Facultad de Ingeniería

Ingeniería Artificial

Obligatorio

Proyecto de Innovación en Movilidad, Automatización y Estrategia: Aplicaciones de IA en el mundo real

Lucas Álvarez - 268323
Matías Nahuel Saad - 251398

Tutores:
Sergio Yovine
Juan da Silva

2024

Índice

Uso de Weights & Biases (Wandb)

Utilizamos la herramienta Weights & Biases (Wandb), una plataforma para la gestión y el seguimiento de experimentos de machine learning. Nos permitió monitorear el progreso de nuestros modelos “Taxi” y “Péndulo”, registrar métricas clave, y visualizar datos de entrenamiento en tiempo real.
Debido a que tuvimos un problema con la organización creada inicialmente, decidimos crear una organización para cada experimento.
Link a la organizacion (taxi): https://wandb.ai/obligatorioia
Link a la organización (péndulo):
Código de la organización (taxi): 3fef7cf73728339de43b5b59f6d9efd892c46f34
Código de la organización (péndulo):

Los comandos utilizados para instalar wandb en el ambiente fueron los siguientes:

1. pip install wandb
2. wandb login

Donde utilizamos los códigos especificados arriba para loguearnos con la organización.

Con esta herramienta, realizamos reportes para cada experimento los cuales adjuntamos a la entrega. Estos resumen los parámetros utilizados y las métricas obtenidas.

Añadimos el siguiente mail del profesor Juan “juanpedrodasilvabarloco@gmail.com” a las organizaciones para que pueda revisar el trabajo realizado ya que las mismas eran privadas.

Taxi

Introducción

El objetivo de este experimento es entrenar un agente utilizando el algoritmo de Q-Learning para resolver el entorno Taxi-v3. El algoritmo de Q-Learning es una técnica de aprendizaje por refuerzo que permite al agente aprender la política óptima para maximizar la recompensa total a lo largo del tiempo.

Los parámetros que decidimos utilizar fueron:

- α - (alpha) es la tasa de aprendizaje. Controla en qué medida se actualizan los valores en la tabla Q en función de las nuevas experiencias.
- γ - (gamma) es el factor de descuento. Determina cuánto valor se le da a las recompensas futuras en el cálculo de la utilidad esperada de una acción. Utilizamos los valores 0.95 y 0.99 con el objetivo de priorizar recompensas a largo plazo.
- ε - (epsilon) controla el equilibrio entre exploración (tomar acciones aleatorias para descubrir nuevas rutas o estrategias) y explotación (tomar la mejor acción conocida según la tabla Q).
- K - Número de episodios de entrenamiento. Determina cuántos episodios completos se utilizarán para entrenar al agente.
- N - Número de iteraciones. Cuántas veces se realizará el ciclo de entrenamiento y prueba completo.
- L - Número de pruebas por iteración. Cuántas veces se ejecutará el agente entrenado para evaluar su desempeño después de cada ciclo de entrenamiento.

El algoritmo consiste simplemente en iterar N veces el entrenamiento y la prueba del modelo. Entrenar es utilizar Q Learning que tiene su propio loop. Y probar es simplemente ejecutar la policy π L veces y ver el resultado.

Como resultado final, esperamos tener una gráfica de la recompensa en función de las iteraciones donde al comienzo debería variar y luego de cierto punto ir mejorando.

Para los valores de los parámetros, decidimos utilizar un valor de K y alpha no muy grandes. El mas chico es el parámetro L ya que la prioridad debe estar en entrenar al modelo y no es necesario hacer miles de pruebas para observar los resultados. Decidimos variar los valores de alpha y el epsilon luego de cada entrenamiento con el objetivo de mejorar la exploración y explotación del agente. Al reducir el epsilon gradualmente, buscamos disminuir la exploración y aumentar la explotación a medida que el agente aprende la política óptima. Por otro lado, ajustar el alpha nos permite controlar la tasa de aprendizaje, permitiendo que el agente se adapte mejor a la dinámica del entorno a lo largo del tiempo.

Al principio, no variamos el valor de alpha, manteniéndolo constante para simplificar el proceso de aprendizaje. Sin embargo, tras observar que la recompensa promedio no mejoraba significativamente, decidimos comenzar a ajustar el alpha de manera similar al epsilon, para ver si conseguíamos mejoras. Esta estrategia nos permitió observar si un ajuste dinámico del alpha podía contribuir a una mejor convergencia y a una mayor estabilidad en el aprendizaje del agente.

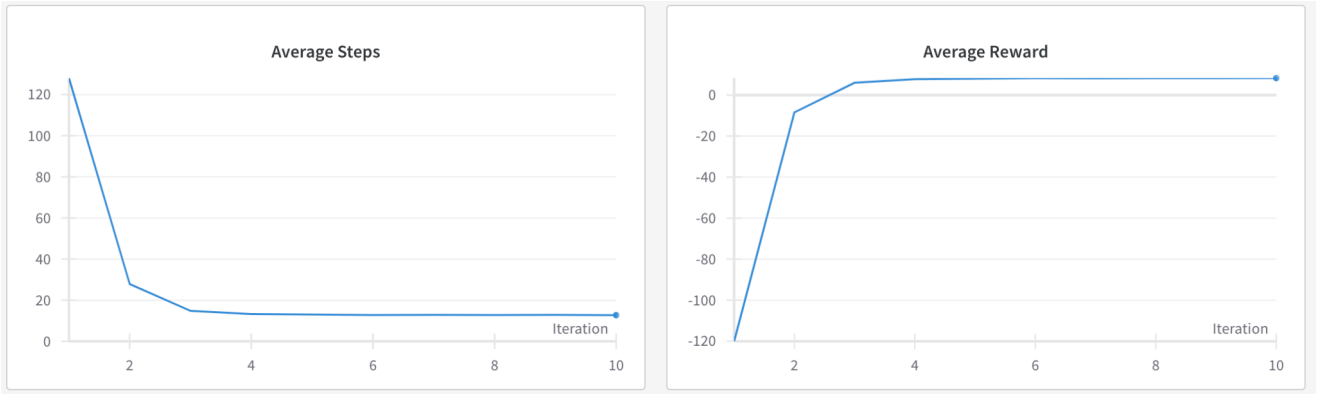
Ejemplos de algunas configuraciones utilizadas

Decidimos comenzar con la siguiente configuración utilizando parámetros bajos para ver cómo funcionaba el experimento:

