

CM0312 Artificial Intelligence Labs (2009/10)

1 Pac-Man Agent

These labs are based on T. E. G. Grenager's CS121 Introduction to Artificial Intelligence course at Stanford University, Summer 2006 using a modified version of his Pac-Man game environment. See <http://www-nlp.stanford.edu/~grenager/cs121/materials.html>. It is intended to get you started with the coursework. Note that this the version of Pac-Man used for this exercise and the coursework is notably different from the arcade version.

The Pac-Man game is played in a 2D maze build on a regular grid that contains dots and the player controls Pac-Man, a yellow disc with a mouth. The maze also contains four ghosts in different colours which move around according to their own strategy. When Pac-Man runs into a dot he eats it. When he runs into a ghost, he dies. When Pac-Man dies, he loses one of his three lives. Once he has eaten all dots in a maze, he moves on to the next level with a new maze. The aim of Pac-Man is to eat as many dots as possible without wasting time, while trying to avoid dying.

The objective is to design and implement a Pac-Man agent which gets a score that is as high as possible. In the lab classes we discuss only the basic environment and some simple approaches. For the coursework you are intended to write the best possible search-based agent as described in the coursework instructions.

1.1 Getting Started

To get started download the `pacman.tar.gz` package from the assignment section in blackboard and unpack it with

```
tar xzf pacman.tar.gz
```

into the current directory where all the files are in the `pacman` directory. The package has the following content:

- `src` — the complete sources for the environment, agents, etc. in Java (you need at least Java 5.0);
- `javadoc` — the javadoc documentation of the sources;
- `classes` — here the compiled classes will be installed. You should add this to your `CLASSPATH`.
- `Makefile` — a UNIX makefile to compile the code, etc.

We first discuss how to compile and run the basic environment. In a shell (terminal window) go to the `pacman` directory and type

```
make
```

to compile all the java sources (the Makefile has been kept simple and it will always compile all sources).

To run the game on a text console simply type

```
java pacman.Game
```

from the `classes` subdirectory (in case you added that directory to `CLASSPATH`, you can run it from elsewhere). You should see a text maze with a move prompt. Pac-Man is the `@` symbol, the ghosts are numbered and the `o` represent the dots, while `.` is a free field in the maze. You can move Pac-Man with `i`, `j`, `k`, `m` and quite with `q`. Press enter after each key. To get a graphical user interface run

```
java pacman.Game -display gui
```

There is also a `none` mode to run the game without display (used for fast evaluation of the agents).

If you are having any problems with this or what follows, please talk to the tutor.

1.2 The Code

Next we briefly discuss the java code of pacman. It is highly recommended that you carefully read the javadoc documentation, starting with the `index.html` file in `javadoc` (use your favourite browser to view this).

There are three java packages in `src`, compiled into classes:

- `pacman` — the core game interfaces and classes;
- `util` — generally useful utility classes;
- `ghosts` — algorithms for controlling the ghosts;
- `player` — implementations of Pac-Man agents.

For the lab classes and the coursework you are supposed to create your own agent in a single file in the `player` directory. Each agent you create should correspond to exactly one file in this directory. The `player` directory contains some example agents and a skeleton to get your started with your own agent. Either generate your own file in that directory or edit `MyPacManPlayer.java` to implement your agent. It is important that you do not edit any of the other files as this will mean your agent for the coursework will not run in the environment it is evaluated in.

Here is an overview of the pacman package:

