PacMan, Artificial Intelligence Approach

Joseph Grendzinski
grend016@umn.edu

March 2021

Abstract

In this paper we will discuss different approaches to applying artificial intelligence to play the classic game PacMan. Screen capturing and game stepping are used to examine the state of a game and make PacMan execute a plotted route of decisions. Run-time and score are examined.

Introduction

PacMan is a classic arcade game in which a user must navigate around a maze collecting pellets while avoiding 4 enemy ghosts that reside in the maze as well. This game provides a great platform to apply the study of artificial intelligence in order to develop a smart agent capable of making decisions within the game for us. There is a large variety of ways to approach this problem of intelligent decision making.

The ultimate goal of our agent is simply to clear one level of PacMan. In order to do this, PacMan must explore every space on the map that contains a yellow pellet whilst avoiding the enemy ghosts. It's important to dissect PacMan's environment closely in order to determine how to break down the problem into simpler components. We can assume that the environment is fully observable as the user can see the entire map (even if "Pacman" cannot). Also, because of the enemy ghosts, the environment can be considered multi-agent and dynamic as well. The dynamic nature of this game makes it much more difficult for a computer to keep up with game states and make correct decisions. In order to avoid issues like this at run-time, many approaches convert the game into steps in which the game would halt and the board would be examined. Examination time is vital as PacMan must be aware of each ghost's position on the map and whether or not they are released yet at a given state. PacMan's status can be tracked by the game's point system as each pellet awards 10 points. While points can also be scored from consuming cherries or ghosts (in flee mode), those will be ignored since we are attempting level completion rather than score.

There is also an interesting aspect to the ghost behaviors within the game. Rather than random wandering, each colored ghost has specific "personality traits" that affect how they will pursue PacMan. This is interesting because it gives the agent more information about which path would be least prone to collision. With enough time, PacMan can be refined to include more heuristic methods for even smarter decision making. However, more heuristics does not always mean better results. Each must be tested and examined to extract which aspects increase level completion/score.

Related Work

One approach began with making a variety of rules in order to determine each ghost's position relative to PacMan [6]. PacMan would simply make decisions depending on 85 adjustable parameters designed to avoid ghost confrontation and collect pellets. Parameter sets were tested using a fitness function in order to determine which was most successful. The algorithm used to create parameters was an implementation of population-based incremental learning (PBIL) in which probability vectors are utilized in place of genetic operations [3]. Rather than basing moves strictly on parameters, many solutions make use of search algorithms to determine the best route for PacMan to take depending on the state of the game. The biggest priority for most approaches is short-range ghost avoidance since encountering a ghost would result in defeat. Short-range ghost avoidance focuses solely ghosts within a certain amount of cells of PacMan [5]. PacMan is expected to find which paths are considered "safe," and select the one that would result in the most pellets being consumed. This short-range approach can be expanded in various different ways to maximize PacMan's score by increasing awareness of ghosts and their path probabilities. In order to actually determine the shortest path to take, one can utilize theories such as A*, Dijkstra, Finite State Machine, or Graph. Such algorithms can also be utilized in order to create paths for the ghosts attempting halt PacMan's progress [10]. These opposing problems have actually lead to competitions between PacMan controllers and Ghost controllers [8]. These competitions lead to even smarter decision making by playing against increasingly difficult opponents.