- **Epsilon**: 0.5
- **Gamma**: 0.95
- **Alpha**: 0.1
- **K**: 1000
- **N**: 10
- **L**: 100

```
Iteration 1/10, Epsilon: 0.495
Average Reward: -120.08, Average Steps: 128.06
Iteration 2/10, Epsilon: 0.49005
Average Reward: -8.49, Average Steps: 27.81
Iteration 3/10, Epsilon: 0.48514949999999996
Average Reward: 5.97, Average Steps: 14.82
Iteration 4/10, Epsilon: 0.480298005
Average Reward: 7.74, Average Steps: 13.26
Iteration 5/10, Epsilon: 0.47549502494999996
Average Reward: 7.94, Average Steps: 13.06
Iteration 6/10, Epsilon: 0.47074007470049994
Average Reward: 8.18, Average Steps: 12.82
Iteration 7/10, Epsilon: 0.46603267395349496
Average Reward: 8.08, Average Steps: 12.92
Iteration 8/10, Epsilon: 0.46137234721396
Average Reward: 8.16, Average Steps: 12.84
Iteration 9/10, Epsilon: 0.45675862374182036
Average Reward: 8.15, Average Steps: 12.85
Iteration 10/10, Epsilon: 0.45219103750440215
Average Reward: 8.26, Average Steps: 12.74
```

Decidimos crear dos gráficas. Por un lado el promedio de pasos en función del número de iteraciones y por otro lado, el promedio de la recompensa en función de las iteraciones.

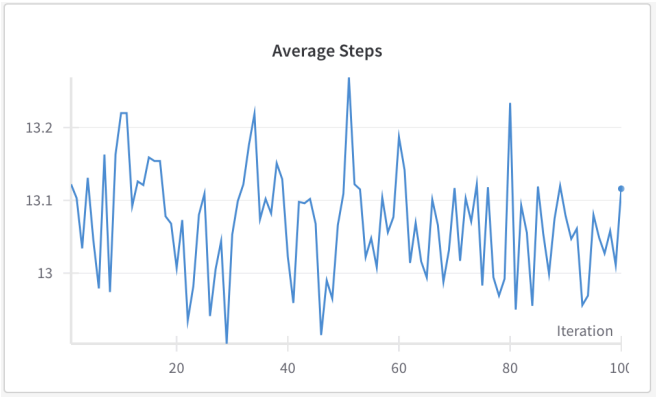


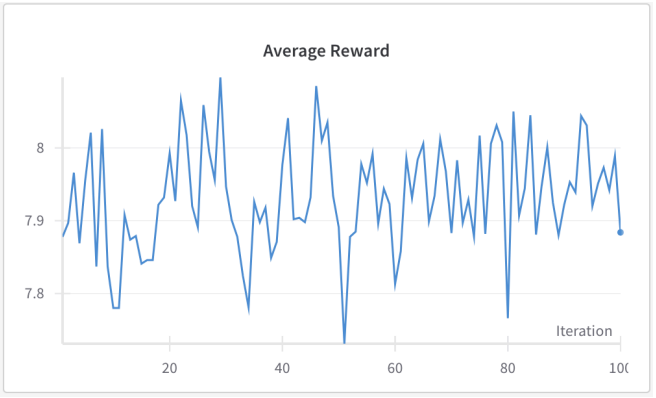
Podemos ver cómo a lo largo de las distintas iteraciones y a medida que se reduce el epsilon aumenta la recompensa promedio y disminuye la cantidad de pasos promedio.

A medida que avanzamos, fuimos modificando los valores de epsilon, alpha, K, N y L.

Para la segunda configuración, utilizamos los siguientes valores de parámetros:

K = 10000 - N = 100 - L = 1000 - alpha = 0.2 - gamma = 0.95 - epsilon = 0.6



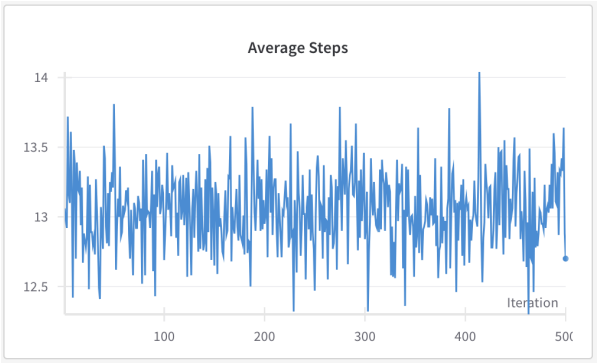


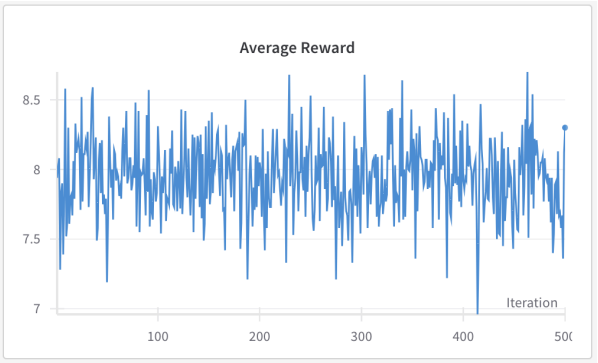
En esta ocasión notamos una gran variación de los promedios de recompensa y pasos en función de las iteraciones.

Debido a que realizamos muchas pruebas con distintas configuraciones, decidimos solamente registrar y analizar en este informe las que consideramos más relevantes. Se puede acceder a la organización de “wandb” (el link se encuentra más arriba) para observar todas las configuraciones utilizadas y los resultados en detalle.

Decidimos aumentar la cantidad de iteraciones y aumentar los episodios de entrenamiento utilizando la siguiente configuración.

K = 20000 - N = 500 - L = 100 - alpha = 0.1 - gamma = 0.95 - epsilon = 0.9





Seguiamos notando que la recompensa promedio no aumentaba por lo que intentamos con aún más iteraciones utilizando la siguiente configuración:

Config parameters: {} 10 keys

alpha_decay: 0.995

alpha_end: 0.01

alpha_start: 0.1

epsilon_decay: 0.995

epsilon_end: 0.1

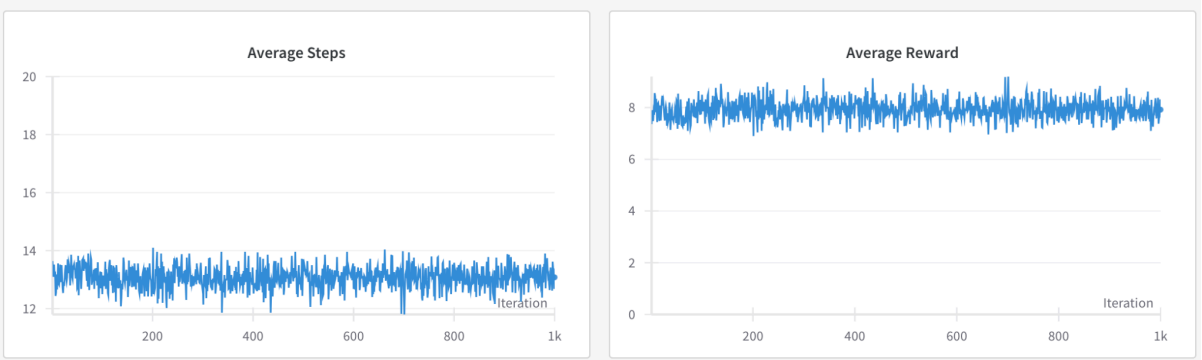
epsilon_start: 0.99

gamma: 0.99

K: 10,000

L: 50

N: 1,000



Pensamos que el problema podía deberse a un sobreajuste por lo que volvimos a bajar la cantidad de iteraciones y de episodios de entrenamiento. Además, decidimos bajar la cantidad de episodios de prueba y subir la cantidad de iteraciones con el objetivo de entrenar mejor el modelo.

