Universidad ORT Uruguay
Facultad de Ingeniería

Misión: vencer a Pac-Man

Obligatorio para
Sistemas Multiagente

Mauricio Repetto - 141045
Federico De León - 252047

Diciembre 2022

TABLA DE CONTENIDOS

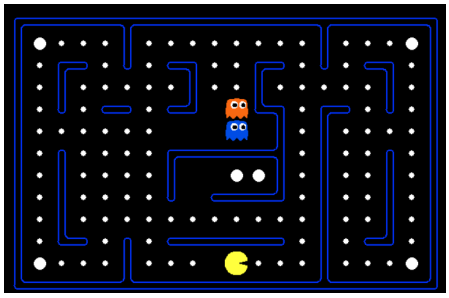
Introducción

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

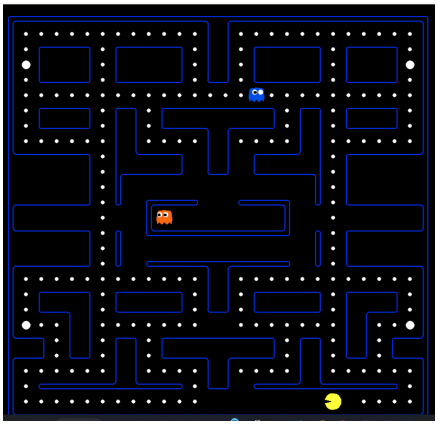
Para lograrlo, se decidió utilizar una versión del juego Pac-Man, especialmente diseñada para la aplicación de algoritmos de inteligencia artificial que fue creada por la universidad de Berkeley. Con el objetivo de habilitar el entrenamiento de fantasmas inteligentes capaces de detener al Pac-Man la versión ha tenido algunos cambios.

Tal como se plantea en la consigna y en el código obtenido, en el juego hay múltiples mapas (layouts), con diferentes niveles de dificultad y dentro de los cuales, se destacan:

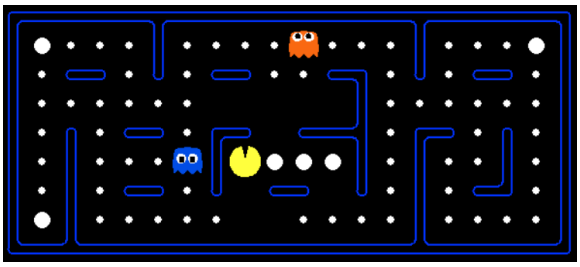
Tricky Classic:



Original Classic:



Contest Classic:



En cada uno de 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 y cuyo objetivo es “comer” a Pac-Man (o correr de él si consumió una cápsula y su tiempo no expiró).

El juego funciona por turnos, en donde el Pac-Man y los ghost determinados realizan una acción. Pac-Man es el agente 0 y la misión de este es: consumir toda la comida más las cápsulas del mapa. Por otro lado, los Ghosts tienen como misión 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 otro lado, toda acción de los fantasmas les cuesta pac-coins.

El juego termina si ocurre algo de lo siguiente:

Condición:	**Ganador:**
Pac-Man es eliminado por un Ghost	Ghost que come a Pac-Man
Pac-Man consume toda la comida y cápsulas del mapa	Pac-Man
Pac-Man agota su crédito.	Nadie

Por otro lado, 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. En caso contrario, se le restará 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.

Tarea

Este trabajo consistió en implementar ***MaxN*** realizando poda por profundidad. Este algoritmo es uno de los más usados en IA para Reinforcement Learning en muchos escenarios de juegos como lo es el Go. La idea detrás de su uso es ayudar a un agente a tomar decisiones más estratégicas en función de los sucesos posibles. Dicho algoritmo es una variante de ***minimax***,

que se usa comúnmente en juegos de dos jugadores como el ajedrez. ***MaxN*** permite al agente considerar múltiples movimientos posibles y elegir el que tiene más probabilidades de conducir a una victoria donde para poder lograr estimar el valor de los estados se realizan rollouts. En dicha implementación también fue necesario desarrollar otras dos técnicas:

- Monte Carlo
- Monte Carlo Tree Search

Tal como se planteaba en la consigna, los métodos de estimación basados en Monte Carlo normalmente requieren construir un episodio completo, es decir, que termine en un estado final. En este caso, esto puede tener un costo prohibitivo en tiempo computacional. Por esta razón, se pedía también implementar los métodos considerando la utilización de una función de evaluación heurística para estimar el valor en un prefijo de episodio, de longitud definida por un parámetro (**max_unroll_depth**), en caso de no llegar a un estado final.

