Pacman: an Exercise in Multi-Agent Strategies

Derrick Stolee
University of Nebraska–Lincoln

April 16, 2009

1 Introduction

Multi-agent systems are generally very complicated. However, the classic arcade game, Pac-Man provides a scenario that is familiar to most students. This allows a quick understanding of the rules which reduces the learning curve required to devise intelligent strategies. What is missing from this combination is a method for students to implement their strategies and test them in the game environment. This project provides a game engine that is modular enough for students to build intelligent agents with a minimum amount of programming, but with immediate visual feedback through the game interface. The multi-threaded approach motivates strong strategies as well as fast algorithms and discourages malicious behavior.

Also, two strategies are implemented as examples. They both take a shortest-path approach but have state-specific goals that modify the weights of the paths. Also, one uses a heuristic to perform cooperation with other agents of the same type.

2 Rules

For the rules of the game, please refer to Shant Karakashian’s writeup. There are slight modifications to the setup, mostly in the file format for the maps. This difference is only in the characters used.

3 Implementation Details

The source files are available in a ZIP archive that can be imported as a project into the Eclipse IDE. It includes the source files in the src folder, binary class files in the bin folder, Javadocs in the doc folder, support files in the test folder, and a launch configuration PacManStartup.launch. The launch configuration will automatically add the Pacman application to the “Run...” menu in Eclipse. The ZIP archive is available at

http://cse.unl.edu/~dstolee/Pacman.zip

3.1 Source Organization

In the src folder, the base package for the Pacman project is edu.unl.cse.ai.pacman. This package contains the three most important classes: PacManStartup, Map, and PositionObjectPair.
The `PacManStartup` class contains the `main` method to start the application, configure the GUI, and initialize the game engine.

The `Map` class contains the grid of positions and lists of the objects contained in the grid. It also dictates how an agent’s action affects the environment.

The `PositionObjectPair` class is the link between the `Map` and the agents themselves. This class contains an object, its position, and information regarding remaining lives and score. This keeps the agents from being able to modify this information themselves (i.e. cheating).

The package `edu.unl.cse.ai.pacman.runtime` contains the classes that control the timing of agents making moves. Each agent is given its own thread, where a loop is run. Each iteration, the thread sleeps for a specified time interval then passes a copy of the `Map` instance to the agent, requesting a move. The copy keeps the agent from modifying the environment or even the other agents. The time interval is constant and universal over all threads, keeping a limit on the amount of computational resources an agent uses and simulating the time it takes to travel between positions on the map. After the agent returns a move, the `Map` instance is updated with that intended move. The move may have no effect, such as when an agent attempts to walk through a wall.

The package `edu.unl.cse.ai.pacman.ui` contains a single class, `MapPanel` that is responsible for drawing the graphics that visualize the map and the agents inside.

Students looking to implement their own agents must understand the `edu.unl.cse.ai.pacman.agents` package. Inside, the class `MapObject` contains the basic information necessary for any object in the map, including dots and fruits. Extending this class is the `AbstractIntelligentAgent` class. This class provides slightly more information than `MapObject`, most importantly is the object’s `TYPE` enumeration, being either `PACMAN` or `GHOST`. This is the class that will be extended in order to create new objects.

The package `edu.unl.cse.ai.pacman.runtime` contains the classes that control the timing of agents making moves. Each agent is given its own thread, where a loop is run. Each iteration, the thread sleeps for a specified time interval then passes a copy of the `Map` instance to the agent, requesting a move. The copy keeps the agent from modifying the environment or even the other agents. The time interval is constant and universal over all threads, keeping a limit on the amount of computational resources an agent uses and simulating the time it takes to travel between positions on the map. After the agent returns a move, the `Map` instance is updated with that intended move. The move may have no effect, such as when an agent attempts to walk through a wall.

3.2 Running the Engine

To run the engine, select the `PacManStartup` runtime configuration from the Eclipse “Run...” menu. A window will be presented that allows selection of the agents to use, such as the one in Figure 1. The left side corresponds to agents that will be the Pacman agents. The right side corresponds to the ghost agents. To increase or decrease the number of Pacman or ghost agents, click the “+” and “-” buttons at the top. Each combobox lists the available classes that can be presented as agents. These can be mixed and matched as pleased.

At the bottom is a file selection that allows you to choose which map file you want to load. It defaults to `test/pacmanmap.txt` which contains the classic Pacman map. To begin the simulation, click the “Start” button. This will load the game engine window and begin the simulation. See Figure 2 for an example.

This will run as an animation in real time with the computation of the agents. Scores are listed to the left for the Pacman agents and to the right for the ghost agents.