Config parameters: {} 10 keys

alpha_decay: 0.995

alpha_end: 0.01

alpha_start: 0.1

epsilon_decay: 0.995

epsilon_end: 0.1

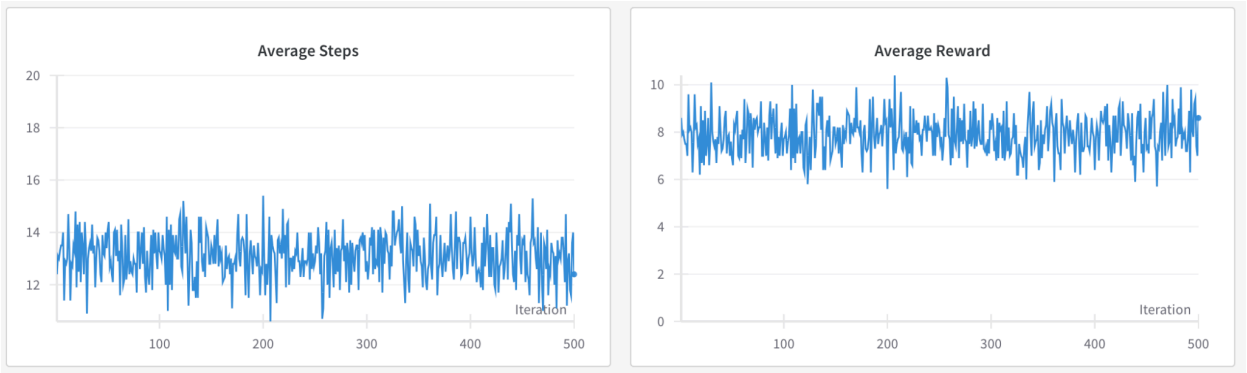
epsilon_start: 0.99

gamma: 0.99

K: 5,000

L: 10

N: 500



En esta ocasión, notamos que luego de la iteración 300, existe un leve aumento de las recompensas en promedio, sin embargo, seguimos buscando algo más notorio.

Probamos también con diferentes valores iniciales de alfa y epsilon. Al bajar el epsilon inicial, se reduce la probabilidad de exploración al comienzo del entrenamiento. Pensamos que esto podría ser beneficioso al aprovechar más las decisiones iniciales informadas que la exploración aleatoria. Además, un valor más alto de alpha inicial permite que las actualizaciones de Q se basen más en las recompensas recientes, acelerando el aprendizaje en las primeras etapas.

