# PACMAN - CTF

## 1. Introduction

The practical work consists of developing intelligent agents for a variation
competitive game Pac-Man. In this game, we control two agents in a maze
with two territories. The two agents must collaborate to defend their territory and
attack opponents' territory, taking as many pacdots as possible from the
enemy.

## 2 Classification of the problem

According to the textbook, the problem can be classified as follows:

• Partially observable, since the agent has no knowledge
the complete state of the environment. More specifically, the agent does not know the
position of your enemies, unless they are close enough.

• Multi-agent, since a certain agent will compete with two enemies for
a higher score, and will have an ally to help you maximize your
punctuation. The environment is multi-competitive and multi-cooperative.

• Deterministic, since the next state of the environment can be predicted
through the current state and the action chosen by the agent. It is worth remembering that, in
definition of deterministic environments used by the textbook, we ignore the
uncertainties that arise only through the actions of other agents. The environment
could be considered stochastic if these uncertainties were considered.

• Sequential, since the choice of a certain action at a given moment affects
all future decisions.

• Semi-dynamic, since the agent has a fixed time (1 second) for
make a decision and, during that time, the environment does not change. Of that
way, the agent knows that the environment does not change while he is deliberating,
but you have to worry about the passage of time.

• Discreet, since the environment is discreet and there is a finite number of
different states and actions in the problem.

---

## 3 Modeling of agents

In this section, the strategies and algorithms used in the implementation will be described
of the two agents developed. One of the agents was made to attack the territory
enemy and eat pacdots, while the second was done to defend their own
Pacdots.

## 3.1 Offensive Agent

The offensive agent uses Monte Carlo simulations to evaluate each possible action
at any given time. The action chosen for execution is the one that was most
well evaluated.
Normally, during a simulation, the game is played randomly by all
agents until it ends, and the result will be, for example, the number
total victories obtained after the execution of several simulations. In our Pac-Man
competitive, however, this approach is not viable: the number of actions taken
until the end of the game it can be very high and we don't have constant visibility of our
opponent to simulate your actions. The solution to the first of these problems is
widely used in real-time applications, and consists of performing simulations
only to a certain depth d, that is, only d actions will be simulated
of each agent. After this partial simulation, it is necessary to use a function of
evaluation in the last obtained state. The solution adopted for the second problem was
fix the actions of all other agents as STOP. Thus, in practice,
it will be necessary to simulate only the actions of a single agent (the one who is
what action to take). This solution, although quite simple, shows results
promising if combined with a good evaluation function.
Two important parameters used in the Monte Carlo simulation implemented
the depth of the simulation to be used and the number of simulations used
to assess a certain state. Generally, the higher the value of these parameters,
the better the results obtained. Thus, it is important to find values
for these parameters that allow good results and leave the execution time
viable.
The random simulation performed by the agent needs some care so that
be more efficient. It is not interesting that the agent always chooses randomly
between all possible actions at each simulation step. This can lead the agent to
going back and forth, or even standing still for a few moments. Obviously,
most of the time, it is not interesting that the agent is stopped or going and coming back
between the same two positions during a simulation. Thus, in the execution
of the random simulation, the agent is forbidden to stand still (execute the STOP action)
or to reverse its direction (take the action in the opposite direction to the current direction).
Note that the agent can perform these actions during the game. What is forbidden is yours
use during random simulations.
The evaluation function used was based on that present in BaselineAgents.
It consists of the linear combination of features and weights associated with the features. At
features considered are the following:
• Score of the state: takes the score of the final state of the simulation. The goal
this feature is to maximize the score obtained by the agent.

• Distance to the nearest pacdot: the distance from the agent to the nearest pacdot next in the final state of the simulation. The purpose of this feature is also maximize the possible score obtained. If, during the simulations, there are two final states with the same score, the one where the agent is closest to another pacdot is best rated.

• Distance to the nearest enemy: the distance from the agent to the enemy closest to the end of the simulation. This feature allows the agent to escape the enemy if you are being chased, while trying to eat more pacdots.

• Pacman: this feature indicates whether the agent is a ghost or a pacman. She is used only in a very specific situation, where it can happen for the agent to return to the defense field to defend himself and not get more return to attack because the defending enemy is on the edge watching the agent (the agent finds it advantageous to stay alive in his territory). In this case, the agent starts to prioritize the fact of being a pacman, and stops defending himself in the form of ghost, prioritizing the attack.

