An agent that plays Pacman

Abbas Mehrabian Arian Khosravi

Department of Computer Engineering,
Sharif University of Technology

{mehrabian, ar_khosravi}@ce.sharif.edu

January 18, 2008

Abstract

A heuristic-based approach for developing a Pacman controller is described. The agent can score as high as an average human player, and the result is excellent compared to few similar works.

Keywords: Computational intelligence; Game controller; Pacman

1 Introduction

We have developed a software controller for Pacman, which is a classical real-time arcade game. We used "Microsoft Revenge of Arcade Ms. Pac-Man Version 1.0." The primary goal was to maximize the player's score. Our agent outperforms any other artificial agent for this version (see section 7).

Our approach has been described in this paper. In section 2 the reader gets familiar with the game and its objectives. We describe the algorithm in sections 3, 4, 5 and 6. Results have been described in section 7, and some ideas to improve the controller can be found in section 8.

2 About the game

The game is played on a rectangular maze with some *walls*. The player controls a yellow creature, the *Pacman*. In the following, the terms Pacman and the player will be considered the same. There are many white circles on the maze, called the *pills*. Pacman scores by eating the pills. When Pacman eats all of the pills, current level is finished and a new level with a new configuration will be started. There are some bad guys, the *ghosts*, who pursue Pacman and try to eat her. Pacman should avoid them. There are four big white circles when a level starts, called the *power pills*. When Pacman eats a power pill, the ghosts become blue for a few seconds, and she can eat them. The blue ghosts are called the *edibles*, and Pacman gets lots of scores by eating them. The eaten ghosts will then re-enter the scene from the center of the maze. There are some *snacks* in each level, which sometimes come and move randomly through the maze. Pacman gets score by eating them, too. The ghosts, edibles and snacks are being controlled by the computer.

We had these objectives in mind when designed our controller:

1. Avoid the ghosts.
2. Eat all of the pills in a level.
3. When edibles exist, try to eat as many of them as possible.
4. Eat the power pills at good times, i.e. when there are lots of ghosts around.

3 Modeling the game

We have used the Java programming language. To design any computer algorithm, a necessary step is to make the state space discrete. In order to do this, we have modeled the whole of the game state as an abstract 31×31 table, which we call the *game table*. The game table contains 961 *cells*.

There are mobile objects that interact in the game, namely Pacman, ghosts, edibles and snacks, we call them *game objects*. We have used an abstract Java class GameObject to model them. Game objects can be still, or move in one of four primary directions. Each game object in the scene occupies one cell of the game table, and has a certain direction at each moment.

A *connected set* in the screen, is a set of pixels with the same color. Every connected set may be the graphical representation of some game object. In fact, the game objects are abstractions of the connected sets.

The *distance* between two cells of the game table, is defined as the minimum number of cells a game object should pass through, in order to get from one cell to the other. Note that some of the cells contain walls, which are obstacles and the game objects cannot move through. This distance completely differs from the usual Euclidean distance, but is similar to the Manhattan distance.

4 Program outline

Here is the program's flow:

repeat forever
  capture the screen
  extract all connected sets from the screen
  for each extracted connected set cs*:*
    find a game object obj *such that* cs *is the representation of* obj
      update position of obj *in the game table*

  among the four neighbor cells of Pacman's current cell:
    find the cell best *such that* evaluate(best) *is maximum*
  press the key to go to best

The program captures the game window by a screen capture, decides about the best move and virtually presses the corresponding key. Although this screen-capture forces a 50% decrease in agent's performance, we had no alternatives. Moreover, since this is an infinite loop, it consumes CPU too much; so the performance is highly dependant on hardware's speed.

It can be seen that decision is made locally, that is, the agent makes decision only at this moment to go to one of the neighbors. This may look greedy, and one may ask why do not we predict future and devise long-term plans, or use min-max algorithm? There are mainly three reasons to use this simple algorithm:

1. The ghosts are hard to predict. Despite the fact that their movement is not completely random and have some general rules, there are several situations that they play quite unexpectedly. Hence we decided not to try to predict their movement.
2. The number of possible future game states is large, and it will take a lot of time to calculate deeper if we want to examine every possible movement. Even if we just consider Pacman and four ghosts, there would be at least 2^5 states reachable from each state, which is a big number. Recall that this is a real-time game.
3. Greedy algorithm is a simple one to implement, and its performance satisfied us.

