# Programming Assignment 1: Pacman

Due: Friday 10/10/2024 @ 11:59pm EST

The purpose of programming assignments is to use the concepts that we learn in class to solve an actual real-world task. We will not be using Sepia for this assignment: I have developed a game engine for us to use. In this assignment we will be writing agents to play a Pacman-esque game.

Pacman is traditionally thought of as a single player game, the purpose of which is to control pacman to eat all of the food pellets on the map. In classical pacman, there are always four ghosts which will lazily pursue you in the map: if they make contact with pacman, pacman loses a life. Lose all lives, and the game is over!

Our version of pacman can be played without ghosts. Like classic pacman, the goal is to eat all of the pellets on the map, but in this version we want you to eat all the pellets in the shortest amount of time possible. We will start out with no ghosts present to test that your agent correctly finds a path that visits all of the pellet locations in the shortest amount of moves possible. Then, when that is working, we will start adding ghosts.

When there are ghosts on the map, your agent will still want to optimize for path length. However, the cost of a path is now no longer solely a function of pellet distance: it needs to account for the risk of running into a ghost! You will need to be clever in order to identify what kinds of risk exist in a world like this, and will need to adapt your definition of risk into your path cost. Be careful! You are solving this game using the A* algorithm, which means you cannot break admissibility and consistency of the heuristic!

## 1. Copy Files

Please, copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/pa1/lib/pacman-X.X.X.jar` to `cs440/lib/pacman-X.X.X.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/pa1/lib/argparse4j-0.9.0.jar` to `cs440/lib/argparse4j-0.9.0.jar`.
  This is a jarfile that `pacman-X.X.X.jar` depends on. It provides similar functionality to Python's `argparse` module. The documentation for `argparse4j` can be found here.
- Copy `Downloads/pa1/src` to `cs440/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/pa1/pacman.srcs` to `cs440/pacman.srcs`.
  This file contains the paths to the `.java` files we are working with in this assignment. Just like in the past, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- Copy `Downloads/pa1/doc/pas` to `cs440/doc/pas`. This is the documentation generated from `pacman-X.X.X.jar` and will be extremely useful in this assignment. After copying, if you double-click on `cs440/doc/pas/pacman/index.html`, the documentation should open in your browser.

## 2. Test run

If your setup is correct, you should be able to compile and execute the following code. A window should appear:

```
# Mac, Linux. Run from the cs440 directory.
javac -cp "./lib/*:." @pacman.srcs
java -cp "./lib/*:." edu.bu.pas.pacman.Main

# Windows. Run from the cs440 directory.
javac -cp "./lib/*;." @pacman.srcs
java -cp "./lib/*;." edu.bu.pas.pacman.Main
```

**NOTE:** The commands above will **not** run your code. There are several command line arguments you can provide, and one of them is to specifiy which agent controls pacman. This argument works like this:

```
java -cp "./lib/*:." edu.bu.pas.pacman.Main -a <YOUR_AGENT_CLASSPATH>
```

Your agent's classpath is `src.pas.pacman.agents.PacmanAgent`. You can alternatively use `--agent` instead of `-a` if you wish. Remember that if you want to see the other command line arguments available you need to add a `-h` or `--help` to the end of your command!

### Task 4: Moving Pacman with `PacmanAgent.java` (40 points)

Our first goal is using the file `src/pas/pacman/agents/PacmanAgent.java` to move pacman around the map. You will need to implement three methods in order for pacman to plan a route to a desired coordinate on the map:

1. `getOutgoingNeighbors(final Coordinate src, final GameView game)`. This method takes a `Coordinate` object as input along with a `GameView` object. A `Coordinate` object is as it sounds: it contains a pair of (x,y) values: x being the horizontal location on the map, and y being the vertical location. The `GameView` class is a read-only data structure that represents the state of the game at this moment in time. This method produces a `Set<Coordinate>` which should be populated with all of the coordinates that you can reach from the current coordinate (by making a legal action). Just to be clear, I would advise against taking any action which would cause you to remain in the current coordinate.

2. `graphSearch(final Coordinate src, final Coordinate tgt, final GameView game)`. This method is where you will implement a single-source shortest path algorithm to find the shortest path from `src` to `tgt` given the (frozen) state of the world `game`. You are welcome to choose whatever single-source shortest path algorithm you want! This method returns a `Path<Coordinate>` datatype. The `Path` datatype is used to *avoid* having to worry about storing a backpointer table. I personally find it really painful to have to debug a table of backpointers when something goes wrong, so the purpose of this datatype is to store the rest of the path in an ever-nesting structure. Please see the documentation for it!