Config parameters: {} 10 keys

alpha_decay: 0.995

alpha_end: 0.01

alpha_start: 0.5

epsilon_decay: 0.995

epsilon_end: 0.5

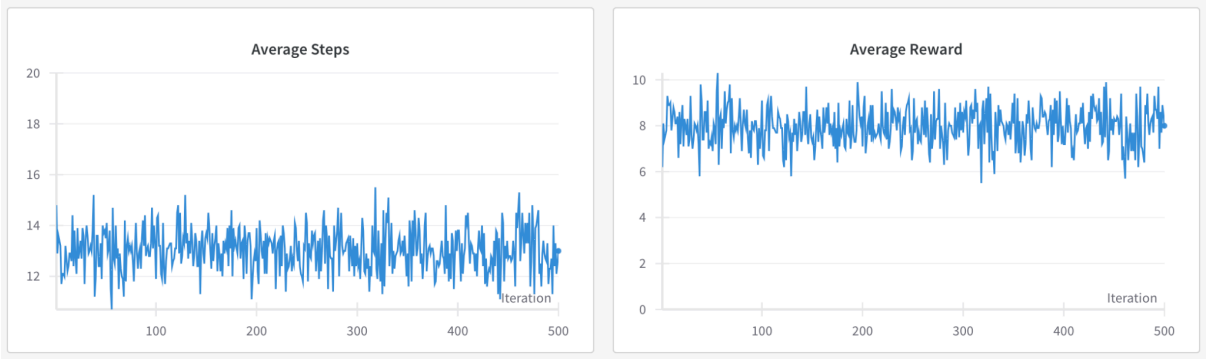
epsilon_start: 0.3

gamma: 0.99

K: 5,000

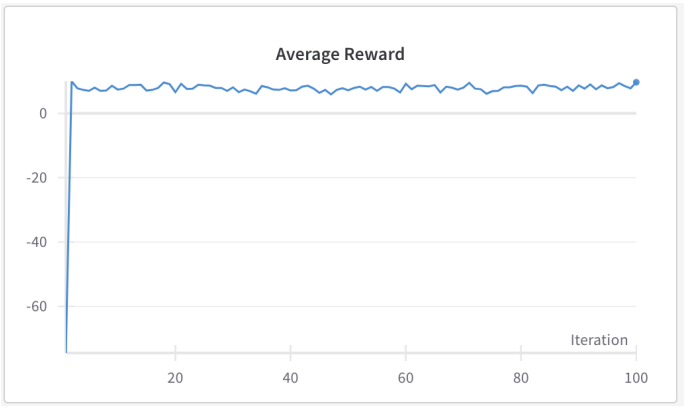
L: 10

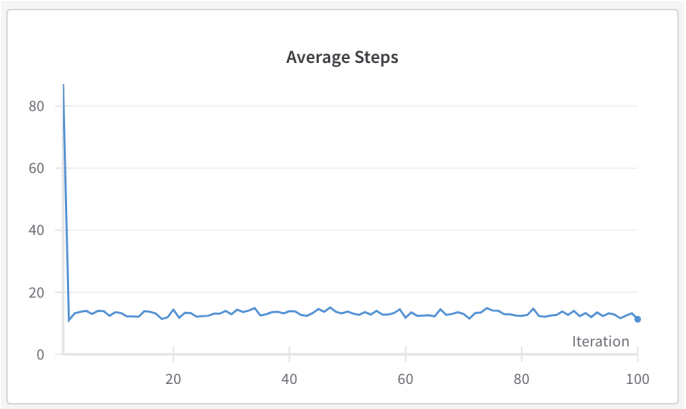
N: 500



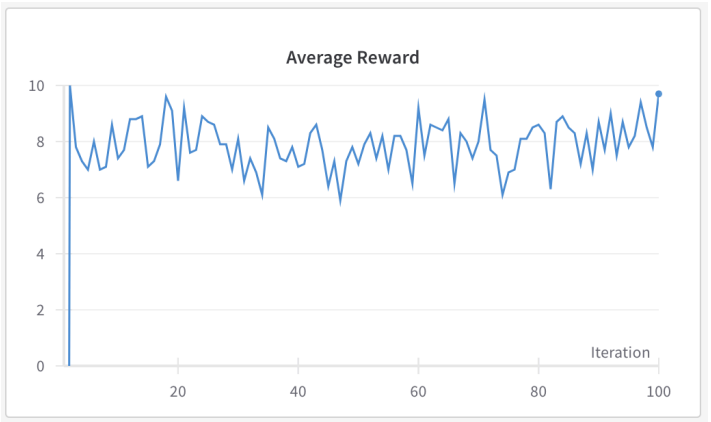
Como podemos observar, continuamos con el mismo problema.

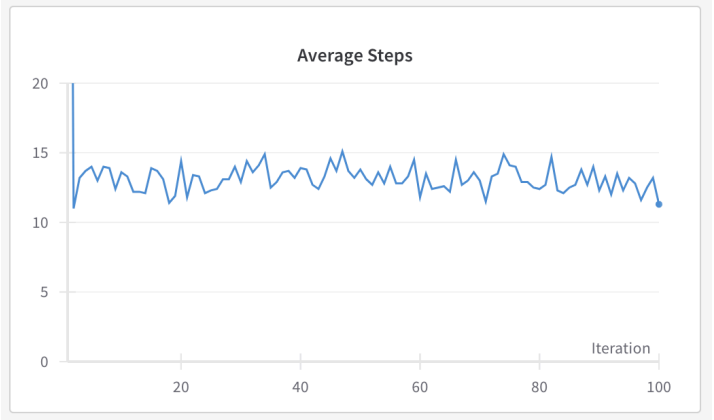
Ya que estábamos probando con muchas iteraciones y estaba demorando mucho tiempo la ejecución sin tener buenos resultados, decidimos bajar la cantidad de iteraciones y de episodios de entrenamiento. Utilizamos la siguiente configuración:





Con un poco más de zoom:





Decidimos quedarnos con esta configuración. A partir de la iteración 85 (aprox), los resultados en ambas gráficas mejoran. Por un lado, en las recompensas, vemos como comienzan a subir, siendo la

última de 9.7. Por otro lado, en los steps, sucede lo mismo pero bajando, siendo el último 11.3. Además de tener estos resultados positivos, fue en un tiempo de ejecución bastante razonable (5 minutos), por lo que decidimos quedarnos con el mismo. En wandb se puede ver con el nombre "*icy-bird-50*".

Conclusión

A lo largo de las iteraciones, monitoreamos la recompensa promedio obtenida por el agente en los episodios de prueba. Idealmente, esperamos ver un aumento en la recompensa promedio a medida que el agente aprende una política más óptima. El número promedio de pasos por episodio también es una métrica importante. Un agente eficiente debería aprender a completar el episodio en el menor número de pasos posible.
Luego de muchas sesiones de experimentación, no logramos que la recompensa aumente luego de cierto punto, en cambio, la misma varía entre los valores de 7.0 y 10.5. A pesar de no obtener los resultados que nos gustaría haber logrado, el experimento siempre nos da recompensas positivas de buen valor. Por otro lado, nos sucede lo mismo con la cantidad de pasos. Los mismos varían entre los valores de 10.5 y 15.3, los cuales consideramos buenos valores.
Sospechamos que la razón por la que la recompensa promedio no mejora es debido a un posible sobreajuste del modelo a ciertas trayectorias específicas del entorno. Este sobreajuste puede ocurrir cuando el agente se vuelve demasiado dependiente de las experiencias pasadas y no generaliza bien a nuevas situaciones. Para resolver este problema, fue que intentamos varias el alpha y epsilon con distintos valores iniciales y tasas de decaimiento buscando equilibrar la exploración y explotación de manera más efectiva y promover un aprendizaje más robusto y generalizable.
Realizamos un total de 43 distintas combinaciones de configuraciones con los diferentes parámetros. Cada una de ellas se pueden ver en la organización de wandb.
Por último, en la carpeta "./Documentación/Taxi" del proyecto tendremos los reportes de Wandb para todas las ejecuciones y el modelo entrenado en formato .plk.

Péndulo

En este ejercicio se abordó el problema de reducir el tiempo de ensamblaje en una fábrica de relojes utilizando un péndulo controlado por un agente de aprendizaje por refuerzo. Se utilizó el algoritmo Q-Learning en el desarrollo del entorno. Una vez completado el código, se probaron varios conjuntos de parámetros para entrenar el agente y evaluar su desempeño en términos de recompensa y número de pasos por episodio.

Interacción con el Simulador

El entorno PendulumEnvExtended se utilizó para simular el péndulo en la fábrica de relojes. Este entorno extiende el clásico Pendulum-v1 y se configuró para reflejar las condiciones operativas específicas de la fábrica. Tras haber encontrado que Pendulum-v1, al utilizar la función step, retornaba un estado de "done" constante en false, se decidió configurar nuestro entorno PendulumEnvExtended para que califique como "done" cuando el péndulo tiene un posición lo suficientemente vertical.

Desarrollo del Código

El desarrollo del código para optimizar el tiempo de ensamblaje en la fábrica de relojes utilizando un péndulo comenzó con la implementación de un agente de aprendizaje por refuerzo utilizando el algoritmo Q-Learning. Se empleó el entorno "Pendulum-v1" de Gymnasium, extendido para ajustarse a las condiciones específicas de la fábrica.

Para la gestión de experimentos y monitoreo del rendimiento del agente, se utilizó Weights & Biases (wandb), una plataforma que facilita la visualización y comparación de los resultados de múltiples ejecuciones. Inicialmente, configuramos un proyecto en wandb donde realizamos diversas pruebas para entender el comportamiento del código y los parámetros.

Primeras Pruebas y Problemas Encontrados

Durante las primeras pruebas, nos enfrentamos a algunos desafíos significativos. Una de las principales dificultades fue la alta duración de las pruebas. En una ocasión, una prueba se prolongó aproximadamente 15 horas, lo que nos obligó a detenerla manualmente. Estas pruebas extensas eran resultado de un mal ajuste inicial de los parámetros de entrenamiento, lo cual afectaba la eficiencia del agente y la convergencia del algoritmo. Se utilizó el proyecto "pendulum-qlearning" de Wandb en esta primera etapa.

