VNUHCM

University of Information Technology

Report for Multi-Agent Search

Nguyen Duy Hoang
22520467
CS106.O21

Contents

1 Introduction

1.1 Overview of Pac-Man Gameplay

Pac-Man is a classic arcade game where players control Pac-Man, a yellow circular character, as it navigates through a maze filled with dots while avoiding ghosts. The main objective is to consume all the dots without getting captured by the ghosts.

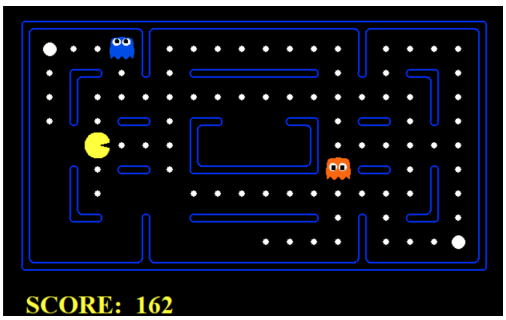

Figure 1: **An example of a Pac-Man gameplay level**

1.2 Multi-Agent Search Methods

Multi-agent search methods are algorithms designed to optimize decision-making in scenarios involving multiple agents, such as Pac-Man and the ghosts in the game. These methods consider the actions and behaviors of all agents simultaneously, aiming to find optimal strategies for achieving specific objectives.

In the context of Pac-Man, multi-agent search techniques are crucial for devising strategies that enable Pac-Man to navigate the maze efficiently while avoiding capture by the ghosts. By analyzing various factors such as movement patterns, potential ghost locations, and available pathways, these methods empower Pac-Man to make informed decisions that maximize its chances of success.

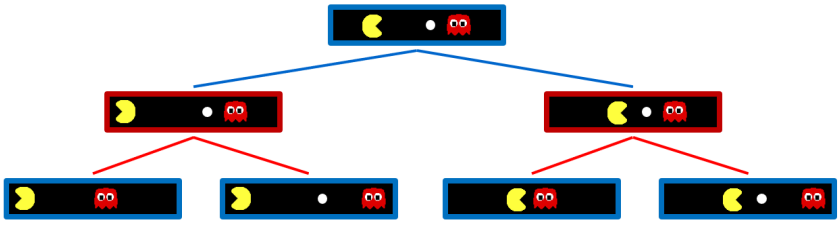

Figure 2: **Adversarial Game Trees**

1.3 Objective of the Homework

In this homework, I will delve into the exploration and implementation of multi-agent search techniques to enhance Pac-Man's gameplay performance. Specifically, I will focus on the following techniques:

- **Minimax**: This technique, which is already provided, involves simulating the moves of both Pac-Man and the ghosts to determine the best move for Pac-Man while anticipating the ghosts' actions.
- **Alpha-Beta Pruning**: A variant of the Minimax algorithm, Alpha-Beta Pruning aims to reduce the number of nodes evaluated in the search tree by eliminating branches that are guaranteed to be worse than the current best option.
- **Expectimax**: Unlike Minimax, Expectimax considers the uncertainty of the opponents' actions and incorporates randomness into the evaluation of potential moves. This technique is particularly useful when the ghosts' behavior is not entirely predictable.

In addition to exploring these techniques, I will also design and implement a function to improve the evaluation mechanism used in Pac-Man gameplay.

2 betterEvaluationFunction

2.1 Why we need to design this function?

The default evaluation function, scoreEvaluationFunction, simply returns the score of the state, which is the same one displayed in the Pacman GUI. However, this basic evaluation function lacks a detailed and specific strategy, leading to an ineffective approach for the Pacman agent. In this context, it becomes necessary to design another evaluation function, namely betterEvaluationFunction, to assist Pacman in creating a more effective strategy and potentially achieving a higher score. This new function aims to incorporate more nuanced considerations and decision-making processes, allowing Pacman to make more informed and strategic moves based on various factors such as ghost positions, food locations, and capsule availability. By designing a better evaluation function, we can enhance the performance of the Pacman agent and improve its overall gameplay experience.

2.2 Idea of betterEvaluationFunction

In the `betterEvaluationFunction` function, the score is initialized as *prev_score*, which represents the score before Pacman takes any further action. Then, the function checks each ghost agent to determine whether it is affected by Pacman eating a capsule (scared ghost) or not affected or no longer affected. Subsequently, it calculates the distances from Pacman to their positions and stores them in two arrays: *scared_ghost_distance* and *not_scared_ghost_distance*.

If a ghost is in a scared state, the score is incremented by 300.0 divided by the number of steps from Pacman to the ghost's position. This implies that the closer Pacman gets to eat/terminate the scared ghost, the higher the score received. When the distance equals 0, the score is further incremented by 500 to encourage the termination of the scared ghost.