The weights of the features are used dynamically: normally, the largest weight goes to the feature referring to the score, a lower weight is given to the distance to enemy, a negative weight is given to the distance to the nearest pacdot (the more closer, better) and a null weight is given to the Pacman feature. If the opponent is in the scared state, the weight given to the distance to the enemy is zeroed. If our agent get stuck in the defense field, the Pacman feature gains a high weight.

Finally, one last feature was added to the offending agent: the ability to city to avoid some alleys without pacdots. Before evaluating the possible actions, it is pre-processing was performed on the action. It is expanded to a depth 5 and, if all paths expanded from that action end in an alley without pacdots, then the agent discards the action. This feature is particularly important at the end of the game, when there are few pacdots left and going into an alley can be quite bad, cornering the agent.

## 3.2 Defensive Agent

The defensive agent is quite simple. It works by defining target positions and moving looking towards them, as well as the defending agent for SimpleTeam. The definition of the target, however, is a little more elaborate. The agent sets targets in order to execute high-level strategies. These strategies are: patrolling central points

from the map, check the position where pacdot disappeared, chase opponents and watch pacdots.
A brief explanation of each strategy follows, explaining when it is chosen:
• Patrol central points: this is the strategy implemented by the agent fensitive most of the time. It consists of choosing a point on the edge the territories of the two teams and move to that point. We call these patrol points. The patrol points for which the agent will be able to move are found during the pre-processing time.
To choose the point to be visited, we calculated some probabilities.
We associate, at each patrol point, a probability that corresponds to ` where the agent chooses to go to that position. The probability is calculated based on the inverse of the distance of the closest pacdot to the position of the point patrol in question. These values are normalized so that the sum probabilities is 1. In this way, the defending agent is more likely to move to patrol points with a nearby pacdot, since the opponent will probably try to catch these pacdots first. Whenever the opponent eats a pacdot, the odds are recalculated.

• Check position where pacdot disappeared: the agent implemented checks, each iteration of the game, if any patdot from your territory has disappeared, a since we have information on the positions of all our pacdots. Case has disappeared, the agent will move to the position of the pacdot that disappeared. Thus, it is expected that if the enemy goes through the patrol of the defending agent, it is possible to quickly identify that our territory was invaded, as well as estimating the attacker's position.

• Pursue opponent: during the patrol and verification previously described, if the agent sees an enemy, he starts chasing him.

• Watch pacdots: at the end of the game, when the number of pacdots to be defended is small, it is more advantageous for the defender to move between these pacdots instead of patrolling the central area of the map. That way, when we have 4 or less pacdots, the agent starts to walk randomly from one to another.

## 4 Complexity analysis

In this session, the complexity of the two agents implemented will be analyzed.

### 4.1 Offensive Agent

In terms of time, the offending agent's most expensive operation is to assess possible actions that will be taken using Monte Carlo simulations. Be the number of actions the agent can choose. For each action, n

random depth simulations d. The evaluation of a state at the end of the
is very simple, and the biggest cost of this operation is to find the pacdot
with less distance to the current position of the agent. Considering that there are p pacdots
in the opponent's field, this evaluation is of the order of O (p), considering that the
getMazeDistance () method provided has constant cost. Thus, the cost
to make a decision is on the order of O (adnp). However, a is a fixed number (the
agent can choose, in the worst case, between 5 actions), and eden are also fixed before
execution. Since c = 5 ∗ d ∗ n, a constant, the temporal complexity will be of the
order of O (cp), or O (p).

The spatial complexity of the offensive agent is constant. During the simulations
and evaluations, only constants are stored and each random simulation used
only uses one copy of the gamestate. Thus, considering that the gamestate
occupy a constant space, the offensive agent has an order of spatial complexity
Constant.

## 4.2 Defensive Agent

As can be seen in the description of the defensive agent, most of his
operations consists of defining targets. Such operations are quite simple and have,
overall, constant cost in terms of time. The defender's most expensive operation
is to calculate and recalculate the probabilities of visiting the patrol points. To calculate
such probability, it is necessary to find, for each patrol point, the pacdot
closer. X being the number of patrol points (which varies depending on
the size of the map or the number of walls in the meeting area between the two
territories) and the number of pacdots in our territory, the temporal complexity
of the defensive agent is, in the worst case, O (xp).
The spatial complexity of the offending agent is O (p + c), with p being the number of
pacdots that will be defended with the c number of patrol points in the center of the
map. This is because the defensive agent always stores the last observation of the
pacdots to be defended (the position of each of the p pacdots) and a dictionary with
the positions of each of the c patrol points, associated with the probabilities
to choose each one.