Para abordar estos problemas, ajustamos los parámetros de entrenamiento, tales como la tasa de aprendizaje (LEARNING_RATE), el factor de descuento (DISCOUNT_FACTOR), el número de episodios (EPISODES), el número de iteraciones (ITERATIONS), y los parámetros de exploración (EPSILON, EPSILON_DECAY, MIN_EPSILON).

Una vez que obtuvimos un código que parecía funcionar correctamente, se realizaron pruebas ejecutando distintos conjuntos de parámetros con la misma cantidad de iteraciones para poder realizar comparaciones precisas. Para esta etapa, se utilizó el proyecto “pendulum-qlearning_V2” de Wandb. Durante estas pruebas, se evaluaron varios conjuntos de parámetros:

Resultados y Comportamiento del Código

Se comprobó que, a pesar de las variaciones en los parámetros, el código tenía un comportamiento esperado, mostrando leves variaciones en los resultados. Estas variaciones se debieron principalmente a:

- **Tasa de Aprendizaje (LEARNING_RATE):** Ajustar este parámetro afectó la rapidez con la que el agente actualizaba sus valores Q. Un valor más alto permitió al agente aprender más rápidamente, pero también hizo que las actualizaciones fueran más volátiles.
- **Factor de Descuento (DISCOUNT_FACTOR):** Este parámetro determinó cuánto valoraba el agente las recompensas futuras. Un valor más alto incentivó al agente a considerar recompensas a largo plazo, lo que fue crucial para estabilizar su política de aprendizaje.
- **Episodios (EPISODES) e Iteraciones (ITERATIONS):** Aumentar estos valores permitió al agente más oportunidades de aprender y ajustar su comportamiento, pero también incrementó el tiempo de entrenamiento.
- **Parámetros de Exploración (EPSILON, EPSILON_DECAY, MIN_EPSILON):** Estos parámetros controlaron el equilibrio entre exploración y explotación. Ajustar el decaimiento de epsilon (EPSILON_DECAY) y el valor mínimo (MIN_EPSILON) permitió al agente explorar suficientes

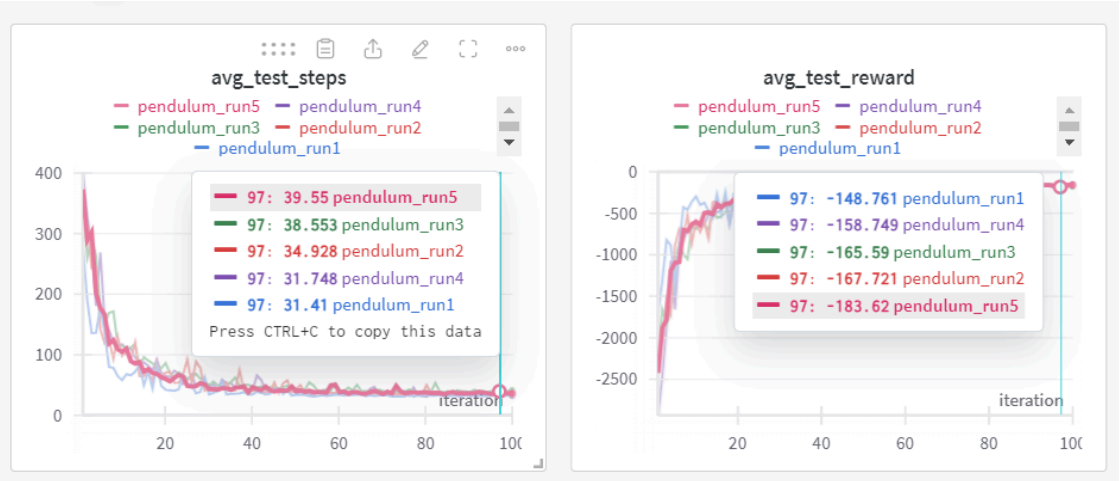
estados al principio y explotar el conocimiento adquirido en etapas posteriores del entrenamiento.

Parámetros usados por ejecución

1. LEARNING_RATE = 0.1 DISCOUNT_FACTOR = 0.99 EPISODES = 2000 TEST_EPISODES = 200 ITERATIONS = 100 EPSILON = 1.0 EPSILON_DECAY = 0.995 MIN_EPSILON = 0.01	2. LEARNING_RATE = 0.05 DISCOUNT_FACTOR = 0.98 EPISODES = 2500 TEST_EPISODES = 250 ITERATIONS = 100 EPSILON = 1.0 EPSILON_DECAY = 0.990 MIN_EPSILON = 0.01

3.
LEARNING_RATE = 0.1 # Alpha
DISCOUNT_FACTOR = 0.95 # Gamma
EPISODES = 3000 # K
TEST_EPISODES = 300 # L
ITERATIONS = 100 # N
EPSILON = 1.0 # Epsilon
EPSILON_DECAY = 0.98 # Decay rate
MIN_EPSILON = 0.05 # Minimum epsilon
NUM_DISCRETE_OBSERVATIONS = [20, 20, 200]
	4.
	LEARNING_RATE = 0.08
	DISCOUNT_FACTOR = 0.99
	EPISODES = 2500
	TEST_EPISODES = 250
	ITERATIONS = 100
	EPSILON = 1.0
	EPSILON_DECAY = 0.995
	MIN_EPSILON = 0.01
	NUM_DISCRETE_OBSERVATIONS = [15, 15, 150]

5. LEARNING_RATE = 0.12 DISCOUNT_FACTOR = 0.98 EPISODES = 3000 TEST_EPISODES = 300 ITERATIONS = 100 EPSILON = 1.0 EPSILON_DECAY = 0.996 MIN_EPSILON = 0.02 NUM_DISCRETE_OBSERVATIONS = [20, 20, 200]	



A lo largo de las pruebas, se observó que el primer conjunto de parámetros produjo los mejores resultados en términos de recompensa promedio y pasos por episodio. Esto indica que un LEARNING_RATE moderado junto con un DISCOUNT_FACTOR alto permite un buen balance entre exploración y explotación. A medida que se realizaron más iteraciones, el agente mostró un comportamiento consistente, confirmando la robustez del enfoque adoptado y la efectividad del algoritmo Q-Learning en este contexto.

Este proceso de pruebas y ajustes permitió mejorar significativamente la performance del agente y establecer una metodología sólida para futuras optimizaciones y evaluaciones.

Por último, en la carpeta “./Documentación/Pendulum” del proyecto tendremos los reportes de Wandb para todas las ejecuciones y el modelo entrenado en formato .plk. Para ejecutar el environment se debe correr el archivo “main.py” el cual se encuentra dentro de la carpeta “./Pendulum”.

Coin Game

El objetivo del ejercicio fue desarrollar un agente inteligente para el juego Coin Game utilizando algoritmos de búsqueda adversaria. Se probaron los dos algoritmos solicitados: Minimax y Expectimax, con el fin de determinar cuál de ellos ofrecía un mejor rendimiento en términos de victorias y tiempo de ejecución.

