Problem modelization
A*-based Pac-Man-like game solver

VALDIVIA David
HUANG Michel

MSc IASD

Université Paris-Dauphine, PSL

Course 2023-2024

1 Pac-Man

Pac-Man is an arcade game first released in Japan in 1980. The player controls Pac-Man, a character that looks like a pizza with a missing slice that serves as a mouth. The goal is to eat all the dots in the game while avoiding four ghosts.

With more than four decades of existence, Pac-Man is one of the most popular games in existence, appearing in multiple games and generating billions of dollars in revenue.

The purpose of this project is to create bots based on the A* algorithm. One of the bots takes the role of Pac-Man and tries to chase down all of the prizes. The other bots take the role of the enemies and try to chase down Pac-Man.

1.1 Gameplay

Pac-Man consists of a series of levels where the player is located in a maze (see fig. 1). The goal is to catch all of the dots while avoiding the ghosts.

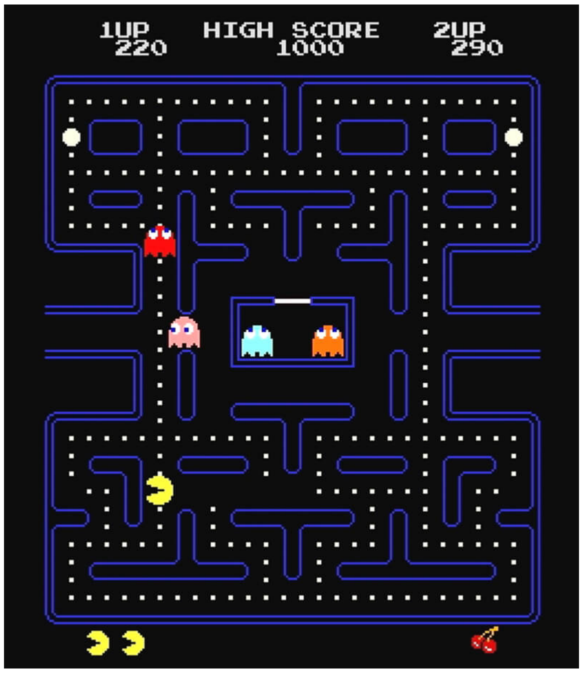


Figure 1: First level of Pac-Man

Pac-Man behavior

Pac-Man can only move horizontally and vertically. It can eat dots as well as fruits that give extra points and round balls that make the ghosts vulnerable. If the ghosts are vulnerable, Pac-Man can eat them and make them disappear. If not, the player loses when Pac-Man collides with a ghost.

Ghost behavior

In the beginning of the games, ghosts are located within a box called the **ghost-house**. One by one, they are released after the game starts. In the original game they have different personalities but for the purpose of this project we'll suppose that they all chase Pac-Man down in a straight-forward manner.

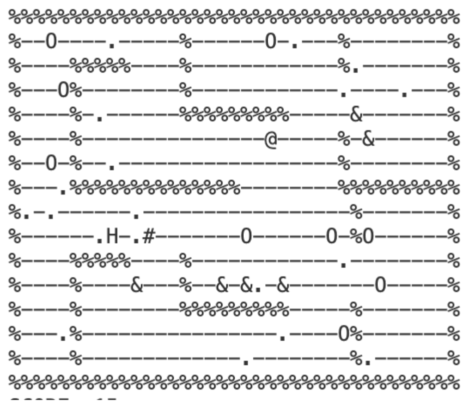
When Pac-Man eats a rounded ball, the ghosts become vulnerable. To heal, they go back to the ghost-house or wait a certain amount of time to heal automatically.

Game evolution

If Pac-Man eats all of the dots, the player advances to the next level. If a ghost and Pac-Man collide, Pac-Man loses a life and the level restarts. If no more lifes are left, the game ends.

2 Game implementation

For this project, we implemented a Python terminal-based Pac-Man like game. The code is available in `https://github.com/zaquastier/pacman_a_star`. An example of a maze is presented in fig. 2.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%--0----.-----%------0-.---%--------%
%----%%%%%----%------------%.-------%
%---0%--------%------------.----.---%
%----%-.------%%%%%%%%%-----&-------%
%----%---------------@-----%-&------%
%--0-%--.------------------%--------%
%---.%%%%%%%%%%%%%%--------%%%%%%%%%%
%.-.------.-----------------%-------%
%------.H-.#-------0------0-%0------%
%----%%%%%----%------------.--------%
%----%----&---%--&-&.-&-------0-----%
%----%--------%%%%%%%%%-----%-------%
%---.%----------------.----0%-------%
%----%-------------.--------%.------%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 2: Terminal-based Pac-Man

2.1 Entities

In the maze, one can find the following entities:

- @: the **player**. It represents Pac-Man. Can move horizontally or vertically, can eat dots and rounded balls and collide with ghosts.
- #: an **enemy**. It represents a ghost. It moves horizontally and vertically and chases down the player.
- **.** : a **prize**. It represents a dot. If the player eats all the dots the game is won. Eating a prize provides +1 of score.
- O: a **power**. It represents a rounded ball. If the player eats one of these, the enemies become vulnerable. Eating a power provides +5 of score.
- §: a **scared enemy**. When the player eats a power, the enemies turn into scared enemies. The player can eat these. Eating a scared enemy provides +10 of score.
- H: the **house**. When enemies are scared, they go here to turn back to normal.
- %: a **wall**. It delimits the map. The player cannot move towards a wall.

Note: we implement scores only to make the game more attractive. They serve no purpose otherwise.