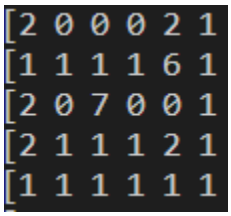
Función de Evaluación

La función de evaluación se realizó recibiendo un estado del juego (información completa del mismo), el cual a través de una función llamada **process_state** lo procesa (ajustando para dar una visión acotada al agente) y luego realiza ciertos cómputos para obtener una *reward* en base a lo que observa.

¿Qué tan acotada es la visión de cada agente en **process_state**? Eso se determina con un parámetro, el cual es una tupla y es llamado **view_distance.** Dicho parámetro puede ser por ejemplo:

- (2, 2)
- (3, 4)
- (5, 5)
- (30, 30)
- etc.

En el caso de ser (5,6) una posible visión del ghost, seria:



Donde cada número simboliza un elemento:

- 0: empty
- 1: wall
- 2: food
- 3: capsule
- 4: ghost
- 5: scared ghost
- 6: Pac-Man
- 7: playing ghost
- 8: playing scared ghost

Tomando los elementos anteriores presentes en el campo de visión acotado para cada agente, se construyó la una función de evaluación que retorna la recompensa acumulada para cada agente (en cada turno) y en base a una serie de situaciones:

Playing Ghost visualiza a Pac-Man

Agregada con el objetivo de incentivar que Ghost esté lo más cerca posible de Pac-Man hasta comérselo.

Reward:

$$10/distance\_to\_pacman$$

Donde:

- *Distance_to_pacman* es la distancia de Manhattan desde el Pac-Man a Playing Ghost

Playing scared Ghost visualiza a Pac-Man

Agregada con el objetivo de incentivar que cuando el Ghost esté asustado se aleje lo máximo posible de Pac-Man para lograr evitar ser comido.

Reward:

$$-10/distance\_to\_pacman$$

Donde:

- *Distance_to_pacman* es la distancia de Manhattan desde el Pac-Man a Playing Ghost.

Playing Ghost no visualiza a Pac-Man

Creada para castigar al Ghost cuando no visualiza a Pac-Man y así tener incentivos a seguir moviéndose hasta encontrarlo.

Reward:

-10

Playing scared Ghost no visualiza a Pac-Man

Agregada para que cuando Ghost se encuentra asustado, premiarlo por no ver a Pac-Man para lograr que esté lo más alejado del mismo y así evitar ser comido.

Reward:

100

Playing Ghost cantidad de comida visualizada

Creada con el objetivo de lograr que Ghost esté cerca de la mayor cantidad posible de comida dado que Pac-Man las necesita para poder ganar el juego.

Reward:

*food_count / processed_obs.shape[0] * processed_obs.shape[1]*

Donde:

- *food_count* representa la cantidad de comida (2: food) visualizada por el Ghost.
- *processed_obs.shape[0] * processed_obs.shape[1]* representa el ratio de elementos observados sobre los que podría visualizar (en el caso de estar en una esquina, lo visualizado es menor al parámetro *view_distance*)

Monte Carlo

Tal como vimos en el curso, la simulación de Monte Carlo es una técnica matemática utilizada para modelar el comportamiento de un sistema generando números (o situaciones) aleatorios y usándolos para evaluar el sistema.

En nuestro caso, para poder implementarlo, definimos una función llamada **random_unroll** en la cual generamos movimientos aleatorios para simular diferentes escenarios y los usamos

para evaluar el valor de cada una de las jugadas. El movimiento con el valor estimado más alto, lo seleccionamos como el mejor movimiento a realizar.

