Model-Based Reflex Agent Design of Pac-Man Strategies with Support of Pathfinding Algorithms in a Partially Observable Environment

Jiachang (Ernest) Xu

Abstract— **This report documents the model-based reflex agent design of my Pac-Man strategies in a partially observable environment. Limited visibility dictates that Pac-Man in such an environment can only see foods that are 5 steps ahead of it, and 1 step left or right of it, if there are no walls at these locations. In addition, Pac-Man can only detect ghosts that are 2 steps from it. The objective of this Pac-Man project is to maximize win rate. In order to accommodate such challenges and the objective, I utilizes Breath-First Search algorithm and A* search algorithm to support Pac-Man's pathfinding functionalities, and breaks down the overall strategies into four major functional components (ordered by priority): (1) map-building operation, (2) survival mode, (3) hungry mode, and (4) corner-seeking mode. Map-building operation scans nearby environment and store such information into Pac-Man's internal memory; survival mode detects imminent threats by using A*, and evades accordingly; hungry mode uses Breath-First Search to route the Pac-Man towards the nearest food; corner-seeking mode guides Pac-Man through unexplored areas in the maze by navigating towards different corners. With such strategies, after running a significant number of tests, Pac-Man's win rate stabilizes at around 90%, with the highest win rate at 94%.**

I. Introduction

Imagine that you are in a maze. The night is dark, so you can only see 5 steps ahead of yourself, and 1 step left or right of yourself. There are ghosts wandering inside the maze, preying on you on silent steps. You can only hear them if they are 2 steps or closer to you. This is the challenges that Pac-Man (the agent) faces in a partially observable environment. This paper documents the design of strategies that the agent uses to maximize win rate. This means that, for each round of game, the agent needs mobility to evade from ghosts, while eating all the foods in the maze.

To achieve such an objective, the agent needs to combines functionalities of `HungryAgent`, `SurvivalAgent`, `CornerSeekingAgent`, and `MapBuildingAgent` from the last few exercises. The new agent class is named `PartialAgent`. I designed `PartialAgent` based on the concept of model-based reflex agents. `PartialAgent` performs its overall strategies using 4 major functional components (ordered by priorities): (1) map-building operation, (2) survival mode, (3) hungry mode, and (4) corner-seeking mode. The map-building operation maintains persistent internal memory, logging new information at every step; the survival mode scans for imminent threats from nearby ghosts, and initiates evasive maneuvers; the hungry mode routes the agent towards the nearest food, provided a safe surrounding; the corner-seeking mode guides the agent through unexplored areas to learn new information about the maze. After a few rounds of fine tuning, this overall strategy of `PartialAgent` has a stabilized win rate of around 90%.

In Section II, I will reason why `PartialAgent` is an example of model-based reflex agents to begin with. In Section III, I will explore how Breath-First Search and A* algorithms are used in this Pac-Man project to find path from the agent's location to any target location. In Section IV, I will explain the overall strategies in depth, including all 4 major functional components. Finally, in Section V, I will briefly discuss how I use creative approaches to optimize code structures, and the agent's performance.

II. Model-Based Reflex Agents

This section iterates the reason why `PartialAgent` is a model-based reflex agent. The book *Artificial Intelligence: A Modern Approach (3rd Edition)* gives a graphic example of how a model-based reflex agent perceives information and acts accordingly (Fig. 1). Due to the challenge of limited visibility inside the maze, the agent should build an internal memory of the environment that is not observable at the moment. In my design, `PartialAgent` maintains a list of internal variables that store coordinates of known foods, walls, and corners. Subsection III-A will discuss such an operation in depth. In summary, this ongoing map-building operation gives the agent newest knowledge about the environment that is beyond the agent's visible range. These internal variables reflects the up-to-date information about the environment. Therefore, `PartialAgent` is a model-based reflex agent.

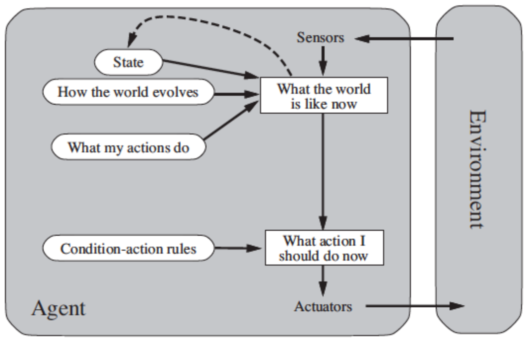

Fig. 1. A example of model-based reflex agents from *Artificial Intelligence: A Modern Approach (3rd Edition)* by Russel and Norvig

III. Pathfinding Algorithms

Before getting into the specifics of my Pac-Man strategies, it is important to go through the two pathfinding algorithms that I used in this project: Breath-First Search and A*. In the following two subsections, I will discuss these two pathfinding algorithms and the scenarios in which I use them.