2.2 Gameplay

The game runs on a small number of frames-per-second. The player moves at each frame while the enemies only move every N frames, where N is hard-coded within the game. We take $N = 4$.

When the player eats all of the prizes, it goes to the next level. When all of the levels are completed, the game ends in victory.

If the player collides with an enemy, the player loses a life. If all lifes are lost, the game ends.

2.3 What we didn't implement

Due to time constraints, some aspects of the original game were left apart:

- In the original game, the house is a space rather than a single point location. It was easier to implement the house as a single point to turn a ghost back to normal since it only requires to detect a collision rather than knowing if the ghost is within the space of the house.
- We left aside teleporting. In the original game, there are some tunnels that teleport you from one side of the map to the other.
- No fruits appear in our version of the game since score doesn't matter.

3 Player bot

The player must chase down all of the prizes while avoiding enemies. It can also eat powers that allow them to chase an enemy, thus reducing the number of enemies.

3.1 Selecting what to eat

Since the player can eat a different number of entities of different classes, we must decide what object to eat first.

The first idea is to calculate the euclidean distance between the player and every object they can eat, and select the object with minimal distance by using:

$$d(player, object) = (x_{player} - x_{object})^2 + (y_{player} - y_{object})^2$$

This however is not optimal. In fig. 3, a case example shows that the closest prize in euclidean terms is not always the easiest to get.

```
%%%%%%%%%%%%%%%%%
%--.------@-#-.%%
%--------------%%
%%%%%%%%%%%%%%%%%
```

Figure 3: Geometrically, the closest prize (.) is the one on the right. However, the nearby enemy # makes it hard for the player @ to go for it

For this, we add a penalization term:

$$d_{penalization}(player, object) = d(player, object) + \frac{\alpha}{\sum_{enemy} d(object, enemy) + \epsilon}$$

Where we take $\alpha = 20$ in our code and $\epsilon = 10^{-6}$. Now, we consider that objects closer to enemies are far away from the player. In the case of fig. 3, the player would select the prize on the left.

3.2 Chasing scared enemies

It is a good idea to chase down scared enemies. Indeed, getting rid of them simplifies a lot the task of getting all of the prizes. Thus, we redefine the search for the prize to look in section 3.1:

$$d_{optimal}(player, object) = \begin{cases} d_{penalized}(player, object)/10, & \text{if object is a scared enemy} \\ d_{penalized}(player, object), & \text{otherwise} \end{cases}$$

This way, the player will try chasing down scared enemies instead of regular prizes.

3.3 Path to selected item

Now, we implement our algorithm to chase down the items. For this we use A* by minimizing:

$$f(n) = g(n) + h(n)$$

Where $g(n)$ is the cost of the path starting from the player to node n and:

$$h(n) = d(n, prize) + \frac{\alpha}{\sum_{enemy} d(n, enemy) + \epsilon}$$

ie.

$$h(n) = d_{penalization}(n, prize)$$

With $\alpha = 20$ and $\epsilon = 10^{-6}$. This gives a path where the player avoids the enemies by circling around them instead of going straight to the prize.

One issue with this algorithm is that it gives a static path from the player to an item. However, since enemies move, they can get along the way, see fig. 4.

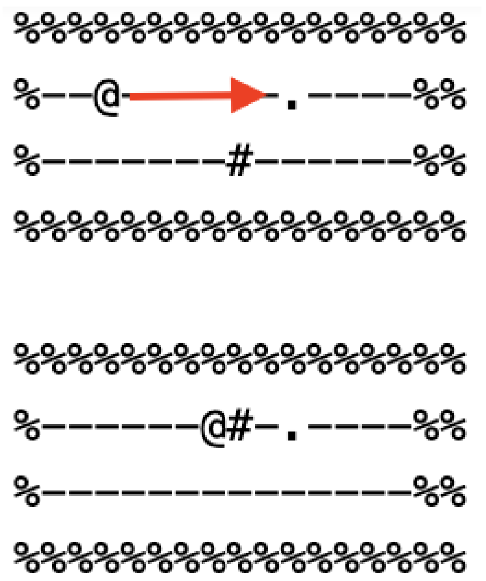


Figure 4: Top: frame 1. Bottom: frame 2. When using a static path, enemies can get in the way of the player

One way to circumvent this is for each frame, to calculate the object to chase, get the path with A* and **applying only the first move of this path**. This way, unexpected collisions are avoided.

Note: this is in no way optimal. One way to optimize this is to follow the path provided by A until enemies move, and only then recalculate the prize to chase and the path to that prize.*

4 Enemies bot

The enemies try to chase down the player. When they are scared, they go to their house marked by an H. The bots work in a similar way than the player bot.

4.1 What to chase

If an enemy is not scared, it will chase down the player using A* with the following heuristic:

$$h(n) = d(n, player)$$

If the enemy is scared, it will go to the house using A* with the following heuristic:

$$h(n) = d(n, house) + \frac{\alpha}{d(n, player) + \epsilon}$$

5 Conclusion

In this project we implemented different versions of A*.

First, we help a player chase an item by avoiding enemies. The item to chase is selected using its distance to the player as well as its distance relative to enemies. We also privilege chasing scared enemies to make the task of getting rid of the prizes easier.

Then, we use a classic A* algorithm to help enemies chase the player. If an enemy is scared, we make them go to their house while avoiding the player (who wants to eat them).

Note that we select the item to chase and calculate A* each frame to avoid unwanted collisions between the player and the enemies and between scared enemies and the player. This is not optimal but works for this exploratory project.