El algoritmo Minimax fue inicialmente implementado con la expectativa de que fuera más eficaz. A pesar de estas consideraciones, los resultados obtenidos con Minimax no fueron satisfactorios, con un alto número de decisiones aleatorias debido a la falta de movimientos óptimos identificados.

Dado que los resultados con Minimax no fueron satisfactorios, se decidió probar el algoritmo Expectimax. Este algoritmo es una variante del Minimax que considera los movimientos del oponente como probabilísticos en lugar de determinísticos. Este algoritmo mostró un rendimiento superior en términos de victorias y una menor cantidad de decisiones aleatorias.

Se realizaron múltiples ajustes a las heurísticas y se probaron diferentes combinaciones de factores para encontrar la mejor estrategia. Cada cambio se evaluó mediante 20 ejecuciones por nivel de dificultad.

Heurísticas Probadas

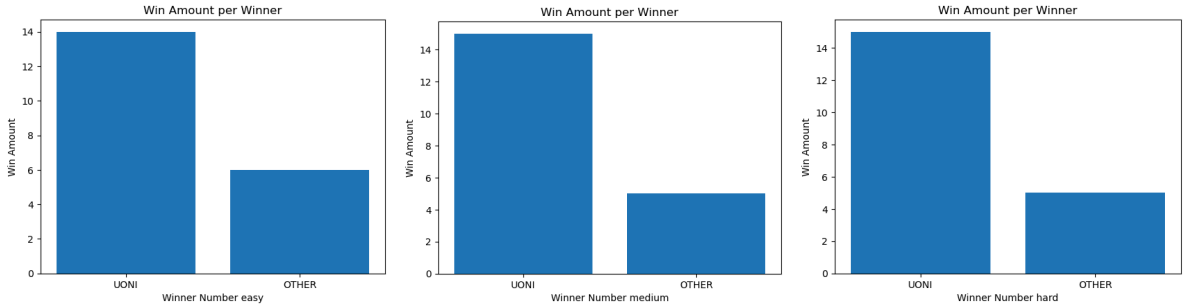
1. (Expectimax)

   Estado terminal: Si el estado es terminal, evaluar si es una victoria o derrota.

   Única moneda restante: Si hay una sola moneda, es una buena posición si es el turno del oponente.

   Paridad de filas: Contar el número de filas con un número impar de monedas y utilizar esto para evaluar la posición.

```python
def heuristic_utility(self, board, player):
    opponent = (player % 2) + 1
    is_end, winner = board.is_end(player)

    # Caso 1: Estado terminal
    if is_end:
        if winner == player:
            return -1  # Perder es malo
        else:
            return 1  # Ganar es bueno

    # Caso 2: Contar el número de monedas restantes
    total_coins = sum(1 for row in range(board.board_size[0]) for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == object)

    # Si queda solo una moneda, es una buena posición si es el turno del oponente
    if total_coins == 1:
        return 1 if player != self.player else -1

    # Caso 3: Paridad de filas
    odd_rows = sum(1 for row in range(board.board_size[0]) if sum(1 for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == objec

    if odd_rows % 2 != 0:
        return 1  # Bueno tener un número impar de filas con un número impar de monedas
    else:
        return -1  # Malo tener un número par de filas con un número impar de monedas
```



Con esta heurística concluímos que su ejecución es lenta y los resultados son desfavorables por lo que no es una buena elección.

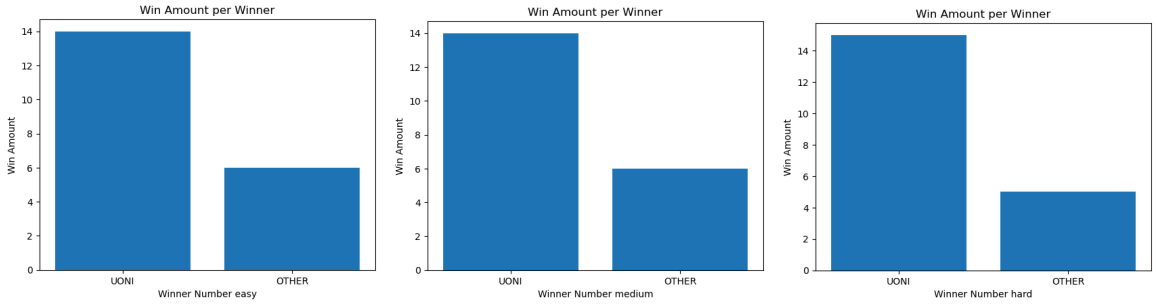
2. (Minimax)

Estado terminal: Si el estado es terminal, evaluar si es una victoria o derrota.

Número de monedas: Si solo queda una moneda es una buena posición.

Paridad de filas: Número impar de filas y número impar de monedas es una buena posición.

```python
def heuristic_utility(self, board, player):
    # Estrategia: Dejar una sola moneda para que el otro jugador pierda
    is_end, winner = board.is_end(player)
    if is_end:
        if winner == player:
            return -1  # Perder es malo
        else:
            return 1  # Ganar es bueno

    # Contar el número de monedas restantes
    total_coins = sum(1 for row in range(board.board_size[0]) for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == object)

    # Si queda solo una moneda, es una buena posición
    if total_coins == 1:
        return 1

    # Si hay un número impar de filas con un número impar de monedas, es una buena posición
    odd_rows = sum(1 for row in range(board.board_size[0]) if sum(1 for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == objec
    if odd_rows % 2 != 0:
        return 1

    return -1
```

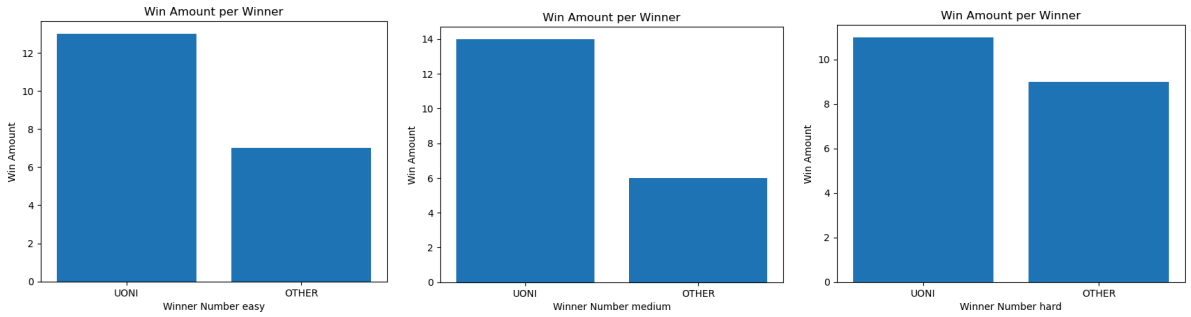


Esta estrategia parecía ser buena pero los resultados no fueron favorables. El tiempo de ejecución no fue extenso.