3. `makePlan(final GameView game)`. This method is used for calling your graph traversal algorithm and post-processsing the `Path<Coordinate>` you receive into a `Stack<Coordinate>`. The purpose of doing so is that the `Path<Coordinate>` object will be in reverse order, and a `Stack<Coordinate>` is just more useful anyways. There is a field in your class (inherited from the super type) where you can store a `Stack<Coordinate>` along with getters and setters, so be sure to use them!

I highly encourage you to write your own tests for this. It is pretty straightforward to look at the map and understand (especially in the absence of any ghosts) what the "true path cost" is to go from any vertex to any other vertex on the map. Once you're happy with your functionality, please move on to eating all of the pellets in the shortest amount of time. Be warned, this is difficult!

**Task 5: Eating all of the Pellets with `PacmanAgent.java` (60 points)**

The file `src/pas/pacman/agents/PacmanAgent.java` has quite a few unimplemented methods that you will need to complete in order for pacman to eat the food pellets in the shortest amount of moves. In order to eat the food pellets in the shortest amount of moves, we would like to employ the A* algorithm (since it is a single-source shortest path algorithm that is goal-aware). The trouble is that the graph A* should search through is not very straightforward. Since our goal is to eat all the food pellets and we can only eat a single pellet at a time, we will need to traverse a graph whose vertices contain collections of (remaining) food pellets. For instance, at the beginning of the game, the collection (or "state" of the food pellet configuration) contains all food pellets on the map. When a pellet is eaten, the "state" of the food pellet configuration has changed and is now missing one of the pellets that existed previously. To search through all available food pellet configurations, the graph we must search through has vertices which contain collections of food pellets.

Of course there are a combinatorially large number of states, so we want to be as thrifty as possible while searching through this graph. To be clear, a single vertex in this graph contains a collection of coordinates: one coordinate for each food pellet that remains on the map. We connect two vertices together if there is at most one food pellet missing between the collections of those vertices. Our initial state is the vertex whose pellet collection matches that of the map we observe, and the goal state is a vertex whose collection is empty.

In order to implement this functionality, you will need to complete the following four methods inside `PacmanAgent.java`:

1. `getOutoingNeighbors(final PelletVertex vertex, final GameView game)`. This method takes as its two arguments a `PelletVertex` object and a `GameView` object. The `PelletVertex` class is a read-only data structure that represents a vertex in the pellet graph and has a collection of `Coordinate` objects inside it. This method produces a `Set<PelletVertex>` which should be populated with all of the outgoing neighbors of the input `PelletVertex`.

2. `getHeuristic(final PelletVertex src, final GameView game)`. This method takes two arguments, a `PelletVertex` object and a `GameView` object. The purpose of this method is to calculate your heuristic value that guesses how expensive it will be to go from `src` to the goal state (no food pellets remaining). The quality of your heuristic will matter for how fast your algorithm runs. Try to think of how you would guess the length of a path (and the vertex doesn't have supplementary data that is easy to process).

   *Hint:* If you had a graph where each vertex was a food pellet location, then the graph will be fully connected (because it is always possible to go between any pair of food pellets). You want to find the shortest path which touches all of the vertices in this graph. Unfortunately, solving this problem in general is NP hard, so we will need to approximate. The questions you should ask yourself here are: What are the edge weights and how can we approximate this path cost?

3. `getEdgeWeight(final PelletVertex src, final PelletVertex dst)`. This method takes two `PelletVertex` objects that are assumed to be neighbors. This method should decide how expensive the edge weight is to go from `src` and arrive at `dst`. Be careful! Remember that edge weights should be non-negative, and be sure not to break admissibility and consistency of your heuristic!

4. `findPathToEatAllPelletsTheFastest(final GameView game)`. This method is where you will do the actual searching. I would recommend writing a more generic version of A* somewhere else in your code and then calling it from here. This method returns a `Path<PelletVertex>`

datatype which should contain the shortest path from the vertex created from `game` to a goal state.

I would highly recommend writing your own unit tests for this part. The kind of path we are searching for is difficult and less abstract than say moving around on the map.

### Task 6: Extra Credit (50 points)

In order to earn the full extra credit for this assignment, you will have to demonstrate (on the autograder) that your agent can win in the presence of ghosts. If you can win at least 25% of games where there is a single ghost on the map (and you get three lives), you will earn 10 of the bonus points. If you can win at least 20% of games where there are two ghosts on the map (and you get five lives), you will earn another 15 of the bonus points. If you can win at least 10% of games where there are four ghosts on the map (and you get seven lives), you will earn the remaining 15 bonus points. This will require you to tweak both of your graph traversals as well as your heuristic in order to represent risk. Being close to ghosts is risky (but is not the only kind of risk!). You will need to be careful not to break admissibility and consistency of your heuristic when changing your edge weights to include risk. You will also need to be implementation-savy in order to not time out the autograder!

### Task 7: Submitting Your Assignment

Please drag and drop your `PacmanAgent.java` on gradescope. There will be multiple autograders, so be sure to submit to them all!