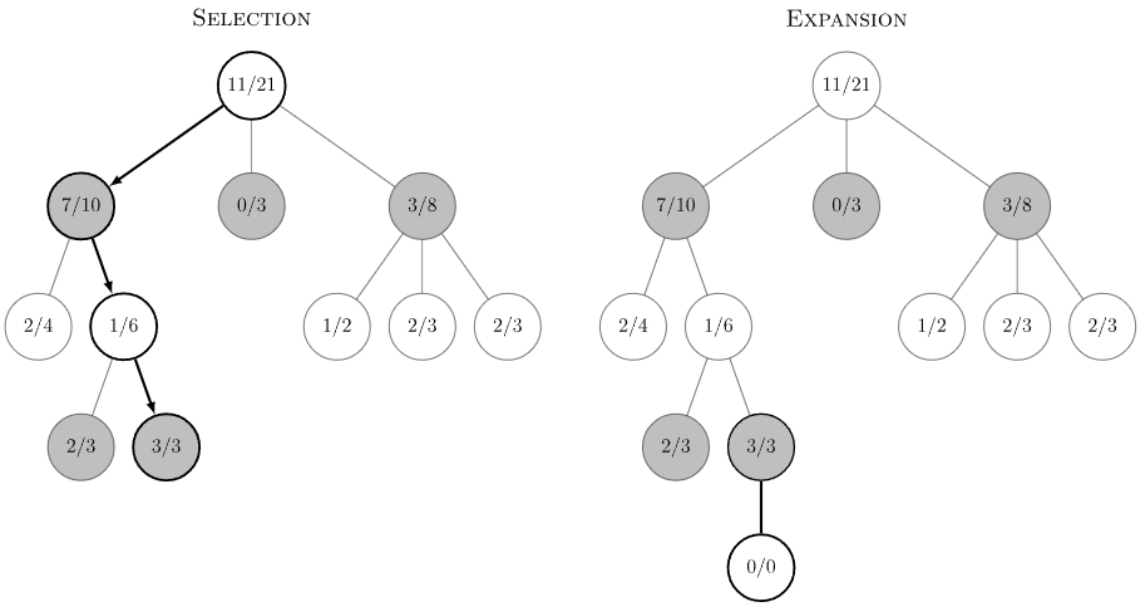
Monte Carlo Tree Search

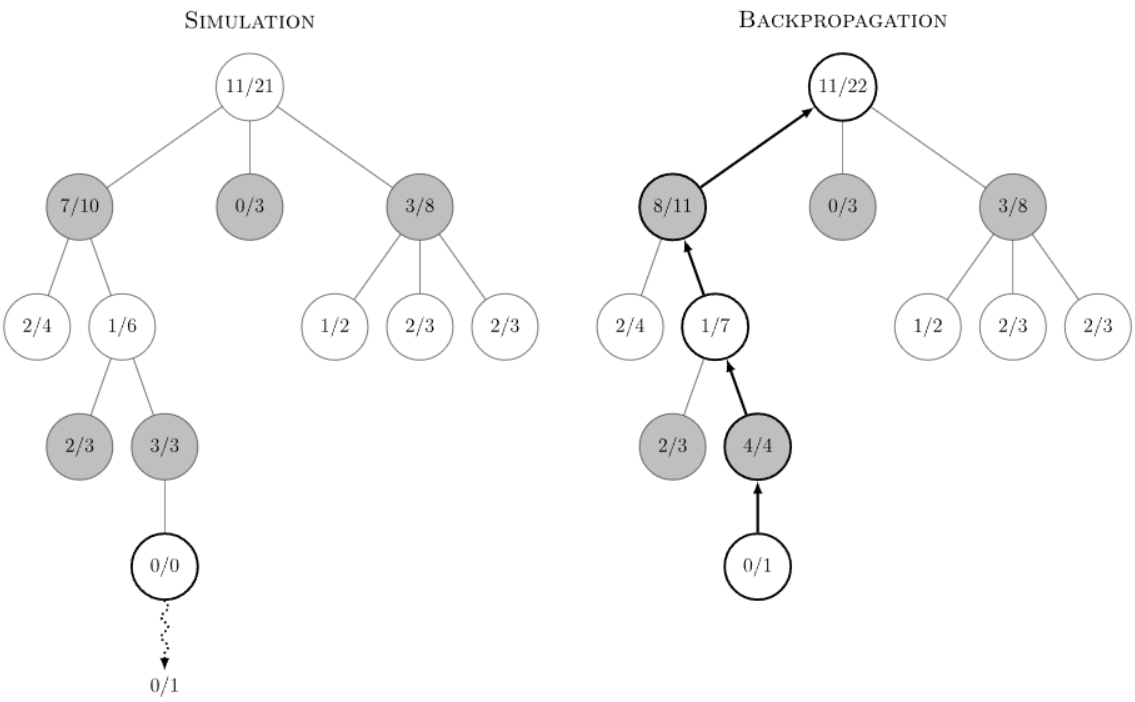
MCTS es un algoritmo de búsqueda heurística que utiliza muestreo aleatorio para estimar el valor de una función. La función que se estima es el valor de una posición de juego, y el algoritmo utiliza un muestreo aleatorio para evaluar diferentes movimientos y seleccionar el mejor.