If a ghost is in a normal state, the function searches in the *not_scared_ghost_distance* array to find the nearest ghost to Pacman. Then, the score is decremented by 20 divided by the number of steps from Pacman to the nearest non-scared ghost. This indicates that as a ghost gets closer to Pacman, the score is negatively affected - deducted - hence Pacman avoids the ghost. Additionally, if the distance to the nearest non-scared ghost equals 0, the score is decremented by 99999. This extremely negative value reflects that the score is severely affected if Pacman collides with a non-scared ghost, emphasizing Pacman's priority to avoid ghosts in all possible circumstances.

Furthermore, the function sets the weights for Pacman's actions regarding food and capsules: Regarding food, it calculates the average distance from Pacman to all food dots. Then, if Pacman is far from any ghosts with a distance greater than or equal to 2, Pacman will prioritize eating capsules or food based on the proximity: the function sets the score to 10 divided by the distance to the nearest food or capsule. This weighting of 10 is chosen to prioritize Pacman avoiding ghosts over eating food or capsules.
However, if the distance to the nearest non-scared ghost is less than 2, Pacman will prioritize eating capsules over food, and above avoiding ghosts to both prevent losing the game and gain points by eating scared ghosts. In this case, the function sets the score to 10 divided by the distance to the nearest food or 30 divided by the distance to the nearest capsule. The decision to assign a higher weight to capsules reflects Pacman's strategic choice to eat them when closer to non-scared ghosts, aiming to both evade danger and gain points by eating scared ghosts.

Additionally, reducing the average distance to remaining food or capsules has a lesser impact on Pacman, as it will not significantly affect Pacman's decision-making process when there are nearby food dots or capsules along its path. However, if all food dots or capsules are concentrated in one side or one corner, reducing the distance between Pacman and the nearest food or capsule will have a greater impact, as it directly affects Pacman's ability to quickly consume nearby food or capsules.

Lastly, the influence of the remaining food dots is established: the fewer food dots remaining, the more positively they affect the score. The weighting for this factor is set to a large value after the -99999 penalty if the distance to the nearest non-scared ghost equals 0, emphasizing one of the most crucial objectives of playing Pacman, which is to eat all the food dots to advance to the next level.

3 Statistical Analysis

3.1 Statistics

In this section, I have developed a Python script named `run_all_map.py` to iterate through and execute all layouts, automatically saving the results to the `output.txt` file. Subsequently, I have created another script named `table.py` with the purpose of reading the input from the `output.txt` file to generate LaTeX tables and save them to the `table.tex` file.

Regarding the details of running the maps, the majority of layouts were executed with the `depth=3` parameter and both evaluation functions: `scoreEvaluationFunction` and `betterEvaluationFunction` (abbreviated as `score` and `better` in the code snippet). However, for the `originalClassic` and `trickyClassic` maps, setting `depth=3` resulted in excessively long computation times for one level, so I adjusted the `depth` parameter to 2 for these two maps. Additionally, for the `openClassic` map, due to the evaluation function using `scoreEvaluationFunction`, which estimates based only on the score and the limited evaluation within three steps, the Pacman agent will only roam around the left half of the map and will not be able to complete the game. Therefore, I omitted the statistical analysis for runs using `scoreEvaluationFunction` on this map.

Below are two example images of runs using `AlphaBeta` and `Expectimax`:

Figure 3: **AlphaBeta**

Figure 4: **Expectimax**

Additionally, here is a video of what I consider the best run when using `Expectimax` with `depth=3` on the `mediumClassic` map: video

Below are the statistical tables for runs with various parameters on different maps:

Map: **capsuleClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	-501	-501	-502	547	547	1171
22520468	-451	-451	-214	186	186	743
22520469	-533	-533	124	1778	1778	1453
22520470	-451	-451	-451	1775	1775	1932
22520471	-443	-443	26	1095	1095	1811

Map: **contestClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	-364	-364	-304	1585	1585	2335
22520468	-494	-494	-434	2383	2383	1351
22520469	1747	1747	156	586	586	1517
22520470	-428	-428	-429	3095	3095	2652
22520471	-23	-23	-23	1865	1865	2774

Map: **mediumClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	-247	-247	-1737	2070	2070	2061
22520468	-1752	-1752	-1752	1693	1693	1931
22520469	1472	1472	1529	2103	2103	2044
22520470	730	730	730	2109	2109	2093
22520471	570	570	-4019	932	932	931

Map: **minimaxClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	-504	-504	-513	-504	-504	-505
22520468	-495	-495	-494	-497	-497	-497
22520469	-495	-495	-497	-497	-497	-497
22520470	511	511	511	513	513	513
22520471	514	514	515	514	514	514

Map: **openClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467				1411	1411	1436
22520468				1433	1433	1428
22520469				1226	1226	1371
22520470				895	895	1347
22520471				1172	1172	1403

Map: **originalClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	-330	-330	-331	1748	1748	2800
22520468	-101	-101	-101	1718	1718	2483
22520469	-36	-36	-36	2348	2348	1400
22520470	634	634	634	-238	-238	2501
22520471	-243	-243	-160	2131	2131	1135

