INFO-H410 - Techniques of artificial intelligence

June 3, 2022

Project report 2021-2022 Deghorain Axelle, Bermudez Loïc, Loems Edwige

1 Abstract

Our project involves implementing a simplified version of the arcade game **PacMan**. In this game, the objective is to maneuver PacMan through a maze so he can eat all the pellets while avoiding the ghosts that pursue him. If PacMan eats all the pellets, the level is complete. If a ghost catches him beforehand, PacMan is defeated, and his score is proportional to the total number of pellets he has eaten.

In our game implementation, both the ghosts and Pacman use search algorithms to navigate the maze. We also simplified the game by limiting the numbers of ghosts and level to two.

2 Introduction

The following argument is based on the third theorithical presentation of the course {cite}`Wiggins`.

2.1 Search algorithms

The search is the process of considering action sequences within a problem formulation. The search algorithm takes a problem as input (formulated as a set of goals and actions to be applied) and returns a solution in the form of an action sequence.

The search algorithm framework is illustrated in figure 1. The Queuing-Fn function determines what kind of search we are doing. The initial state and the operators define the start-state. The goal test determines when the goal state is reached (boolean functions over states, true if the input is the goal state). The function returns a sequence of operators (describe actions in terms of states reached by applying them) leading from start to goal state. The action sequences are generated using an agenda. The agenda starts at the initial state, checks if the goal state is reached, and considers other states by applying the operators to the present state to generate a new set of states (expand function) until the goal state is reached. The state space can be structured into a tree where the actions corresponds to arcs and states correspond to nodes.

```
function AgendaSearch( start-state, actions, goal-test )
                                        returns action-sequence or failure
    static: seq, an action sequence, initially empty
            agenda, a state sequence, initially contains start-state
    loop do
            if Empty(agenda) then return failure
            if goal-test(First(agenda)) then return seq
            agenda ← Queuing-Fn(Rest(agenda), Expand(First(agenda), actions))
            seq ← Append( seq, Action(First(agenda)))
    end loop
```

Figure 1. Search algorithm framework (Agenda-based search)

2.2 Breadth-First-Search (BFS)

For our project, we implemented a **Breadth-First-Search (BFS)** to determine the action of the ghosts. In the BFS algorithms, the tree is constructed by going across it repeatedly, replacing each node with its children. The BFS has the advantage of finding the solution if there are one and guarantees to find the shortest action sequence. It is not suited for large search spaces, which was not our case.

In this algorithm, the Queuing-Fn function removes the current node from the agenda and appends its children to the back of the agenda. The agenda is rotating, constantly dealing with its first item.

We also considered the Depth-First-Search (DFS). This algorithm constructs the tree by exploring each branch until it fails before backtracking to another branch. It is not suitable in our case because the search for a path between two positions in our grid is unlimited.

2.3 Heuristic Search Strategies

To determine each action of the PacMan, we needed a strategy that does not strictly search for the shortest path to the pellets but also consider the movements of the ghosts.

The heuristic search refers to a search strategy that attempts to optimize a problem by iteratively improving the solution based on a given heuristic function or a cost measure {cite}`Russell_artificial_2003`.

We implemented the algorithm A*. The heuristic function (f) is defined by :

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

- g is the cost-so-far function
- h is the heuristic

The A* algorithm has the adventage of using a heuristic that is admissible (never over-estimate the distance to the closest solution). A* is also optimal (we are guaranteed a solution if there is one, the fewest possible nodes will be expanded, given the problem formulation).

3 Methods

The methods describe the implementation of the two ghosts and the Pacman using Breadth-first and A* algorithms.

The implementation of the graphic user interface (GUI) is inspired by the implementation of the Snake game available on the ai-book website {cite}`AI_en_pratique_avec_python`.

The structure of the mazes is inspired by levels present in the original Pacman game. In figure 2, we describe the content of each file necessary to execute the game and the class used.

File	Class	description	ARGs
main.py		Launch the game, with the given arguments	
game.py		Control the game	
	Game	The logic of the game and all the variable use for the game	**grid** = current grid of the game, **pacman_alive** = boolean , **pacman** = instance of Pacman class, **ghosts** = instance of Ghost class, **score**, **nbr_item** = initial number of items.
	GUIGame	The GUI of the game, the parent class is Game	
pacman.py	Pacman	The controller of the pacman movements	**pos** = current position, **direction** = current direction, **search** = instance of Search
ghost.py	Ghost	The controller of the ghosts i.e the position, the move,..	**ghosts** = dictionnary of the ghosts positions (key = ghost number, value = current position), **search** = instance of Search Class

File	Class	description	ARGs
search.py		The differents algorithms for search the good path and the necessary to manage	
	Node	Definitition of the node needed for the graph search	**movement** = mouvement to go on, **position**, **prev** = the precedent node , **pos_acestor** = list of position visited in the path to this node. If it is a node for A* there is also, **g** , **h** and **f** (f= g-h)
	Search	the different functions for search the good path like the A* and BFS function	**node_pacman** (jut for the ghosts)
utils.py		All the variables needed by the different Class i.e the Grid base, the different CHAR and images of the elements,the movement,...	