La idea básica detrás de MCTS es construir un árbol de posibles movimientos, en el que cada nodo del árbol representa una posición potencial del juego. El algoritmo comienza en el nodo raíz y selecciona los nodos secundarios hasta alcanzar un nodo hoja (etapa *Selection*), luego y partiendo del nodo al que llegamos, generamos uno o más movimientos y elegimos uno de ellos (etapa *Expansion*). Del nodo elegido, aplicamos Monte Carlo hasta llegar a un estado final (etapa *Simulation*) y una vez terminado, usamos esa evaluación para actualizar el valor del nodo y propagar este valor hacia arriba en el árbol (etapa *Backpropagation*). Este proceso se repite hasta alcanzar el número deseado de iteraciones, momento en el cual el algoritmo selecciona el movimiento que conduce a la posición del juego con el valor estimado más alto.

Tal como hemos visto, se ha demostrado que MCTS es eficaz en una variedad de escenarios de juego (ver AlphaGo) y, a menudo, se usa junto con otros algoritmos de búsqueda para mejorar el rendimiento.

En la siguiente imagen podemos ver las cuatro etapas:





DQN

En este problema podemos usar Deep Q Learning para entrenar a un Ghost que navegue por el tablero de manera efectiva. Esto le va a permitir al agente aprender jugando, en base a las recompensas que va a ir recibiendo en cada turno. La red usada en el algoritmo de DQN nos va a permitir aproximarnos a la función acción-valor óptima de cada agente en cada layout, de modo que el Ghost pueda elegir la mejor acción a realizar en cada estado. Con el tiempo, la política del agente debería mejorar a medida que aprende a maximizar sus recompensas.

Implementación

La arquitectura que utilizamos tiene dos componentes principales: una serie de capas totalmente conectadas (nn.Linear) que procesan los datos de entrada y un clasificador que asigna los datos de entrada procesados a las predicciones de salida del modelo. El número de predicciones de salida se especifica mediante el parámetro n_actions, que se pasa al método __init__ cuando se inicializa el modelo:

```
class DQN_Model(nn.Module):
    def __init__(self, input_size, n_actions: int):
```

```
        super().__init__()
        self.nn = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True)
        )
        self.classifier = nn.Sequential(
            nn.Linear(512, n_actions)
        )

    def forward(self, env_input):
        x = torch.flatten(env_input, start_dim=1)
        x = self.nn(x)
        x = self.classifier(x)
        return x
```

La implementación que realizamos, fue en base al trabajo final desarrollado para Agentes Inteligentes (ver repositorio). De todas formas, tuvimos que realizarle algunas adaptaciones para lograr obtener el funcionamiento deseado:

Convertir acciones a índice

Esto lo realizamos en el archivo **abstract_agent.py** a modo de poder usar las acciones como input en nuestro modelo y al output, interpretarlo como una acción. Para lograrlo, definimos dos diccionarios, action_to_index e index_to_action, que se utilizan para mapear acciones (por ejemplo, moverse hacia el norte, sur, este, oeste o detenerse) a índices numéricos y viceversa.

```
        self.action_to_index = {
            Directions.NORTH: 0,
            Directions.SOUTH: 1,
            Directions.EAST: 2,
            Directions.WEST: 3,
            Directions.STOP: 4
        }

        self.index_to_action = {
            0: Directions.NORTH,
            1: Directions.SOUTH,
            2: Directions.EAST,
            3: Directions.WEST,
            4: Directions.STOP
        }
```

Filtrado de Legal Actions

La red, en ciertos momentos, nos puede dar una acción que no está dentro de las legales (por ejemplo, puede existir en ese lugar una pared, etc). Para solucionar este problema, limitamos la selección de la mejor acción a la mejor que se encuentra dentro de las legales. Esto fue implementado en la función **_predict_action**.

