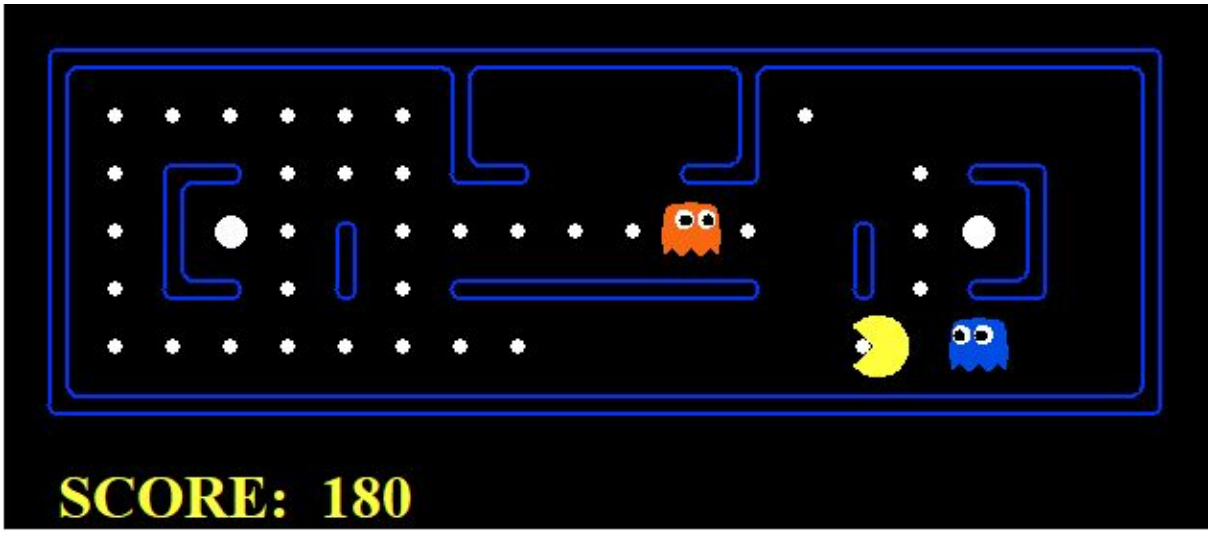
Obligatorio Sistemas Multiagente 2020

Misión: vencer a Pac-Man

1 Introducción

El obligatorio tiene como objetivo la aplicación de diferentes técnicas de planificación y aprendizaje de estrategias en un entorno competitivo con múltiples agentes que interactúan.



Para esto se utilizará una versión del juego ***Pac-Man*** especialmente diseñada para la aplicación de algoritmos de inteligencia artificial, creada por la universidad de Berkeley, a la que se le han realizado modificaciones solicitadas por la Ghost Co., con el objetivo principal de habilitar el entrenamiento de fantasmas inteligentes capaces de detener el avance del diabólico Pac-Man.

> ***Pac-Man*** *(パックマン Pakku Man[2]), también conocido como* ***Comecocos*** *en España, es un videojuego arcade creado por el diseñador de videojuegos Toru Iwatani de la empresa Namco (basado supuestamente en la forma de una pizza con un trozo faltante), y distribuido por Midway Games al mercado estadounidense a principios de los años 1980.*
> *[..]*
> *El protagonista del videojuego Pac-Man es un círculo amarillo al que le falta un sector, por lo que parece tener boca. Aparece en laberintos donde debe comer puntos pequeños (llamados «Pac-dots» en inglés), puntos mayores y otros premios con forma de frutas y otros objetos. El objetivo del personaje es comer todos los puntos de la pantalla, momento en el que se pasa al siguiente nivel o pantalla. Sin embargo, cuatro fantasmas o monstruos, Shadow (Blinky), Speedy (Pinky), Bashful (Inky) y Pokey (Clyde), recorren el laberinto para intentar capturar a Pac-Man.*
>
> Fuente: Wikipedia (https://es.wikipedia.org/wiki/Pac-Man)

2 El juego

Existen múltiples **mapas** (*layouts*), con diferentes niveles de dificultad. En estos se pueden encontrar tres tipos de elementos:

- *Food*: otorga pac-coins a Pac-Man al consumirlas
- *Capsules*: permiten a Pac-Man eliminar fantasmas (y ganar pac-coins) por un corto periodo de tiempo.
- *Ghosts*: fantasmas que deambulan por el ambiente

El juego funciona en un esquema por **turnos**, donde cada agente (Pac-Man y ghosts), realiza una acción. Pac-Man es agente **0**. La misión de Pac-Man es consumir toda la comida y las cápsulas del mapa. La Ghost Co. destina G ≥ 2 ghosts inteligentes, identificados como agentes **1, …, G** y otros R ≥ 2 ghosts random, numerados **G+1, …, G+R**, cuya misión es eliminar a Pac-Man.