- `Game` — class for Pac-Man game's core functionality, including the game's current state, the current level, number of points, etc. The key method is `playGame()`, and the `main()` method is called from at the command line.
- `State` — class to capture important elements of the game state within a particular level, including the locations of the dots, the locations of the the ghosts, and of course, the location of Pac-Man. All these aspects of the State can be gotten easily with "getter" methods. Note that Pac-Man's location is represented with a Location object, the ghost locations are represented with a `java.util.List` of objects of type Location, and the dots are represented using a LocationSet object. A State object also implicitly represents a history of states, since it contains a link to a parent State which you can get with the method `getParent()`.
- `Location` — class which represents a location in the Pac-Man maze. Location objects are used to represent the location of dots, ghosts, and Pac-Man. One can ask for the distance between locations with the `getEuclideanDistance()` and `getManhattanDistance()` methods.
- `LocationSet` — efficient implementation of the `java.util.Set` interface that is for Location objects only to makes comparison of states fast.
- `GhostPlayer` — the interface for classes that are called by the game to make ghost moves. The primary implementing class is `BasicGhostPlayer` which uses a simple deterministic algorithm to choose ghost moves, but for variety there are a few others.
- `PacManPlayer` — the interface for classes that will be called by the game to make Pac-Man moves. This is the interface that your class(es) will implement.

The `player` directory contains a simple agent to play the game in `SimplePacManPlayer.java`. To run this agent type

```
java pacman.Game -display gui -pacman player.SimplePacManPlayer
```

which tells the program which agent function to use to control Pac-Man. Once you have a version of your own player you can run it via

```
java pacman.Game -display gui -pacman player.MyPacManPlayer
```

(or whatever you called your player). If you want to quickly check your player run

```
java pacman.Game -display none -pacman player.MyPacManPlayer
```

to avoid wasting time on the displaying.

1.3 Building Your Own Agent

For the coursework you have to submit your own agent implementation by editing the file `MyPacManPlayer.java` in `src/player`. Only this file is to be submitted. The file already contains a basic skeleton for a Pac-Man agent function and you can have a look at `SimplePacManPlayer.java` to see how to use it to implement a simple player.

The core method you need to implement is the `chooseMove(Game game)`. This takes as argument a `Game` object which contains everything the agent needs to know about the game. You can get access to the current state (`getState()`), the ghost players (`getGhostPlayers()`), and the score (`getPoints()`). The method returns an object of type `Move`, which is a simple java `enum` of all possible moves: `Move.UP`, `Move.DOWN`, `Move.LEFT`, `Move.RIGHT`, `Move.NONE`.

The simplest Pac-Man player you can implement is

```
public Move chooseMove(Game game) {
    List<Move> legalMoves = game.getLegalPacManMoves();
    return legalMoves.get(0);
}
```

This quite obviously always picks the first legal move. Of course your task is to implement a better player than this and also not just to copy the provided simple player.

Note that the simple player computes an evaluation function for each legal move and selects a legal move by randomly sampling one of them based on the evaluation function value (without the random sampling the agent can easily get stuck in a loop). The evaluation function simply takes the game state the move would result in and returns a very small value for a losing state, 0 for a winning state, and otherwise combines the distance from the nearest dot and ghost to evaluate the result of the move. You can start by changing what this agent does to gain an understanding of the task, e.g. change how the next legal move is selected.

By the end of the first lab class you should at least be at this stage so that you can start to write your own agent. The second lab class is there to help you with implementing your own agent, but the tutor is not there to do your implementation. You can of course approach him with any questions or problems you are having with the task.

2 Pac-Man Search Agent

For the coursework you are expected to design and implement a Pac-Man search agent. Here are some clues on how to start designing an agent based on a search strategy.

Pac-Man is played in a maze *environment*, which is a 26×29 grid of squares. Each square is either free or contains an obstacle. If it is free, Pac-Man and the ghosts may occupy it. A free square may also contain a dot. Obstacle squares cannot be occupied and can not contain dots. The centre of the maze is the ghost-pen, which is occupied by a ghost until it is released, but otherwise is an obstacle square. At each time step t of the game, Pac-Man and the ghosts select a move $m_T \in M = \{\text{UP, DOWN, LEFT, RIGHT, NONE}\}$, of which only some are legal.

A *state* of the maze $s \in S$ consists of the location p of Pac-Man, four ghost locations $G = (g_1, g_2, g_3, g_4)$ and the locations of the remaining dots D , as well as the sequence of previous moves taken by each of the ghost $m = (m_1, m_2, m_3, m_4)$.

The *initial state* is s_0 in which Pac-Man is at $(12, 6)$, the ghosts g are all in the ghost-pen, and the set of dot locations D is initialised to all free positions, except for a pre-defined pattern of free squares without dots.