Uso de Padding

Para realizar el entrenamiento, con la idea de poder usar siempre una misma red que pueda entrenar en diferentes layouts con diferentes visiones, hicimos uso de Padding para poder dejar fijo el input de la red. En nuestro caso y a modo de poder diferenciar a los elementos visualizados, introducimos el valor 9 para indicar el “espacio no explorado”.

Entrenamiento

El entrenamiento fue realizado en una notebook (**ghost_dqn_training.ipynb)** en Colab en base a los siguientes hiperparametros:

TOTAL_STEPS = 50000000
EPISODES = 100000
STEPS = 100000

EPSILON_INI = 1
EPSILON_MIN = 0.02
*EPSILON_TIME = (EPSILON_INI - EPSILON_MIN) * TOTAL_STEPS*

EPISODE_BLOCK = 10
USE_PRETRAINED = True

BATCH_SIZE = 32
BUFFER_SIZE = 10000

GAMMA = 0.99
LEARNING_RATE = 0.0001

SAVE_BETWEEN_STEPS = 100000

MATRIX_SIZE = 30
ACTION_SPACE_N = 5
AGENT_INDEX = 1
ENV_NAME = 'GhostDQN'

Resultados

En las siguientes tablas mostramos, para algunos layouts, las rewards promedio (calculadas realizando 50 iteraciones) de uno de nuestros tres Ghost (DQN Ghost, MC Ghost y MCTS Ghost) versus las obtenidas por el random en los tres test realizados. Nuestro objetivo es ganarle a Random en al menos uno de los test:

Tabla 1: Layout Original Classic

Las rewards obtenidas por Random Ghost son superadas solo por nuestro MC Ghost.

Test	Avg Time	Ghost 0	Random Ghost
MCTS Ghost vs Random Ghost	33.97	-34,468	-21,314
MC Ghost vs Random Ghost	3.97	-3,470	-4,017
DQN Ghost vs Random Ghost	0.6	-47	593

El layout de la Tabla 1, por su tamaño y particularidades, es uno de los más complejos tal como se puede observar en la captura de nuestra introducción.

Tabla 2: Layout Minimax Classic

En este caso, nuestros tres Ghost logran superar al random siendo DQN el de mejor desempeño.

Test	Avg Time	Ghost 0	Random Ghost
MCTS Ghost vs Random Ghost	0.1	400	226
MC Ghost vs Random Ghost	0.02	501	401
DQN Ghost vs Random Ghost	0.01	816	137

El ambiente de la tabla anterior y tal como se puede observar en la columna Avg Time que, muestra el tiempo promedio de cada iteración en cada test, es relativamente sencillo pero nos permite ver el potencial de DQN en el juego de realizarse un mayor training. En nuestro caso y debido al uso de Colab, nos vimos limitados en el uso de GPU y por ende, no pudimos llegar al desempeño deseado en los layouts más complejos.

Los testeos los realizamos en los 9 layouts donde se permitían al menos dos Ghosts. En el Anexo I se pueden ver todos los resultados de las pruebas “Ghost implementado vs Random Ghost” y en el único que no somos vencedores, es en el layout *trappedClassic*.

En tres layouts (custom1, originalClassic y trickyClassic) se permitían al menos cuatro Ghosts al mismo tiempo. Realizamos el test con 50 iteraciones y en verde, podemos observar los agentes nuestros que tuvieron un desempeño mejor al random:

Layout	MCT Ghost	MC Ghost	DQN Ghost	Random Ghost
custom1	-9178.76	-9347.84	-25200.02	-23235.4
originalClassic	-3961.34	-3846.04	-9098.94	-7228.48
trickyClassic	-2280.58	-2133.32	-5858.62	-4873.58

En el Anexo II, podemos encontrar los resultados de todos los testeos realizados.

Conclusiones

Tal como fuimos mencionando, el objetivo de nuestro proyecto fue crear un algoritmo que pudiera mejorar el comportamiento de un Ghost aleatorio en el juego Pac-Man y finalmente, que pueda vencer a Pac-Man. Después de una amplia experimentación y pruebas, creemos que hemos sido capaces de desarrollar con éxito varios agentes que logran superar al random Ghost mediante el uso de MaxN con Monte Carlo, Monte Carlo Tree Search y Deep Q Learning.