Next we discuss how the function evaluate works. Because of the locality in making decisions, this function becomes the most important part of our algorithm. It gets a cell n, a possible future position, as input, computes certain parameters $p_1(n), p_2(n), \ldots, p_m(n)$ related to that cell, and returns the score of that position, which is a linear combination of these parameters:

$$score(n) = c_1 p_1(n) + c_2 p_2(n) + \ldots + c_m p_m(n).$$

Some of these parameters are:

- Distance of n to the nearest pill
- Distance of n to the nearest ghost
- Distance of n to the nearest power pill

The coefficients $c_1, c_2, \ldots, c_m$ (that are constant during the game) are very important, and we have tuned them manually to get a good performance. When a coefficient is large, it means that the corresponding parameter is important.

Coming back to the game objectives, which were described in section 1, one sees that in order to achieve each of these objectives, some parameters should be defined. In the next section, we will discuss the parameters that are related to first objective, i.e. escaping.

5 Details of escaping parameters

Avoiding the ghosts is the most important and challenging goal, hence it is worth to be described in more detail. Three parameters that were used are:

Distance to the nearest ghost This is the most obvious parameter that one could think of. The agent should try to maximize it. But it can be easily seen that this parameter is not perfect, and we should have a more universal point of view.

Safeness First we need a definition: Location f is called *available from* cell p, if Pacman can start from p and reach f before any ghost (supposing the worst circumstance, when all ghosts are also going to f). Now, *safeness* of a cell is the number of available locations from that cell. The more safeness a cell has, the more likely the agent is to go there.

Ghost energy The ghosts rarely reverse their direction, so we can use this property to predict them a little. Imagine that every ghost, spreads its "energy" throughout the game table such that most of the energy goes forward. Now, *energy* of a cell is the summation of the ghosts' energies at that cell, and illustrates the probability of the ghosts' presence in near future. Thus the agent prefers to go to a cell with less energy.

6 An improvement in the algorithm

Notice that the larger a coefficient c_i is, the more effect the parameter p_i would have on Pacman's move. But we certainly do not want a certain parameter to have a constant importance in all situations. For example, escaping parameters should affect more when the situation is risky. Therefore, the coefficients should not be constant during all phases of the game. This idea leaded us to the following algorithm:

There are four pre-defined situations: DANGEROUS, NORMAL, SAFE, and HUNTING. With each of them, a set of constants c'_1 to c'_m is associated. In each iteration, the agent first examines the current game state, and according to some parameters, classifies the current situation as one of these four situations; then sets the coefficients. For example, when a ghost is close to Pacman, the agent decides that situation is dangerous, and the coefficients are set to those associated with the dangerous situation. The remaining steps are same as the described algorithm.

7 Results and related work

The first version our controller, which was completed on 14 Sep. 2007, is comprised of about 2400 lines of Java code, has 4 different situations and 14 parameters. Its record is 21090 points, and gets 17300 points on average on authors' PC. The result is outstanding, since this is almost the same as performance of an average human player. We also have the best official record of an artificial agent that plays this version of Pacman (see [1]), which is 20640 points.

8 Conclusions and future work

We have developed a heuristic-based agent for Pacman game. Algorithm outline is simple, but implementation and the details are complicated. The result is satisfactory.

A good human player can get 40,000 points in the game. We believe that human's advantage over our controller is that he devises long-term plans, while our agent is unable to do so, and just considers its next move. On the other hand, the advantage of the agent over human is its ability to calculate very quickly and change its decision instantly.

However, we will have to change our program fundamentally if we want our agent to decide globally, which is costly. Thus we omitted this idea and are working on the next one. It would be nice if someone come up with a controller that can think about future and have plans.

There are 56 coefficients in the program (fourteen coefficients for each of the four situations), which we have tuned manually. We can use learning algorithms to tune these coefficients. In every learning algorithm, there should be some way to evaluate the current playing strategy. Here, the strategy is represented by the coefficients. Based on the observation that a good human outperforms our player, we have decided to use log-based learning algorithm to improve our agent. The idea of this algorithm is described next.

There is another program, called the *teacher*, that improves the agent's ability over time. We assume that there is a *human master* who plays the game very well. First, the master plays. At every moment teacher saves the parameters and master's move in a *log file*. Second, teacher reads the log file, and at every moment the agent is provided with the parameters; its move is logged, and is compared with master's move. The more the agent plays like the master, the better the strategy is. Teacher then tries to change the agent's coefficients to arrive at the optimal strategy.

Acknowledgement. The authors are indebted to Ramin Halavati for his support and ideas.

References

[1] S.M. Lucas, "Ms pac-man competition CEC 2007 results," University of Essex, Dec. 2007, Dec. 2007, `http://cswww.essex.ac.uk/staff/sml/pacman/CEC2007Results.html`.