A. Breath-First Search Algorithm

The first pathfinding algorithm that I implemented was the Breath-First Search (BFS) algorithm[1]. The BFS algorithm finds the shortest path from the agent to the closest food in the maze. The intuition of the BFS algorithm adapted to this Pac-Man project is that, given the state of the game, I opened an imaginary tap at the agents location to flood fill the maze, the frontiers of the flood expand at the same speed while recording the back-tracing relations, and when the first flood gets flooded, the algorithm traces the shortest route from that food back to the agent.

B. A* Search Algorithm

The second pathfinding algorithm that I implemented was the A* search algorithm[2]. The A* algorithm is the combination of the Dijkstra algorithm and a heuristic function (i.e. in this project, the Manhattan distance from a node to the goal node). The A* algorithm finds the shortest path from the agent to a designated target (i.e. a ghost or a corner). I implemented the A* algorithm in addition to the BFS algorithm, because A* runs more efficiently than the BFS algorithm.

IV. Overall Strategies

The Pac-Man agent I designed is named `PartialAgent`. Its overall strategy combines the functionalities of `HungryAgent`, `SurvivalAgent`, `CornerSeekingAgent`, and `MapBuildingAgent`. The `PartialAgent` performs map-building operation through the entirety of the game. At every new location (i.e. every time the game calls the `getAction()` method on the agent), the `PartialAgent` prioritizes the survival mode first, the hungry mode second, and the corner-seeking mode third. This overall strategy aims to maximize the win rate, while trying to gain full visibility of the maze by maintaining an internal memory of the map. In the following four subsections, I will discuss these four functional components of my Pac-Man strategies in depth.

A. Map-Building Operation

As briefly reasoned in Section II, map-building operation follows the concept of model-based reflex agent to build an internal memory that models the environment that is beyond the agent's observable range. More specifically, the `PartialAgent` maintains persistent internal memory of (1) a set of coordinates of known, uneaten foods (`self.foods`), (2) a list of coordinates of walls (`self.walls`), (3) a list of coordinates of corners (`self.corners`), (4) an integer registering which corner the `PartialAgent` should seek (`self.counter`), and (5) a stack of coordinates that can guide the `PartialAgent` to the closest food (`self.path_to_food`). In each round, the first time the `PartialAgent` calls the `getAction()` method, it populates internal memory of walls and corners by invoking the related functions in the api.py file, eliminating any future computational redundancy. At every new location, the `PartialAgent` removes its current location from `self.foods`, because this location does not have any food after this time stamp. If the current location of the `PartialAgent` is one of the corners, the `self.counter` switches its corner-seeking object to another corner, so that during the next time of corner-seeking mode, the `PartialAgent` does not keep coming back to the same corner. After the map-building operation is done, the `PartialAgent` decides if it needs to enter the survival mode.

B. Survival Mode

For the decision of the survival mode, the `PartialAgent` scans for any nearby ghosts. If there are ghosts present nearby, the `PartialAgent` runs the A* algorithm on each nearby ghost to generate the path from the agent to each ghost. Therefore, the `PartialAgent` is able to anticipate which direction the most imminent threats come from. However, the `PartialAgent` does not evade from every imminent threat. For example (Fig. 2), the Manhattan distance from the agent to the ghost is 2, so the agent can detect this ghost. However, the actual path for the ghost to each the agent is not short enough to pose credible threat upon the agent, because of the long wall between them. My solution is to set a threshold of 3 steps on the calculated path to each ghost. If the length of the path from the agent to the ghost is less than 3, the `PartialAgent` will identify that ghost as a credible threat and remove the incoming direction of this threat from the available legal actions. The final step of the survival mode is to pick randomly from the pool of available legal actions. The `PartialAgent` clears its internal memory of `self.path_to_food`, which reminds itself to recalculate the route the next time it is in the hungry mode. If there are not any credible threats present nearby, the `PartialAgent` decides if it needs to enter the hungry mode.