Uno de los principales retos a los que nos enfrentamos durante el desarrollo de este algoritmo fue la complejidad del propio juego en general y más aún, en cierto tipo de layouts (por ejemplo: Original Classic). Pac-Man es un juego muy dinámico e impredecible, lo que dificulta el desarrollo de un agente fiable que pueda tomar siempre las decisiones correctas. Además, la enorme cantidad y combinaciones de movimientos posibles en el juego dificulta la evaluación del rendimiento de nuestro algoritmo en un tiempo razonable.

A pesar de lo anterior, consideramos que nuestro algoritmo fue capaz de explorar eficazmente el vasto espacio de posibilidades y tomar decisiones informadas basadas en la experiencia pasada. Además, creemos que todo el proyecto realizado, refuerza el potencial de las cosas vistas durante el curso para resolver problemas complejos.

Por otro lado, queremos destacar que de cara al futuro, hay muchos desarrollos y mejoras potenciales que podrían hacerse sobre nuestro trabajo. Por ejemplo, podríamos incorporar otras técnicas de IA para mejorar aún más el rendimiento de nuestro algoritmo, así como:

- ➔ Centrar nuestro entrenamiento en diseños de juego más complejos para preparar mejor nuestro algoritmo para situaciones difíciles.
- ➔ Mejorar la función de evaluación.
- ➔ Mejorar la red neuronal en nuestro modelo de Deep Q Learning para hacerlo más eficaz.
- ➔ Investigar el uso de técnicas de Pooling para mejorar aún más el rendimiento de nuestro algoritmo.

Repositorio Github

Creamos un repositorio que luego de la entrega, quedará público para compartir y aportar material a la comunidad. El mismo se puede encontrar en el siguiente enlace: https://github.com/fededemo/ObligatorioMultiagentes

Anexo

Anexo I

TestName	Time Avg	Ghost 0	Random Ghost
custom1 rnd pcmn MCTS Ghost vs Random Ghost	33.97	-34468.18	-21314.38
custom1 rnd pcmn MC Ghost vs Random Ghost	3.97	-3470.88	-4017.48
custom1 rnd pcmn DQN Ghost vs Random Ghost	0.6	-47.06	593.04
capsuleClassic rnd pcmn MCTS Ghost vs Random Ghost	1.3	-1178.22	-818.02
capsuleClassic rnd pcmn MC Ghost vs Random Ghost	0.21	285.02	157.38
capsuleClassic rnd pcmn DQN Ghost vs Random Ghost	0.12	245.68	645.88

contestClassic rnd pcmn MCTS Ghost vs Random Ghost	4.01	-3967.4	-2983.42
contestClassic rnd pcmn MC Ghost vs Random Ghost	0.46	-25.44	-52.62
contestClassic rnd pcmn DQN Ghost vs Random Ghost	0.18	265.78	585.98
mediumClassic rnd pcmn MCTS Ghost vs Random Ghost	4.75	-4446.86	-3251.1
mediumClassic rnd pcmn MC Ghost vs Random Ghost	0.59	-156.56	-114.46
mediumClassic rnd pcmn DQN Ghost vs Random Ghost	0.24	66.88	746.98
minimaxClassic rnd pcmn MCTS Ghost vs Random Ghost	0.1	399.56	226.24
minimaxClassic rnd pcmn MC Ghost vs Random Ghost	0.02	500.74	400.56
minimaxClassic rnd pcmn DQN Ghost vs Random Ghost	0.01	816.44	136.96
originalClassic rnd pcmn MCTS Ghost vs Random Ghost	22.07	-9196.18	-6944.02
originalClassic rnd pcmn MC Ghost vs Random Ghost	4.29	-1317.82	-1714.72
originalClassic rnd pcmn DQN Ghost vs Random Ghost	2.14	-347.34	492.7
smallClassic rnd pcmn MCTS Ghost vs Random Ghost	1.98	-1981.98	-1444.16
smallClassic rnd pcmn MC Ghost vs Random Ghost	0.44	-54.84	-131.86
smallClassic rnd pcmn DQN Ghost vs Random Ghost	0.09	161.34	761.48
trappedClassic rnd pcmn MCTS Ghost vs Random Ghost	0.01	83.44	889.96

trappedClassic rnd pcmn MC Ghost vs Random Ghost	0.01	173.92	814.32
trappedClassic rnd pcmn DQN Ghost vs Random Ghost	0.01	-1.94	998.06
trickyClassic rnd pcmn MCTS Ghost vs Random Ghost	10.06	-9272.9	-6388.46
trickyClassic rnd pcmn MC Ghost vs Random Ghost	1.53	-894.48	-1092.58
trickyClassic rnd pcmn DQN Ghost vs Random Ghost	0.48	-3.44	716.6