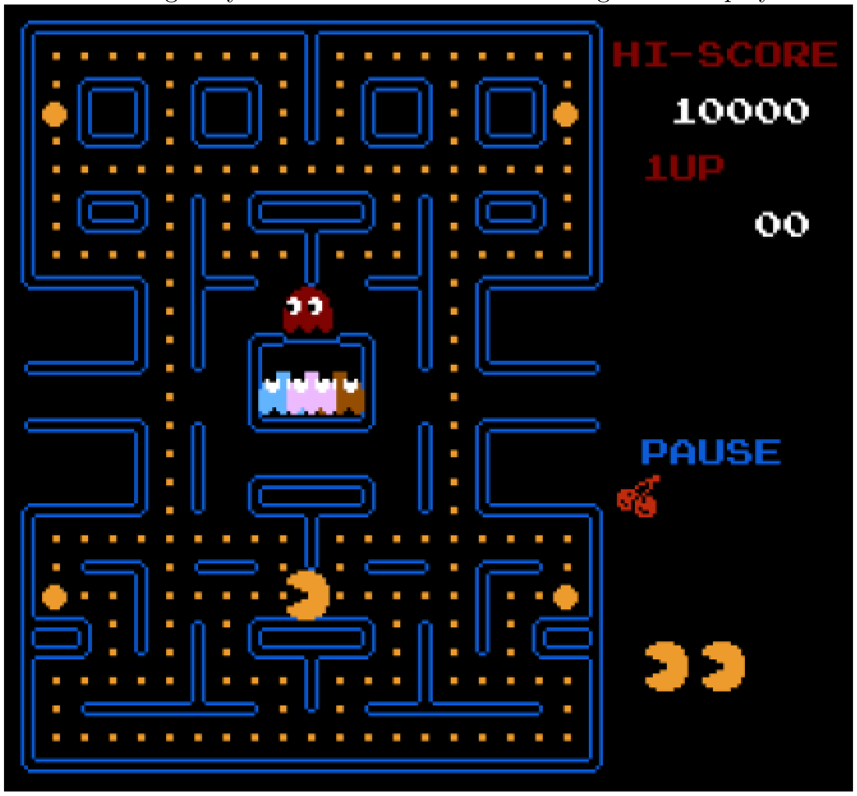
Monte Carlo Tree Search (MCTS) is a popular search to be used in games but it requires a decent amount of problem modification if it is to used for our PacMan game [9]. Despite this, MCTS has been very successful when it comes to General Game Playing [4]. Problems with MCTS and PacMan stem from the fact that the game can end up in an infinite loop without losing or making anymore progress within the game. It is important to make specific parameters in order to avoid large tree branches or repeated moves. Also, the real-time game play makes it quite difficult to execute the necessary computations to determine which route to take in time. To address this problem, one can stop the game in order to allow the computer time to think and plot a route. This route would then be taken until it is complete and another stoppage in play would occur. Depending on how long the time is between intervals, this may affect performance by trying to plan too far ahead in the game. The main advantage of MCTS that it can be considered an "anytime algorithm" [7]. This means that theoretically, the search can be stopped during any one of its iterations and still provide a reasonable solution. This is especially useful when trying to use AI to solve games since there may not always be time to find the most optimal solution.

Approach

The approach taken divides the game into short intervals in order to give the computer time capture the screen and to do calculations. We focus on short-range ghost avoidance and pellet proximity for decision making. Since a python version of PacMan could not be found to engage with, a Nintendo Entertainment System (NES) simulator was downloaded along with the NES PacMan game. I created a python script to launch the game and convert the game state into objects. PyAutoGui [2] is used heavily to capture screenshots and locate scripts. Script positions are converted from pixels to map coordinates in order for our agent to make use of them. PyDirectInput [1] is used in order to send key commands to the application to start the game and direct PacMan's movements. Each game interval consists of five decisions. PacMan plots his route by moving one cell in his current direction and checking if that cell is an intersection where a decision must be made. Once PacMan reaches an intersection, he makes a decision and begins to increment in that direction until the next. Each decision is noted until the route is complete and PacMan has commands to execute. After this, the game is resumed until the route has been executed. This continues indefinitely until PacMan either completes the level or loses all three lives.

Initially, only ghost avoidance was considered for path finding. If any ghosts exist within 10 cells of PacMan, the closest one is noted and PacMan creates preferences depending on where the ghost is located relative to PacMan. For example, if the ghost is to the right of PacMan, "left" would be added to PacMan's preferences. Now when PacMan is plotting his next five moves he will take one of the preferred paths if it is a valid option at each intersection. If none of the preferences are available, PacMan selects a random route. By just avoiding ghosts and staying alive, PacMan will randomly collect pellets.

Once PacMan could detect ghosts and make decisions depending on their locations, the next step was to incorporate some sort of pellet detection as well. Due to the uncertainty of locating scripts and executing routes perfectly, a more general approach was taken. Similar to the set of preferences our agent has, a new list of choices was created designed to guide PackMan towards areas with a large amount of pellets. To do this, I found the pixel coordinate of the top left pellet and incremented across each cell in the map. The RGB value of each pixel is taken and compared to the color of a pellet to determine if a pellet still exists there. This information is converted into a 21x27 two-dimensional board array in which each cell is designated by a 0 (open cell with pellet), 1 (wall), or 8 (visited open cell, "ate"). Using this array, the board is split into quadrants and the amount of pellets in each quadrant is summed. The quadrant with the maximum amount of pellets at that current state is noted. The new list of choices is constructed depending on PacMan's proximity to the maximum. For example, if PacMan is in the bottom left quadrant and the top right quadrant is the maximum, "right" and "up" will make up the list of prosperous directions. Exactly like the ghost avoidance, if one of these choices is available, it will be taken. However, the ghost avoidance preferences take priority over the prosperous directions since it is more important to remain alive than take a greedy route. The initial state of the game is displayed below.