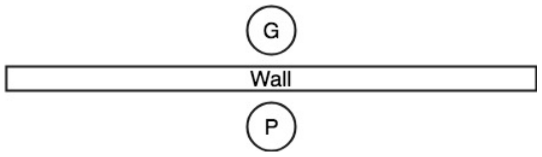

Fig. 2. An example of non-imminent threat from a nearby ghost

[1]I have learned the Breath-First Search (BFS) algorithm during my undergraduate study at University of Southern California, in pursuit of the B.S. Computer Science/Business Administration degree, from the class CSCI 104: Data Structures and Object-Oriented Design.

[2]I have learned the A* search algorithm during my undergraduate study at University of Southern California, in pursuit of the B.S. Computer Science/Business Administration degree, from the class CSCI 360: Introduction to Artificial Intelligence.

C. Hungry Mode

For the decision of the hungry mode, the scans for any nearby foods. If there are foods present nearby, the `PartialAgent` adds their coordinates to the `self.foods` internal memory. Given that there are any known locations in the `self.foods`, if the `self.path_to_food` memory is empty, the `PartialAgent` uses BFS algorithm to calculate the shorted path to the closest food and store the path in the `self.path_to_food` memory. After ensuring the `self.path_to_food` memory is not empty, the `PartialAgent` simply pops the first location from `self.path_to_food` memory and moves to that location. By storing the path to closest food in its internal memory, the `PartialAgent` prevents recalculation of the path, unless it encounters ghosts. This design reduces computational redundancy. If there are neither credible threats from ghosts nearby nor any food locations to the its knowledge, the `PartialAgent` enters the corner-seeking mode.

D. Corner-Seeking Mode

For the decision of the corner-seeking, the `PartialAgent` use the `self.corners` list and the `self.counter` index to locate the target corner. The `PartialAgent` then runs the A* algorithm on the target corner and follows the calculated path to build map knowledge about the unexplored areas. While exploring new areas, if the agent encounters ghosts or finds new foods, it reacts accordingly in the next action.

V. Creativity, Optimization and Evaluation

After several rounds of fine tuning, I have increased the win rate of 50 rounds from around 70% to around 90%. The highest win rate I have achieved so far is 94% in the `mediumClassic` layout. I the following 4 subsections, I will discuss the creative steps that I have taken to exercise object-oriented design, reduce computational redundancy, predict imminent threats, and detect death traps.

A. Object-Oriented Design

It is a good practice in object-oriented design to exercise modular programming. Because the BFS and A* algorithms used in this project are general-purpose pathfinding algorithms, all the pathfinding methods related to BFS and A* is defined in the class `SearchAgent`, which inherits from `Agent`. Any further developments of this Pac-Man project that require BFS and/or A* algorithms to perform pathfinding functionalities can implement more classes that inherit from `SearchAgent`. For example, the `PartialAgent` inherits from `SearchAgent`. Therefore, `PartialAgent` automatically possesses the pathfinding functions defined in `SearchAgent`. This technique increases modularity of code structure (Fig. 3), making future development easier.

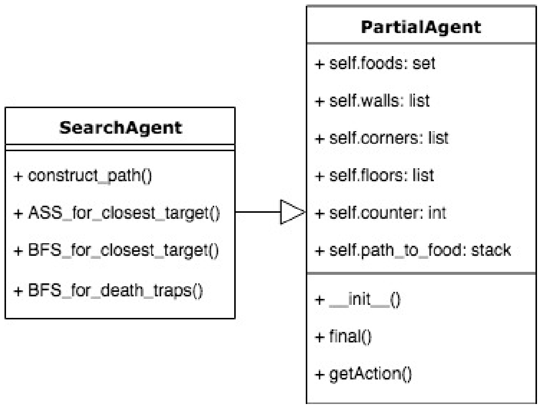

Fig. 3. Class diagrams of SearchAgent and PartialAgent

B. Reduce Computational Redundancy

To reduce computational redundancy, I use persistent internal memories, as stated in Section III-A. The persistent internal memories of the `PartialAgent` prevent from calculating the coordinates of walls and corners unless the agent is at the very first location of a round. In addition, the `PartialAgent` remembers the shortest path to the closest food unless disrupted by a close encounter with ghosts. This approach prevents the program from calling the BFS algorithm at every step, further reducing computational redundancy.