Anexo II

Layout	MCT Ghost	MC Ghost	DQN Ghost	Random Ghost
custom1	-9178.76	-9347.84	-25200.02	-23235.4
custom1_rnd_pcmn_mcts_vs_random	-34468.18	-21314.38	NaN	NaN
custom1_rnd_pcmn_mc_vs_random	-3470.88	-4017.48	NaN	NaN
custom1_rnd_pcmn_dqn_vs_random	-47.06	593.04	NaN	NaN
custom1_rnd_pcmn_mcts_vs_mc	-36813.36	-24218.76	NaN	NaN
custom1_rnd_pcmn_mcts_vs_dqn	-39482.06	-24581.08	NaN	NaN
custom1_rnd_pcmn_mc_vs_dqn	-5165.9	-6287.46	NaN	NaN
capsuleClassic_rnd_pcmn_all	-772.36	-921.88	-2300.86	NaN
capsuleClassic_rnd_pcmn_mcts_vs_random	-1178.22	-818.02	NaN	NaN
capsuleClassic_rnd_pcmn_mc_vs_random	285.02	157.38	NaN	NaN
capsuleClassic_rnd_pcmn_dqn_vs_random	245.68	645.88	NaN	NaN
capsuleClassic_rnd_pcmn_mcts_vs_mc	-1548.86	-1163.18	NaN	NaN
capsuleClassic_rnd_pcmn_mcts_vs_dqn	-1337.72	-1500.84	NaN	NaN

capsuleClassic_rnd_pcmn_mc_vs_dqn	506.2	-341.2	NaN	NaN
contestClassic_rnd_pcmn_all	-1115.62	-1340.56	-2635.22	NaN
contestClassic_rnd_pcmn_mcts_vs_random	-3967.4	-2983.42	NaN	NaN
contestClassic_rnd_pcmn_mc_vs_random	-25.44	-52.62	NaN	NaN
contestClassic_rnd_pcmn_dqn_vs_random	265.78	585.98	NaN	NaN
contestClassic_rnd_pcmn_mcts_vs_mc	-3267.86	-2397.22	NaN	NaN
contestClassic_rnd_pcmn_mcts_vs_dqn	-3415.9	-2330.86	NaN	NaN
contestClassic_rnd_pcmn_mc_vs_dqn	49.82	272.86	NaN	NaN
mediumClassic_rnd_pcmn_all	-5244.84	-4089.96	NaN	NaN
mediumClassic_rnd_pcmn_mcts_vs_random	-4446.86	-3251.1	NaN	NaN
mediumClassic_rnd_pcmn_mc_vs_random	-156.56	-114.46	NaN	NaN
mediumClassic_rnd_pcmn_dqn_vs_random	66.88	746.98	NaN	NaN
mediumClassic_rnd_pcmn_mcts_vs_mc	-5156.96	-3855.78	NaN	NaN
mediumClassic_rnd_pcmn_mcts_vs_dqn	-4955.8	-4224.18	NaN	NaN
mediumClassic_rnd_pcmn_mc_vs_dqn	-229.76	-835.6	NaN	NaN
minimaxClassic_rnd_pcmn_all	418.58	240.78	-106.92	NaN
minimaxClassic_rnd_pcmn_mcts_vs_random	399.56	226.24	NaN	NaN
minimaxClassic_rnd_pcmn_mc_vs_random	500.74	400.56	NaN	NaN
minimaxClassic_rnd_pcmn_dqn_vs_random	816.44	136.96	NaN	NaN
minimaxClassic_rnd_pcmn_mcts_vs_mc	446.16	148.64	NaN	NaN
minimaxClassic_rnd_pcmn_mcts_vs_dqn	512.86	176.08	NaN	NaN
minimaxClassic_rnd_pcmn_mc_vs_dqn	601.58	301.86	NaN	NaN
originalClassic	-3961.34	-3846.04	-9098.94	-7228.48
originalClassic_rnd_pcmn_mcts_vs_random	-9196.18	-6944.02	NaN	NaN
originalClassic_rnd_pcmn_mc_vs_random	-1317.82	-1714.72	NaN	NaN