3.3 Implementing an Agent

For a student to implement their own agent, they must create a new class that extends the `AbstractIntelligentAgent` class and implements the `getAction` and `copy` methods. The `getAction` method provides the agent with an updated `Map` instance to run path finding algorithms. The `copy` method allows for duplication of the object that will be given
to the other agents in the field. To protect oneself from malicious competing programmers, this copy method can be written to provide only a shell of an object, as long as it contains the same position member. A copy constructor is provided in the base class to make this a very simple chore.

Once the class is written, the full class name must be added to the file test/classes.txt in a new line. The file is already populated with two random agents and the examples found in the following two sections. This allows the game engine to be aware of the newly implemented class and will allow the user to select the agent in the selection screen.

4 Example Strategies

A couple strategies were implemented to demonstrate the implementation of an intelligent agent. This also provides a benchmark for students wanting to compete in the game.
4.1 An Example Pac-Man Strategy

The class `edu.unl.cse.ai.pacman.dstolee.SmartPacMan` implements a strategy for a Pacman agent. At each request for a move, the agent follows a two-stage procedure. First, all ghosts in the map are located and breadth-first search is initiated with those positions as goals. Second, all dots and fruits are located and a similar approach is taken.

A breadth-first search is run on the positions of the map where the Pacman can move. This includes open squares, dots, and fruits, but does not include ghosts, ghost zones, or walls. Also, squares are linked over the border by the toroidal topology of the map. Each move is weighted one.

After the breadth-first search is executed for the ghosts, the distance from the agent to the nearest ghost is determined. If this distance is small enough, a decision is made to either evade the ghosts or chase them. This depends entirely on whether or not a fruit was eaten recently by checking the timer on the `Map` instance. If the timer is still running, the agent chooses to move to the neighboring position that has the smallest distance to a ghost, regardless of if the ghosts are close or not. If the timer is not running and the distance is small enough, the agent chooses the move that will change location to the highest distance from the ghosts. This move is returned immediately to avoid wasting time computing the other goal of eating dots. This is very important as the ghosts could have a computational advantage, since they have an easier goal.

If evasion or chasing is not taken in the first stage, breadth-first search is initiated from all dots and fruits. The adjacent position with the shortest distance to a dot or fruit is taken.

In either stage, randomness is used to break ties in equally-weighted moves, giving a more dynamic behavior pattern when running the game engine. Also, the breadth-first search is only recomputed if a certain time interval is passed, reducing the time it takes to return a move, but delaying the response from pursuing ghost agents. The delay is kept small enough that a ghost could not reach the Pacman without being noticed in the bounds given by the first stage of the algorithm. Moreover, the breadth-first-search information is shared among the Pacman agents through a `static` variable, reducing average computation when multiple agents are used in the game.

4.2 An Example Ghost Strategy

The class `edu.unl.cse.ai.pacman.dstolee.SmartGhost` implements a strategy for a ghost agent. At each request, Dijkstra’s algorithm is used to calculate the move that will bring the ghost closest to a Pacman agent without interfering with other ghost agents, using a heuristic in the weighting of the path. The heuristic is simple: starting at the Pacman agents, moves between adjacent squares are weighted one, and the weight is increased by a significant amount if a ghost agent already resides in that square. This helps cooperation between the ghosts as they will not all follow a single path to the Pacman if multiple paths are available. The simple procedure of adding weights also allows the ghost agents to be entirely anonymous, as they only need to consider the weights of adjacent squares.

As with the Pacman agent example, the weighting algorithm is shared among the agents and updated only after a certain time interval has passed. When equally-weighted options are provided, randomness breaks the ties for more interesting results.

4.3 Comparing Example Strategies

It is easy to propose that the `SmartPacman` and `SmartGhost` agents can defeat their random counterparts in almost all situations (there is always a minute chance that the perfect counterplay will be randomly selected). Competing against each other, the results are mixed. Almost always, four `SmartGhost` agents will surround and overpower a single `SmartPacman`. The Pacman is at a distinct disadvantage given the rules, as the ghosts can evade the Pacman at the same rate as the Pacman chases, giving little chance of eating a ghost without help. Multiple `SmartPacman` agents can surround multiple `SmartGhost` agents if they are lucky enough to eat a fruit before being surrounded by
the ghosts. In the standard map, the fruits are located such that it is hard to corner the Pacman agents without trapping them in an area that includes a fruit.

In the open map, all bets are off, as the toroidal topology and the equal speeds of travel make most games very uninteresting. They usually end up running around in circles.