C. Predict Imminent Threats

I change the survival strategy from scanning a search grid to running the A* algorithm to anticipate the incoming direction of the valid threat. The problem of scanning a search grid might remove more than 1 direction from the pool of available legal actions, which is more than I need. For example (Fig. 4), the ghost is located to the northwest, so the agent need to remove both north and west from available legal actions, reducing the agents mobility. On the other hand, the A* algorithm anticipates that the most imminent threat from this ghost comes from the north, so the agent only removes north from available legal actions, which means more mobility. According to my experiments[3], this new approach has been proved effective. The old method, using a search grid, has a win rate stabilizing[4] at 70%; the new method, A* algorithm, has a win rate stabilizing at 85% (Table I).

TABLE I
Comparing Different Ways to Predict Imminent Threats

	Search grid	A* algorithm
Stabilized win rate	70%	85%

[3]In order to draw conclusions, I always run 500 rounds of games each on treatment and control.

[4]Due to the pseudo randomness present, one version of implementation shows small fluctuation around a stabilized win rate

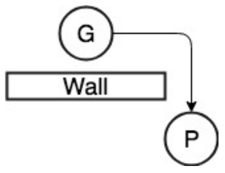

Fig. 4. By running A* on the nearby ghost, the agent predicts that the most imminent threat will come from north, so moving west is still a good option.

D. Detect Death Traps

During my experiments, I noticed that the agent gets cornered very easily by ghost within certain wall configurations. These wall configurations are identified with only one point of exit and entry, which make them death traps. For example (Fig. 5), the horseshoe-shaped wall in the center of the `mediumClassic` layout. If the agent accidentally wanders into such death traps with ghost following them, the agent is very likely to get cornered by ghosts, leading to a loss. The agent needs mobility to successfully evade from ghosts. Therefore, if the agent could detect death traps early on and avoid going in those directions, such tactic would theoretically increase the agent's win rate. My experiments on death trap detection[5] showed that by avoiding death traps, the agent has a even higher stabilized win rate of 90% (Table II).

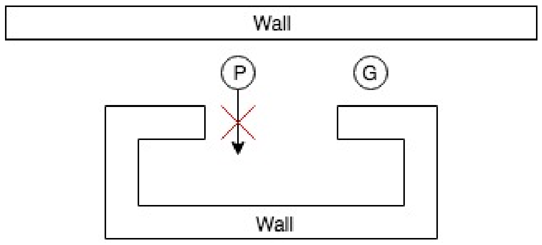

Fig. 5. Death trap caused by the horseshoe-shaped wall in the center of the `mediumClassic` layout. The agent should avoid moving south.

TABLE II
EFFECTIVENESS OF DEATH TRAP DETECTION

	No death trap detection	Death trap detection
Stabilized win rate	85%	90%

VI. CONCLUSION

To overcome the challenge of partially observable environment, and to maximize win rate, I designed a prioritized sequence of Pac-Man strategies that are based on the concept of model-based reflex agents. The Pac-Man agent of my design breaks down the overall strategies into four major functional components. First, map-building operation builds an internal memory of the environment to accommodate the limited visibility inside the maze. Second, survival mode scans for imminent threats from nearby ghosts by using A* algorithm on each ghost to predict its attacking route, and counters such threats by fleeing in other directions. Third, if no imminent threats present, hungry mode calculates the shortest path from the agent's location to the nearest food using the BFS algorithm, and the agent follows this path to eat food. Finally, if neither imminent threats nor known locations of food are present, corner-seeking mode routes the agent to unexplored areas to gather new information of the environment. Throughout this project, I have applied several techniques to provide creative solutions. Those include the good practice of object-oriented design, reducing computational redundancy, changing the way how the agent predicts the direction from which a ghost will attack, and developing a method to avoid death traps in the maze.

[5]A* algorithm is already in place to predict imminent threats for experiments on death trap detection.

REFERENCES

[1] Russell, Stuart J.; Norvig, Peter (2010), Artificial Intelligence: A Modern Approach (3rd ed.), Upper Saddle River, New Jersey: Pearson Education, Inc., ISBN 0-13-604259-7, chpt. 2.
[2] Moore, Edward F. (1959). "The shortest path through a maze". Proceedings of the International Symposium on the Theory of Switching. Harvard University Press. pp. 285292.
[3] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4. 4 (2): 100107.