Experiment

For our experiment, two separate routes are tested on ten games of three lives. Score and game time are recorded. To increase execution accuracy, the simulation was set to thirty-five percent. Key inputs are timed by distance (in terms of cells) to next decision. The first route tested was using only the ghost avoidance techniques (Table 1). The second route utilized pellet detection along with ghost avoidance (Table 2). Run-time includes think time and game time.

Table 1: PlotRoute Regular Ghost Avoidance

	Points	Run-time (s)
Trial 1	1480	376
Trial 2	1790	383
Trial 3	2010	631
Trial 4	1400	380
Trial 5	2020	723
Trial 6	1690	410
Trial 7	1930	561
Trial 8	1200	396
Trial 9	2490	677
Trial 10	970	435
Average	1698	497

Table 2: PlotRoute Ghost Avoidance with Pellet Detection

	Points	Run-time (s)
Trial 1	1890	563
Trial 2	1170	442
Trial 3	1610	457
Trial 4	1870	486
Trial 5	1320	318
Trial 6	1960	642
Trial 7	1630	300
Trial 8	660	250
Trial 9	1440	520
Trial 10	1580	581
Average	1513	456

Analysis

According to the data, implementation of the pellet detection actually produced worse results. This may be due to the danger that PacMan is faced with when exploring a larger amount of the map. After a couple pellets are consumed, the maximum quadrant may change fairly quickly leading to less efficient pellet consumption in PacMan's current quadrant. With only ghost avoidance, PacMan tended to be pushed down into the bottom half of the map and remain there. Because of this, less of the map is explored but the pellet consumption is much more efficient and PacMan can reach a fairly high score before multiple ghosts are free.

Both average a decent score but neither complete the level in three lives. More implementation can potentially improve results and decrease execution uncertainty. Re-design of the pellet detection function to include short-range pellet detection could prove to be beneficial.

Conclusion

Overall, both algorithms performed fairly well considering large amounts of uncertainty. Ghost avoidance is the most important factor for score as it increases the odds of surviving. Pellet detection improvement/expansion along would be my first step in improving results along with a more refined script capturing method. This problem is far from over! Expanding the thinking process is a vast subject that can incorporate any game detail no matter how small. While seemingly a simple game, breaking down the problem in order for a computer to understand is not so trivial.

Future Work

A large portion of the time spent on this project involved manually testing to ensure that details from the screenshots were being converted properly. Because of this, there was less time for testing more precise algorithms due to the complexity of the code. In the future I would like to incorporate short-range pellet detection and path-finding algorithms in order to select more prosperous paths when there are no ghosts nearby. I would also like to implement the personality traits within each ghost class to try and better predict what moves the ghosts are going to make when trapped in a complex position.

References

[1] Pydirectinput.

[2] Welcome to pyautogui's documentation!

[3] A. M. Alhejali and S. M. Lucas. Using genetic programming to evolve heuristics for a monte carlo tree search ms pac-man agent. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, 2013.

[4] Y. Bjornsson and H. Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[5] J. Flensbak and G. N. Yannakakis. Ms. pacman ai controller. 2008.

[6] M. Gallagher and A. Ryan. Learning to play pac-man: An evolutioanary, rule-based approach. In *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, volume 4, pages 2462–2469. IEEE, 2003.

[7] D. Perez, S. Samothrakis, and S. Lucas. Knowledge-based fast evolutionary mcts for general video game playing. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[8] P. Rohlfshagen and S. M. Lucas. Ms pac-man versus ghost team cec 2011 competition. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 70–77. IEEE, 2011.

[9] S. Samothrakis, D. Robles, and S. Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154, 2011.

[10] M. Zikky. Review of a*(a star) navigation mesh pathfinding as the alternative of artificial intelligent for ghosts agent on the pacman game. *EMITTER International Journal of Engineering Technology*, 4(1):141–149, 2016.