originalClassic_rnd_pcmn_dqn_vs_random	-347.34	492.7	NaN	NaN
originalClassic_rnd_pcmn_mcts_vs_mc	-14317.12	-11158.22	NaN	NaN
originalClassic_rnd_pcmn_mcts_vs_dqn	-18869.74	-15419.2	NaN	NaN
originalClassic_rnd_pcmn_mc_vs_dqn	-1546.98	-2629.14	NaN	NaN
smallClassic_rnd_pcmn_all	-3382.14	-2229.16	NaN	NaN
smallClassic_rnd_pcmn_mcts_vs_random	-1981.98	-1444.16	NaN	NaN
smallClassic_rnd_pcmn_mc_vs_random	-54.84	-131.86	NaN	NaN
smallClassic_rnd_pcmn_dqn_vs_random	161.34	761.48	NaN	NaN
smallClassic_rnd_pcmn_mcts_vs_mc	-3016.72	-2185.8	NaN	NaN
smallClassic_rnd_pcmn_mcts_vs_dqn	-2609.76	-2264.08	NaN	NaN
smallClassic_rnd_pcmn_mc_vs_dqn	336.58	-79.86	NaN	NaN
trappedClassic_rnd_pcmn_all	116.04	843.48	NaN	NaN
trappedClassic_rnd_pcmn_mcts_vs_random	83.44	889.96	NaN	NaN
trappedClassic_rnd_pcmn_mc_vs_random	173.92	814.32	NaN	NaN
trappedClassic_rnd_pcmn_dqn_vs_random	-1.94	998.06	NaN	NaN
trappedClassic_rnd_pcmn_mcts_vs_mc	48.86	897.92	NaN	NaN
trappedClassic_rnd_pcmn_mcts_vs_dqn	78.28	887.06	NaN	NaN
trappedClassic_rnd_pcmn_mc_vs_dqn	53.28	933.62	NaN	NaN
trickyClassic	-2280.58	-2133.32	-5858.62	-4873.58
trickyClassic_rnd_pcmn_mcts_vs_random	-9272.9	-6388.46	NaN	NaN
trickyClassic_rnd_pcmn_mc_vs_random	-894.48	-1092.58	NaN	NaN
trickyClassic_rnd_pcmn_dqn_vs_random	-3.44	716.6	NaN	NaN
trickyClassic_rnd_pcmn_mcts_vs_mc	-9771.1	-7162.04	NaN	NaN
trickyClassic_rnd_pcmn_mcts_vs_dqn	-9184.38	-6916.24	NaN	NaN
trickyClassic_rnd_pcmn_mc_vs_dqn	-673.04	-1141.12	NaN	NaN

Bibliografia

- https://github.com/davislf2/AI_Pacman
- https://github.com/dwaynee174/Ghost-Hunter-Pacman
- http://ai.berkeley.edu/contest.html
- https://github.com/thiagov/pacman-ai
- https://inst.eecs.berkeley.edu/~cs188/fa18/assets/slides/lec6/FA18_cs188_lecture6_adversarial_search_4pp.pdf
- https://courses.cs.washington.edu/courses/cse573/16wi/pacman/ps5/pomdp.html)
- https://abhinavcreed13.github.io/projects/ai-team-pacamon/#approach-four-monte-carlo-tree-search
- https://www.researchgate.net/publication/260583298_Monte_Carlo_Tree_Search_for_Collaboration_Control_of_Ghosts_in_Ms_Pac-Man?enrichId=rgreq-87cb4fda86c0431e1fb4904474eab75d-XXX&enrichSource=Y292ZXJQYWdlOzI2MDU4MzI5ODtBUzoyNjUwNDA1OTA3OTg4NTFAMTQ0MDIwMjAwOTg4MA%3D%3D&el=1_x_3&_esc=publicationCoverPdf
- https://arxiv.org/abs/2103.04931
- https://upload.wikimedia.org/wikipedia/commons/2/21/MCTS-steps.svg