The *successor function*, $f : S \times M^5 \rightarrow S$ maps a game state and a set of moves for Pac-Man and the ghosts to a new state in which Pac-Man and the ghosts have each moved one square and the square now occupied by Pac-Man is cleared of its dot, if it had one. Pac-Man and the ghosts choose their moves *simultaneously*, i.e. each turn in the game is one application of f .

The *goal test* checks to see if D is empty, i.e. Pac-Man cleared the maze, or the square Pac-Man occupies is also occupied by a ghost, giving a losing condition.

The *path cost* is simple: every application of f has a cost of one, i.e. the more steps Pac-Man takes to clear the maze, the costlier this path is.

As a search problem, the Pac-Man agent will search over future states of the game offline, without affecting the state of the actual game. This is not a real-time game, so the ghosts wait for Pac-man to make its decision and vice versa. The `Game` class contains various static methods to support the search:

- `getNextState(State s, Move pacManMove, List<Move> ghostMoves)`
The successor function of the game, getting the new State resulting from all the moves.
- `getNextState(State s, Move pacManMove)`
A modified partial successor function, getting the new State resulting from just Pac-Man's move, assuming the ghosts do not move.
- `getNextState(State s, List<Move> ghostMoves)`
A different modified partial successor function, getting the new State resulting from all the ghost moves, assuming that Pac-Man does not move.
- `getProjectedStates(State s, Move pacManMove)`
Another modified successor function, getting the sequence of new States resulting from repeated application of Pac-Man's move, ending either at a final game state or when Pac-Man has to make another choice. Pac-Man is not allowed to reverse direction.
- `isFinal(State s)`
True iff the state is final (i.e. losing or winning).
- `isLosing(State s)`
True iff the state is losing for Pac-Man (i.e., Pac-Man runs into a ghost).
- `isWinning(State s)`
True iff the state is winning for Pac-Man (i.e. losing or winning).
- `getLegalPacManMoves(State s)`
Returns a List of Moves that are legal for Pac-Man in the give state.
- `getLegalGhostMoves(State s, int ghostIndex)`
This state returns a List of Moves that are legal for the ghost specified by the index in the give state.
- `getLegalCombinedGhostMoves(State s, int ghostIndex)`
This state returns a Set of combined ghost moves (represented as a `List<Move>`) that are legal for all the ghosts to make collectively in the give state).

With these functions you could for instance implement a depth-first search agent. In principle this would require to consider all possible moves of Pac-Man and the ghosts and we do not know which paths the ghosts will actually take. So one simplifying assumption could be to assume that the ghosts do not move. While this is false, we only have to decide upon the next move and can then consider the new position of the ghosts. Another issue is that it will take at least 245 moves to clear the maze, which is clearly too deep for an exhaustive search. A common approach to this is to use a depth limit, but for this we require an evaluation function for the game state. This evaluation function is crucial to the way a depth-first player behaves and is usually based on a weighted sum of some basic game state properties, e.g.:

- is the state a winning or losing state?
- number of dots left in the maze;

- distance between Pac-Man and the ghosts;
- distance between Pac-Man and the nearest remaining dots;
- average distance between Pac-Man and the remaining dots;
- are nearby ghosts headed towards Pac-Man or moving away?
- ...

If one considers the ghosts as agents, a minimax search strategy may be better. While we do not know what the ghosts do, we can make the usual minimax assumption that the ghosts will do the worst possible for Pac-Man. To avoid having to deal with too many ghosts, it may be a good idea to consider the ghosts as one adversary which chooses the combination of four moves in one go. In order to fit it in the minimax framework we also have to assume that Pac-Man and the ghosts are taking turns rather than moving simultaneously, and the ghosts actually always know what Pac-Man is doing. Note that the above functions support this by having getNextState functions using only Pac-Man's move and only the ghosts' move. Also, as usual, the search depth for minimax has to be limited and we need to choose again a good evaluation function.

But this is only the beginning for designing a good Pac-Man agent...