimax	(513, 1)	(-498, 0)	(-499, 0)	(-498, 0)	(-498, 0)	(-498, 0)	(-498, 0)	(-498, 0)	(-498, 0)
16	minimax	(-495, 0)	(505, 1)	(-499, 0)	(-497, 0)	(505, 1)	(-499, 0)	(-497, 0)	(505, 1)	(-499, 0)
17	minimax	(-494, 0)	(-497, 0)	(-497, 0)	(-495, 0)	(-497, 0)	(-495, 0)	(-495, 0)	(-497, 0)	(-495, 0)
18	minimax	(513, 1)	(512, 1)	(513, 1)	(513, 1)	(-495, 0)	(513, 1)	(513, 1)	(-495, 0)	(513, 1)
19	minimax	(-497, 0)	(511, 1)	(-499, 0)	(-497, 0)	(-499, 0)	(-499, 0)	(-497, 0)	(-499, 0)	(-499, 0)
20	test	(420, 1)	(562, 1)	(564, 1)	(420, 1)	(564, 1)	(564, 1)	(420, 1)	(564, 1)	(564, 1)
21	test	(540, 1)	(561, 1)	(562, 1)	(540, 1)	(563, 1)	(561, 1)	(540, 1)	(563, 1)	(561, 1)
22	test	(552, 1)	(562, 1)	(562, 1)	(518, 1)	(562, 1)	(562, 1)	(518, 1)	(562, 1)	(562, 1)
23	test	(546, 1)	(561, 1)	(564, 1)	(544, 1)	(558, 1)	(556, 1)	(544, 1)	(558, 1)	(556, 1)
24	test	(520, 1)	(560, 1)	(564, 1)	(520, 1)	(564, 1)	(562, 1)	(520, 1)	(564, 1)	(562, 1)
25	trapped	(-502, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)
26	trapped	(-502, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)
27	trapped	(-502, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)
28	trapped	(-502, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)
29	trapped	(-502, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)	(-501, 0)	(-502, 0)	(-502, 0)

Table 2: Statistics Table

-
- **meanAvgScore**: mean of Average Scores at each map
- **meanWinRate**: mean of Win Rates at each map

- The computers used for these experiments are **Kaggle CPUs**. We also limit the run time for each map is **10 minutes**. Despite of a powerful computation and setting the **algorithm's depth limit to 3**, we were only able to run successfully on 6 maps, the remaining 5 maps would probably need more time to complete all 5 runs.

- The result of report is complete objectivity, we only are limited by the time, not exist cherry picking the maps in the result of experiments.

	mertric	scExp	defExp	myExp	scMin	defMin	myMin	scAlp	defAlp	myAlp
0	meanAvgScore	219.27	376.87	710.33	45.43	175.53	681.20	45.43	175.53	681.20
1	meanWinRate	0.47	0.60	0.70	0.37	0.47	0.67	0.37	0.47	0.67

Table 3: Evaluation Table

3.1 Evaluating the evaluation functions

- Following Table 3, easy to say that our proposal evaluation achieves the **state of the art** on both score and win rate metrics. Even the results it achieves are always significantly higher than the other two functions for the three algorithms used.

- At the same time, we also see that the better Evaluation Function achieves better results than the Score Evaluation Function.

- This is quite understandable to show that the higher the complexity function, the higher the opportunity for better results.

- However, the cost of complex computation is not easy. we need to collect some data and use techniques such as Caching, Scaling, Pre-building, etc to optimize the programs.

3.2 Evaluating the algorithms

- Following Table 3, perhaps **Expectimax achieves the best result** in the three algorithms. It is a little better than the rest.

- This shows that the ghosts' strategies are not too dangerous and tend to be chosen randomly. Thus, being too pessimistic with Minimax may be redundant.

- The results achieved by AlphaBeta and the policy it recommends are the same as Minimax. Because it is just a pruned version of Minimax for better efficiency.

- At the same time, **AlphaBeta** also is an algorithm that achieves the **best efficiency** when it may be just search on half of the game tree instead of the whole tree like the other two algorithms.

4 Discussion and Future development directions

- Note that in the theory above, simulating the actions of the opponents (ghosts) did not affect the features too much as we used the positions of the ghosts in the root state instead of the positions of them in a simulated state, but the results are better.

- **Simulating the actions of the ghosts** seems only to support hypotheses about the opponents' choice of actions in the algorithms.

- This is because we find that the Pacman Game is **not a perfect adversarial game**. When Pacman and the ghosts don't have a competing goal. Pacman's goal is to eat food dots and capsules while the ghosts do not accomplish this goal, they only play the role of preventing Pacman from achieving this goal.

- Furthermore, these are **static goals**, they do not move at all. Unlike chess, the goal of two players is to compete with each other, and the pieces can always be moved.

- Therefore, simulating Pacman's actions only allows us to make a safer journey when moving to the goals. However, it is quite redundant, as we only need to know the current positions of the ghosts to stay away from them.

- In the future, based on these, people can design an algorithm that does not need to simulate the ghosts' actions to help improve the performance of the algorithm.

- People can also try more experiments with the parameters we prepared in the CONTROL.py file. It is quite easy to people will achieve a better result for this game with these ideas.

5 References

- All about source code, experiment process documentation, and YouTube demo (if there) in this GitHub link: GitHub[1] to visit them.
- Video Demo[2] to see the demo of our proposal.
- Kaggle Experiments[3] to see the run space of our experiments.
Author: Quan Hoang Ngoc. Thank you for your reading.