3. (Minimax)

Paridad de filas: Número de filas y de cantidad de monedas impar es una buena posición.

```python
def heuristic_utility(self, board, player):
    # Contar el número de filas con un número impar de monedas      You, 1 second ago • Uncommitted changes
    odd_rows = sum(1 for row in range(board.board_size[0]) if sum(1 for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == objec

    # Si el número de filas con un número impar de monedas es impar, es una buena posición para el jugador
    if odd_rows % 2 != 0:
        return 1
    else:
        return -1
```

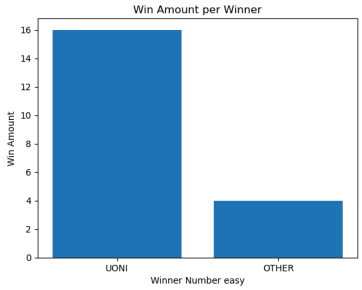


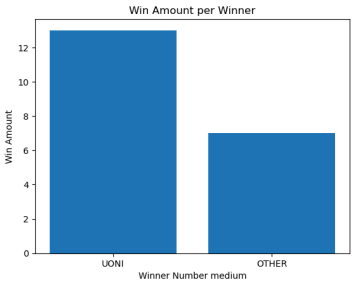
Para esta estrategia, los resultados no fueron buenos ya que en ningún nivel el agente pudo superar en victorias a las derrotas

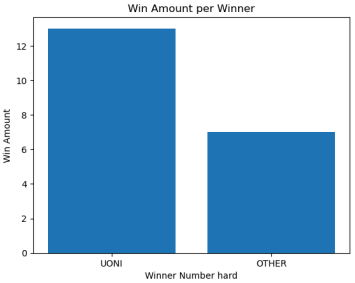
4. (Minimax)

Nim-Sum: Devuelve un valor alto si el jugador actual está en una posición ganadora y un valor bajo si está en una posición perdedora.

```python
def heuristic_utility(self, board, player):
    opponent = (player % 2) + 1
    # Calcular el Nim-Sum
    nim_sum = 0
    for row in range(board.board_size[0]):
        row_sum = sum(1 for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == object)
        nim_sum ^= row_sum

    # Si el Nim-Sum es 0, es un estado perdedor para el jugador que tiene el turno
    if nim_sum == 0:
        return -1
    else:
        return 1
```







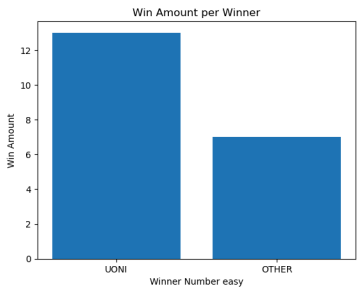
Con esta heurística, no obtuvimos resultados favorables para ninguna de las dificultades planteadas.

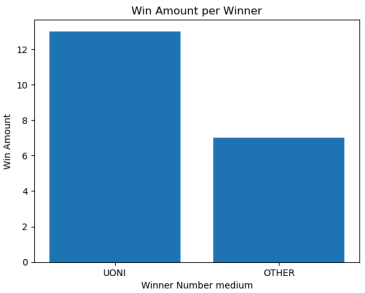
5. (Minimax)

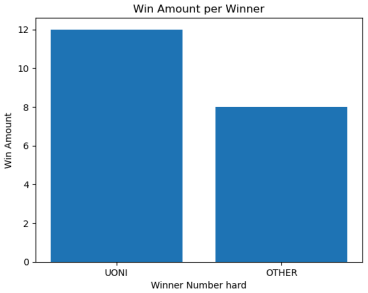
   Estado terminal: Si gana o pierde el agente.

   Cantidad de filas: Menor cantidad de filas es mejor.

```python
def heuristic_utility(self, board, player):
    if board.is_end(self.player):
        return -1  # Pérdida para el jugador actual
    elif board.is_end((self.player % 2) + 1):
        return 1  # Victoria para el jugador actual
    else:
        return -sum(board.rows)  # Por ejemplo, cuanto más vacías estén las filas, mejor
```







Obtuvimos resultados no favorables pero la ejecución fue rápida debido a la simpleza en la heurística.

6. (Expectimax) Óptima

Estado terminal: Si el estado es terminal, evaluar si es una victoria o derrota.

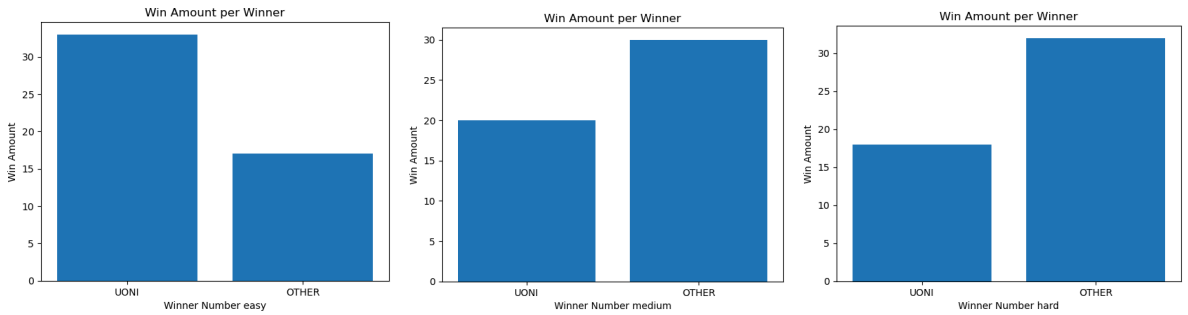
Única moneda restante: Si hay una sola moneda, es una buena posición si es el turno del oponente.

Paridad de filas: Contar el número de filas con un número impar de monedas y utilizar esto para evaluar la posición.

Nim-Sum: Evaluar la métrica Nim-Sum típica en los juegos como este,.