Los agentes se mueven en el mapa de acuerdo a un conjunto de acciones posibles (derecha, izquierda, arriba, abajo). Luego de ejecutar una acción Pac-Man acumula pac-coins si consume comida o cápsulas, o si elimina fantasmas, y se le descuentan en caso contrario. Pac-Man tiene una autorización de sobregiro de hasta **C** pac-coins. Por su lado, toda acción de los fantasmas les cuesta pac-coins.

El juego **termina** si se cumple alguna de las siguientes condiciones:

- Pac-Man es eliminado por un ghost. Gana el ghost que elimina a Pac-Man.
- Pac-Man consume toda la comida y cápsulas del mapa. Gana Pac-Man.
- Pac-Man agota su crédito. Pierden todos.

La **utilidad** de los agentes al final del juego es la siguiente:

- Si gana, Pac-Man recibe el saldo (eventualmente negativo) de pac-coins acumulados. Caso contrario, se le resta al saldo una multa **M**.
- Cada ghost recibe el crédito de pac-coins acumulado, a lo que se le suma, en caso de ganar, un bonus **B.**

3 Tarea

3.1 MaxN

Se debe implementar el algoritmo **MaxN**, con poda por *profundidad*.[1] El parámetro **max_depth** determina la cantidad máxima de acciones realizadas[2]. Se sugiere realizar poda poco profunda, por ejemplo, **max_depth = 2**.

[1] Esto no es “shallow pruning”.
[2] Se decrementa cada vez que juega un jugador

3.2 Estimación del valor de los estados

Al llegar a la profundidad definida, se debe utilizar técnicas de estimación de valor de estados mediante rollouts para estimar el valor de los estados alcanzados. Se pide implementar los dos métodos vistos en clase:

1. Roll-out
2. Monte Carlo Tree Search

3.3 Funciones de evaluación

Los métodos de estimación basados en Monte Carlo normalmente requieren construir un episodio completo, es decir, que termine en un estado final. En este juego, esto puede tener un costo prohibitivo en tiempo computacional.

Por esta razón, se pide implementarlos considerando la utilización de una función de evaluación heurística para estimar el valor en un *prefijo* de episodio, de longitud definida por el parámetro **max_unroll_depth,** en caso de no llegar a un estado final.

La función de evaluación debe de realizarse mediante una heurística que recibe el estado del juego, lo procesa utilizando la función **process_state** y luego hace computos con ese resultado. La función **process_state** recibe la información completa del estado del juego y la ajusta para dar una visión acotada al agente (habilitando a representaciones de estados más concisas).

3.4 Opcional: Estimación de la función de evaluación mediante DQN

De modo opcional se puede realizar la estimación del valor del estado al final de cada prefijo de episodio con técnicas paramétricas de estimación del valor (por ejemplo, DQN). Ésta se deberá implementar en el archivo **qlearning.py** (dentro de la carpeta obligatorio) y ser utilizada dentro de la función **evaluationFunction** en el agente **maxN**.

4 A entregar

Se deberá entregar el código que implementa lo solicitado (archivos **.py** de la carpeta **entregables**), junto con un informe en formato pdf. El código tiene que ejecutar sin errores. Se realizará verificación de autoría. Los grupos son de hasta **tres** estudiantes.

4.1 Entregable opcional:

En caso de haber realizado la tarea opcional, entregar un archivo **Qlearner###.pkl** (comprimido en un zip de hasta 40MB) con el learner entrenado (con la cantidad de iteraciones indicada).

Anexo

En esta sección se describen brevemente la estructura y las principales funciones del código base provisto, y las funciones a implementar.[3]

Se provee un archivo **main.py** qué contiene una ejecución de ejemplo. Al mismo nivel hay tres carpetas: **game_logic, layouts** y **entregables**:

- La primera contiene como su nombre lo dice, la lógica relacionada al juego, agentes, reglas del juego, y gráficos.
- La segunda contiene archivos qué definen los layouts o mapas del juego. Son libres de crear los suyos.
- La tercera contiene y debe contener todo el código qué se entregará. A la hora de ser presentada la letra, contiene la clase **MaxNAgent**, que presenta el esqueleto a completar.

Son libres de agregar los archivos de código qué se consideren necesarios.

Dentro de **MaxNAgent** están las siguientes funciones:

`__init__(index, max_depth, unroll_type, max_unroll_depth, number_of_unrolls, view_distance)`

- Parámetros: número de agente, profundidad máxima de maxN, tipo de unrolling a realizar (“MC” - Monte Carlo o “MCTS” - Monte Carlo Tree Search), largo máximo de unrolling, cantidad de unrolls por estado a evaluar y distancia máxima de visión para la función heurística.
- Retorna: nuevo agente maxN

`getAction(gameState)`

- Parámetro: estado del juego para el cual hay que calcular la acción a realizar
- Retorna: la acción a realizar

`evaluationFunction(gameState)`

- Parámetro: estado del juego para el cual hay que calcular el valor
- Retorna: valor calculado

`maxN(gameState, agentIndex, depth)`

- Parámetros: estado del juego para el cual hay que calcular la acción a realizar, agente al qué le toca jugar, profundidad máxima
- Retorna: la acción a realizar y vector de valores del estado para todos los agentes

`getNextAgentIndex(agentIndex, gameState):`

- Parámetros: estado del juego y número de agente
- Retorna: siguiente agente

[3] Nota: agentIndex==0 es Pac-Man.