Map: **powerClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	95	95	2913	4207	4207	4837
22520468	774	774	2554	2835	2835	4910
22520469	2615	2615	676	3273	3273	5276
22520470	723	723	1977	3495	3495	4987
22520471	1259	1259	3724	4538	4538	3394

Map: **smallClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	305	305	1307	1767	1767	1726
22520468	1175	1175	1175	1727	1727	1746
22520469	-74	-74	1153	1413	1413	1717
22520470	1257	1257	809	1720	1720	1774
22520471	-438	-438	-440	1566	1566	1424

Map: **testClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	546	546	542	560	560	560
22520468	484	484	486	561	561	564
22520469	484	484	488	564	564	562
22520470	540	540	540	562	562	562
22520471	492	492	492	564	564	564

Map: **trappedClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	-501	-501	532	532	532	532
22520468	-501	-501	-502	-502	-502	-502
22520469	-501	-501	-502	-502	-502	-502
22520470	-501	-501	532	532	532	532
22520471	-501	-501	-502	-502	-502	-502

Map: **trickyClassic**

Seed	Score			Better		
	Minimax	AlphaBeta	Expectimax	Minimax	AlphaBeta	Expectimax
22520467	382	382	696	1647	1647	1378
22520468	1015	1015	762	1716	1716	2074
22520469	595	595	63	3549	3549	3102
22520470	-286	-286	-41	4106	4106	1282
22520471	217	217	-357	2345	2345	657

3.2 Analysis

In this section, I analyze the performance of different evaluation functions and search methods based on the statistics obtained from the tables presented in the subsection "**Statistics**" above. Specifically, I have written a new Python script named `avg_score.py` to calculate the average score for each column corresponding to each evaluation function and search method.

The following table provides an overview of the analysis of the 11 game data tables in terms of win rate and average score:

Map: **capsuleClassic**

	Score			**Better**		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	0%	0%	0%	40%	40%	80%
Avg Score	-475.8	-475.8	-203.4	1076.2	1076.2	1422

Map: **contestClassic**

	Score			**Better**		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	20%	20%	0%	40%	40%	60%
Avg Score	87.6	87.6	-206.8	1902.8	1902.8	2125.8

Map: **mediumClassic**

	Score			**Better**		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	20%	20%	20%	80%	80%	80%
Avg Score	154.6	154.6	-1049.8	1781.4	1781.4	1812

Map: **minimaxClassic**

	Score			**Better**		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	40%	40%	40%	40%	40%	40%
Avg Score	-93.8	-93.8	-95.6	-94.2	-94.2	-94.4

Map: **openClassic**

	Score			**Better**		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)				100%	100%	100%
Avg Score				1227.4	1227.4	1397

Map: **originalClassic**

	Score			**Better**		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	0%	0%	0%	20%	20%	60%
Avg Score	-15.2	-15.2	1.2	1541.4	1541.4	2063.8

Map: **powerClassic**

	Score			**Better**		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	20%	20%	60%	20%	20%	80%
Avg Score	1093.2	1093.2	2368.8	3669.6	3669.6	4680.8

Map: **smallClassic**

	Score			Better		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	40%	40%	80%	100%	100%	100%
Avg Score	445	445	800.8	1638.6	1638.6	1677.4

Map: **testClassic**

	Score			Better		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	100%	100%	100%	100%	100%	100%
Avg Score	509.2	509.2	509.6	562.2	562.2	562.4

Map: **trappedClassic**

	Score			Better		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	0%	0%	40%	40%	40%	40%
Avg Score	-501	-501	-88.4	-88.4	-88.4	-88.4

Map: **trickyClassic**

	Score			Better		
	Minimax	**AlphaBeta**	**Expectimax**	**Minimax**	**AlphaBeta**	**Expectimax**
Win Rate (%)	0%	0%	0%	40%	40%	20%
Avg Score	384.6	384.6	224.6	2672.6	2672.6	1698.6

The comparison between the `betterEvaluationFunction`(better) and `scoreEvaluationFunction`(score) reveals a substantial advantage of the former over the latter. Across all games, the `betterEvaluationFunction` consistently delivers superior results, indicating its effectiveness in guiding the agent's decision-making process.

In terms of the search algorithms employed, both MiniMax and AlphaBeta Pruning demonstrate identical outcomes in terms of game performance. This parity suggests that the efficacy of these methods remains consistent across various game scenarios. However, AlphaBeta Pruning demonstrates a notable advantage in runtime efficiency over MiniMax, particularly evident in specific game instances where node expansion dynamics play a pivotal role.

Conversely, Expectimax emerges as the frontrunner in terms of win rate and average score across all methods tested. This superiority holds true even when employing both the `scoreEvaluationFunction` and `betterEvaluationFunction`, underscoring the robustness of the Expectimax algorithm in capturing game dynamics and optimizing agent performance.

In summary, if we were to rank the search algorithms based on their effectiveness (win rate) and performance (avg_score), the order would be as follows: Expectimax > AlphaBeta > Minimax. However, it is important to note that the relative performance of these algorithms may vary depending on the specific characteristics of the game environment and the nature of node expansions.