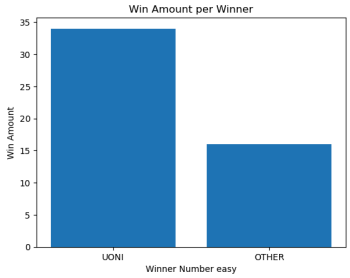
```python
def heuristic_utility(self, board, player):
    opponent = (player % 2) + 1
    is_end, winner = board.is_end(player)

    # Caso 1: Estado terminal
    if is_end:
        if winner == player:
            return 1  # Ganar es bueno
        else:
            return -1  # Perder es malo

    # Caso 2: Contar el número de monedas restantes
    total_coins = sum(1 for row in range(board.board_size[0]) for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == 1)

    # Caso 3: Paridad de filas
    odd_rows = sum(1 for row in range(board.board_size[0]) if sum(1 for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == 1) % 2

    # Evaluar el Nim-Sum
    nim_sum = 0
    for row in range(board.board_size[0]):
        row_sum = sum(1 for col in range(board.board_size[1] * 2 - 1) if board.grid[row, col] == 1)
        nim_sum ^= row_sum

    # Heurística mejorada
    utility = 0

    # Factor 1: Maximizar el número de monedas propias restantes
    utility += total_coins
```

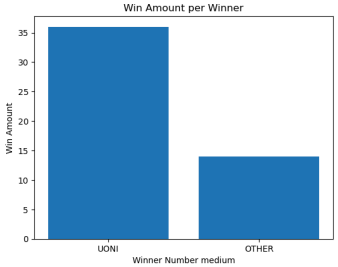
(function) board_size: Any

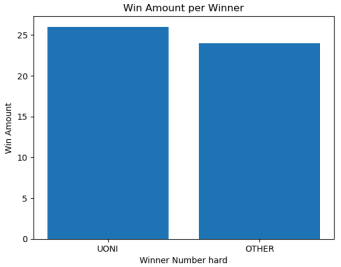


Con esta heurística obtuvimos el mejor resultado superando la cantidad de victorias a las derrotas incluso en los tres niveles en algunas pruebas. Además, la ejecución es rápida.

7. (Minimax) Misma heurística que 6.







Los resultados no fueron del todo favorables, superando siempre las derrotas a las victorias.

Tiempo de Ejecución

En promedio, las pruebas de cada conjunto de 20 ejecuciones tomaron alrededor de 3 minutos. Esto incluye el tiempo necesario para evaluar todas las posibles acciones en cada estado del juego, calcular las heurísticas y determinar el mejor movimiento para cada agente.

Conclusión

En conclusión, el uso de Expectimax con una heurística mejorada resultó en un agente más eficaz para el juego Coin, mostrando un mejor desempeño en términos de victorias y decisiones óptimas.

Análisis del Artículo "Alpha-Beta Pruning" de Carl Felstiner

1. ¿Cuál es el objetivo del autor? El objetivo del artículo es explicar cómo las computadoras pueden jugar juegos de mesa utilizando algoritmos de búsqueda como Minimax y optimizaciones como la poda Alpha-Beta. Además, se presentan otras técnicas para reducir la porción del árbol de juego que necesita ser generado para encontrar la mejor jugada.

2. ¿Cuál es el desafío que tienen las computadoras en cuanto a los árboles de juego? El principal desafío es la explosión combinatoria, donde el número de posibles configuraciones del tablero es extremadamente grande. En juegos como el ajedrez, con más de 1012010^{120}10120 posibles configuraciones, es imposible para una computadora evaluar todo el árbol de juego. Por lo tanto, es esencial encontrar formas de evitar la evaluación de ciertas partes del árbol.

3. ¿Cómo funciona el algoritmo Minimax? El algoritmo Minimax es un método recursivo utilizado para tomar decisiones en juegos de dos jugadores. Asigna valores a los nodos del árbol de juego, comenzando desde las hojas. El jugador que maximiza (Max) busca obtener el valor máximo, mientras que el jugador que minimiza (Min) busca obtener el valor mínimo. En cada turno, el algoritmo selecciona el mejor movimiento basado en los valores de los nodos hijos.

4. ¿Cómo funciona el algoritmo Alpha-Beta Pruning? La poda Alpha-Beta es una mejora del Minimax que reduce el número de nodos evaluados. Mantiene dos valores, alpha y beta, que representan los límites superiores e inferiores de los valores encontrados hasta ahora. Durante la búsqueda, si se encuentra que un nodo no puede mejorar el resultado actual (dado por alpha y beta), ese nodo se poda y no se evalúa más. Esto permite ignorar grandes porciones del árbol de búsqueda, mejorando la eficiencia del algoritmo.

5. Ejemplo de Tic-Tac-Toe con Alpha-Beta Pruning: En un juego de Tic-Tac-Toe, la poda Alpha-Beta puede identificar rápidamente que ciertos movimientos no son necesarios de evaluar completamente, ya que se puede determinar que no ofrecerán una mejor jugada que la ya encontrada. Por ejemplo, si un movimiento de un jugador lleva a una victoria inmediata o a un bloqueo de la victoria del oponente, esos movimientos se priorizan y se evita evaluar movimientos que no influyen en el resultado. Estos algoritmos se aplican no solo a Tic-Tac-Toe, sino también a otros juegos como Hex y 3D Tic-Tac-Toe, mostrando cómo las técnicas de optimización pueden reducir significativamente el tiempo de cálculo y mejorar la toma de decisiones en juegos más complejos.

6. Mejoras Adicionales en el Algoritmo: El artículo también menciona mejoras adicionales como:

- Detectar y realizar movimientos ganadores inmediatos.
- Bloquear movimientos ganadores del oponente.
- Terminar la búsqueda si se encuentra una jugada ganadora.
- Búsqueda aleatoria para mejorar la eficiencia de la poda.

Después de leer el artículo sobre Alpha-Beta Pruning de Carl Felstiner, nos dimos cuenta de cuánto se ha avanzado en el desarrollo de algoritmos que permiten a las computadoras jugar juegos de mesa de manera eficiente. Lo que más nos llamó la atención es cómo estos algoritmos no sólo son teóricos, sino que se han probado en juegos concretos como el Ta-Te-Ti. Por ejemplo, el autor muestra cómo usando la poda Alfa-Beta, una computadora puede encontrar el mejor movimiento inicial en Tic-Tac-Toe generando sólo el 0.34% de los nodos posibles.

Además, nos gustó cómo el artículo aborda las mejoras adicionales al algoritmo, como la detección de movimientos forzados y la búsqueda aleatoria, y cómo estas mejoras pueden reducir aún más el tiempo de cálculo. Estas técnicas son fundamentales para entender cómo las máquinas pueden tomar decisiones inteligentes sin necesidad de explorar cada posible resultado, lo cual es un concepto central en inteligencia artificial.

Lo más relevante para el curso de inteligencia artificial es cómo estos principios de minimización y poda se aplican en otros contextos y juegos más complejos. Nos ayuda a comprender cómo los algoritmos pueden optimizar la toma de decisiones y la búsqueda de soluciones en problemas con muchas posibilidades. Además, es una forma más divertida de aprender, lo que lo hace mucho más llevadero e interesante.

En resumen, este artículo no solo nos proporciona una base sólida en algoritmos de juegos, sino que también nos muestra aplicaciones prácticas y mejoras que son cruciales en el campo de la inteligencia artificial. Nos sentimos más preparados para entender y aplicar estos conceptos en otras materias, sabiendo que la eficiencia y la optimización son claves en el desarrollo de la inteligencia artificial.