Table 1. Description of the files and the class used for the pacman game

On the class Search, we implemented the Breadth-First algorithm. The Breadt-First algorithm is used both by the Ghosts and Pacman. The nodes are the possible position in the grid game. To reduce the number of nodes visited, the position that are occupied by walls are not considered. An attribute ancestor_position is present in the class Node to prevent the return in the precent position and avoid loop.

The Breadth-First algorithm goes through the tree of possible movements repeatedly, replacing each node with its children. The function removes the current node from the agenda and appends its children to the back of the agenda. This methods ensurs to obtain the shortest path.

The Breadth-First algorithm returns the target node. To retrieve the path to get there, you have to do some backtracking. This is possible thanks to the Prev argument of the returned node which points to the node visited just before.

3.1 Movement of the ghosts

Pacman is purchased by two ghosts whose movements are determined by the same Breadth-First algorithm. The first ghost set its goal position to the current position of Pacman (targets Pacman's position).

The second ghost target a position three grid cases in front of Pacman and in its direction. However, he uses the position of the first ghost stocked in the class Search to calculte its own target position. If the considered new position is occupied by a wall, a new direction will be chosen to obtain a position three positions forward the Pacman).

3.2 Movement of Pacman

The next movement of Pacman is chosen with an A* algorithm. We used the same Breath-First search algorithm as for the ghost and added an heuristic.

The objective of Pacman is to get closer to the pellets while avoiding the ghosts. We want to minimize the heuristic function (f() = g() - h()) by minimizing Pacman's distance with the pellets (g()) while maximizing its distance with the ghosts (h()).

The state of the game is constantly evolving (ghosts and Pacman positions). Instead of searching for a succession of steps approaching pacman from its objective, we will evaluate the possibility of movement in the boxes adjacent to Pacman.

The function g() can be calculated by finding the shortest distance between Pacman and the pellets. As our grid has walls, we used the Breadth-First algorithm to calculate the real shortest path between Pacman and the pellets (and ghosts).

The same method is used to calculate the function h. However, if the distance between the considered position and the ghost is equal to 1, we change the value of h to -100. This allows us to increase the cost of the move when the position considered by the Pacman is next to a ghost.

4 Discussion

We initially limited the implementation of the game to only one level. We added a second level to test and compare the performance of Pacman in different mazes.

In level 1, Pacman manages to eat all the pellets and win with a score of 32 (figure 3). This situation is observed every time we launch the level. However, in level 2, Pacman always loses with a score of around 44 (figure 4). We explain this difference by the presence of dead ends on the level 2 map.

In the current A* implementation of Pacman, the only objective of Pacman is to get closer to the pellets while getting away from the ghosts. When the new positions are considered for Pacman's next move from the current position, the position will not be evaluated if a wall fills it. This strategy allows Pacman only to consider paths that are accessible but not to avoid dead-ends.

This issue could be resolved in several ways. We could consider the distance between the Pacman and a dead-end in assigning a cost to a path. This solution reduces the risk of losing but keeps the Pacman away from the pellets present inside the dead-end.

We could also add special pellets to the level. In the original game, Pacman has the ability to eat blue pellets that temporally give him the ability to eat ghosts.