`random_unroll(gameState, agentIndex):`

- Parámetros: estado del juego y número de agente
- Retorna: valor del estado luego de realizar un unroll aleatorio

`montecarlo_eval(gameState, agentIndex):`

- Parámetros: estado del juego y número de agente
- Retorna: valor del estado luego de realizar una evaluación de tipo Monte Carlo

`montecarlo_tree_search_eval(gameState, agentIndex):`

- Parámetros: estado del juego y número de agente
- Retorna: valor del estado luego de realizar una evaluación de tipo Monte Carlo Tree Search

`selection_stage(node, gameState):`

- Parámetros: nodo del monte carlo tree y estado del juego
- Retorna: nodo del monte carlo tree y estado del juego luego de haber realizado la etapa de selección del algoritmo

`expansion_stage(node, gameState):`

- Parámetros: nodo del monte carlo tree y estado del juego
- Retorna: nodo del monte carlo tree y estado del juego luego de haber realizado la etapa de expansión del algoritmo

`back_prop_stage(node, gameState):`

- Parámetros: nodo del monte carlo tree y estado del juego
- Retorna: nada. Se encarga de flotar los valores del gamestate a partir del nodo recibido hacia la raiz.

Se explicarán algunas clases y funciones dentro de **game_logic** qué son de particular interes.

Dentro de la clase **GameStateExtended** se destacan las siguientes funciones qué pueden ser de utilidad:

`getLegalActions(agentIndex)`

- Parámetros: índice del agente para el cual se quiere saber las acciones posibles
- Retorna: lista de acciones que el agente puede realizar a partir de ese estado

`getNumAgents()`

- Parámetros: void
- Retorna: la cantidad de agentes en el juego

`generateSuccessor(agentIndex, action)`

- Parámetros: índice del agente y acción que realiza
- Retorna: un nuevo estado, resultado de aplicar la acción parámetro

`isEnd()`

- Parámetros: void
- Retorna: True si el juego terminó (de acuerdo a lo definido en la letra).

`get_rewards()`

- Parámetros: void
- Retorna: El valor del estado para cada agente (si el estado es final, sino un vector de ceros).

`deepCopy()`

- Parámetros: void
- Retorna: Un estado copia del estado original.

Dentro del archivo **game_util.py** se encuentra la función:

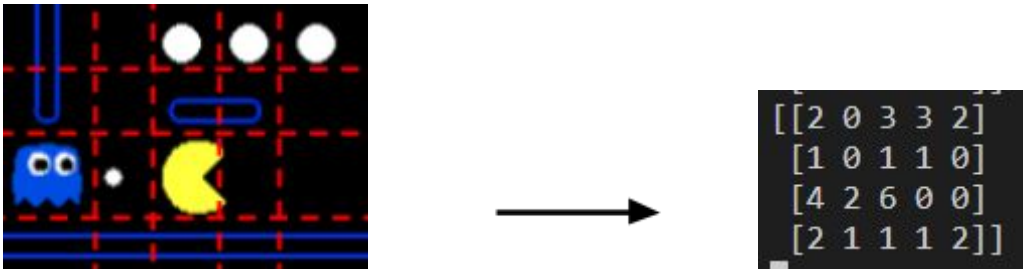
`process_state(gameState,view_distance, agentIndex)`

- Parámetros: estado del juego, distancia máxima de visión (tupla), indice del agente qué está jugando

- Retorna: Una matriz de numpy con las siguientes características:
    - Shape: (view_distance[0]*2+1, view_distance[1]*2+1) (salvo que área fuera de juego sea visible)
        - Ej: view_distance: (2,2), shape: (5,5)
    - Contienen los elementos que se encuentran rodeando al agente, con éste en el centro (en la posición (view_distance[0], view_distance[1])
    - Cada elemento está simbolizado por un número:
        - 0: empty
        - 1: wall
        - 2: food
        - 3: capsule
        - 4: ghost
        - 5: scared ghost
        - 6: Pac-Man
        - 7: playing ghost
        - 8: playing scared ghost
    - Si el agente se encuentra el un borde del mapa, y área fuera de juego queda dentro del view_distance, esta área es omitida, generando una observación con una shape modificada.

- En las extremidades de la observación (arriba-izquierda, arriba-derecha, abajo-izquierda, abajo-derecha) se marca con un 2 si hay food en el cuadrante correspondiente y 0 si no. **Esto se puede modificar para mejorar la estimación de los estados en caso de que el agente sea un fantasma.**

Ejemplo:



*el borde inferior es el límite del mapa