Figure 2. Pacman Level 1

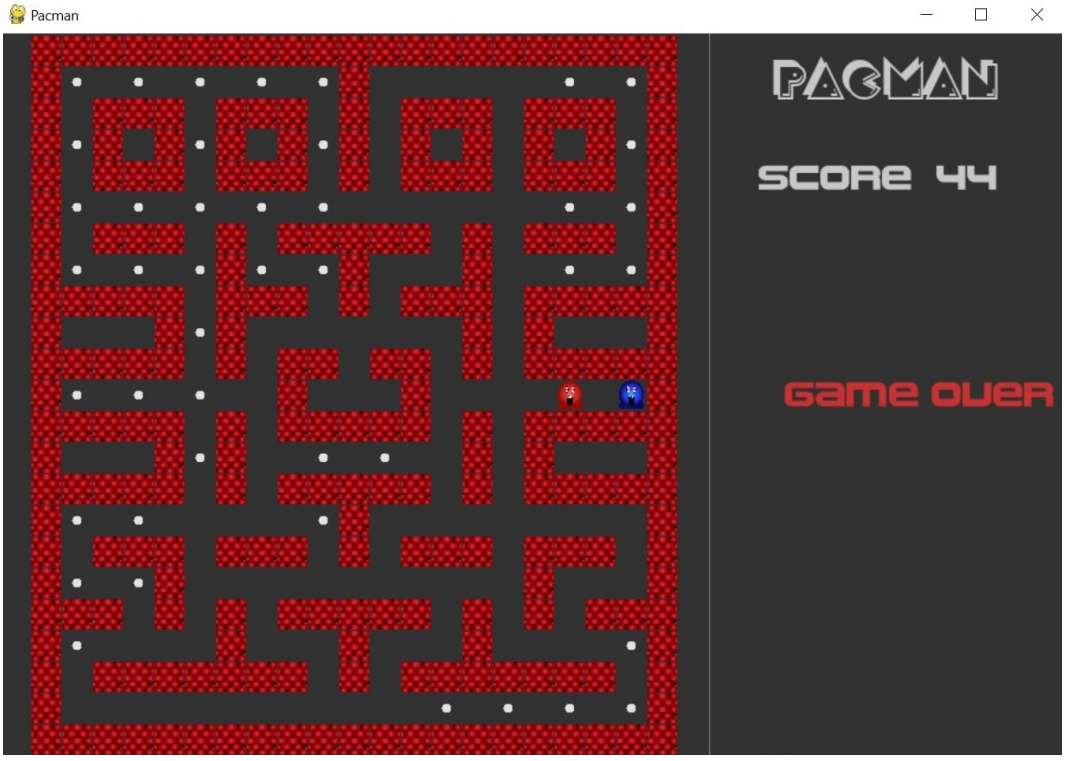

Figure 3. Pacman Level 2

5 Conclusion

During this project, we implemented an AI with the minimum requirement of the game Pacman. The behavior of the ghosts is satisfactory and expected. With our implementation of A*, Pacman could maneuver through a simple maze. However, the implementation of a second level highlighted the limitations of our algorithm.

6 References

[AI_en_pratique_avec_python]. L'ia en pratique avec python. URL: https://iridia.ulb.ac.be/~khasselm/ai-book/ (visited on 2022-06-02).

[Russell_artificial_2003]. Stuart J. Russell and Peter Norvig. Artificial intelligence: a modern approach. Prentice Hall series in artificial intelligence. Prentice Hall/Pearson Education, Upper Saddle River, N.J, 2nd ed edition, 2003. ISBN 9780137903955.

[Wiggins]. Geraint Wiggins. Problem solving and search.

7 Appendix 1 : Instructions for using the PacMan project

7.1 Installation of Poetry

To install Poetry, refer to the instruction information to install Poetry under your operating system

Under Linux, we used the following command line :

[1]:
```
curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/master/
↪get-poetry.py | python -
```

Under Windows powershell instruction :

[ ]:
```
(Invoke-WebRequest -Uri https://raw.githubusercontent.com/python-poetry/poetry/
↪master/get-poetry.py -UseBasicParsing).Content | python -
```

When installing poetry under the windows operating system, we ran into the issue python cannot be found because the path to our python interpreters were not referenced in our environmental variables. The issue has been solved by adding the following lines to our system variables.

[ ]:
```
C:\Users\*username*\AppData\Local\Programs\Python\Python310
C:\Users\*username*\AppData\Local\Programs\Python\Python310\scripts
```

Under both systems, the poetry bin directory must be added to the $PATH environment variable, either automatically or manually.

7.2 Installation of the project dependencies

Access the game folder :

[ ]:
```
cd Pacman
```

Install the project dependencies :

[ ]:
```
poetry install
```

7.3 Launch the game in the virtual environment

To launch the game in the player mode :

[2]:
```
poetry run python main.py -p
```

The player mode allows the manual maneuver of Pacman through the game. Use the direction key to move the Pacman and start the game. Once level one is complete, level two will automatically launch. Press any direction key to start the game. When level two is complete, the game will automatically close.

To launch the game with Pacman A* :

[ ]:
```
poetry run python main.py -x
```

The game will automatically start, and when level one is over, it will automatically skip to the next level. Once level two is complete, the game will automatically close.