Documento de Contribución Individual Proyecto: Agente Autónomo para Connect-4 (MCTS) **Nombre:** John Jairo Rojas Vergara

1. Resumen de Contribuciones Técnicas

Mi rol principal dentro del equipo fue garantizar la eficiencia computacional del agente. Dado que el algoritmo Monte Carlo Tree Search (MCTS) depende directamente del número de simulaciones que puede ejecutar por segundo, mi enfoque se centró en eliminar los cuellos de botella en el motor del juego y en guiar la búsqueda mediante conocimiento del dominio. A continuación, describo mis dos aportes técnicos más relevantes.

Aporte 1: Vectorización del Motor de Juego con NumPy

Contexto y Desafío El perfilado inicial del código (profiling) reveló que el 70% del tiempo de cómputo se gastaba en la función check_win. La implementación original utilizaba bucles anidados (for loops) nativos de Python para verificar las condiciones de victoria en cada dirección. Dada la naturaleza interpretada de Python, esto generaba un *overhead* masivo, limitando al agente a menos de 50 simulaciones por jugada cuando el servidor estaba bajo carga. Necesitábamos una solución que se acercara a la velocidad de C sin cambiar de lenguaje.

Implementación Técnica Reescribí completamente la clase GameLogic para aprovechar la computación vectorial de la librería NumPy. En lugar de iterar celda por celda, traté el tablero como una matriz de 6x7 y utilicé operaciones de "ventana deslizante" (sliding window) y transformaciones matriciales.

Para las diagonales, que son computacionalmente costosas de verificar iterativamente, implementé una solución basada en álgebra lineal:

1. Utilicé np.diagonal() para extraer las diagonales principales de sub-bloques de 4x4
2. Para las anti-diagonales, apliqué una transformación de espejo horizontal al tablero completo con np.fliplr() antes de extraer la diagonal, permitiendo reutilizar la misma lógica vectorizada.

Esta optimización redujo el tiempo de verificación de estado terminal en un orden de magnitud, permitiendo que el MCTS ejecute cientos de simulaciones adicionales en el mismo intervalo de tiempo.

Evidencia de Código

- **Archivo:** policy.py
- **Clase:** GameLogic, método check_win.
- **Lógica Clave:** np.all(np.fliplr(b[r-3:r+1, c:c+4]).diagonal() == player_id)
- **Commit:** feat: optimize-win-check-numpy-vectorization https://github.com/Diegosuar/Proyecto-Final/commit/003e804b193c2af2dc1f6ac2aa6ae107c5bb6686

Aporte 2: Sesgo Estratégico de Selección (Center Bias)

Contexto y Desafío Observamos que el agente MCTS "puro" tardaba demasiado en converger hacia buenas jugadas en la apertura del juego. Desperdiciaba recursos computacionales explorando las columnas laterales (0 y 6), las cuales, según la teoría de Connect-4, son subóptimas ya que ofrecen menos líneas de conexión posibles (solo 3 verticales, vs 13 líneas posibles desde el centro). El agente necesitaba una "intuición" inicial para priorizar zonas de alto valor.

Implementación Técnica Implementé un sistema de "Conocimiento del Dominio" (Domain Knowledge) inyectado directamente en la estructura del árbol de búsqueda.

Primero, modifiqué el generador de movimientos get_valid_moves para que no retornara las columnas en orden ascendente (0..6), sino ordenadas por importancia estratégica: [3, 2, 4, 1, 5, 0, 6]. Esto asegura que, ante nodos no visitados, el algoritmo siempre expanda el centro primero.

Segundo, refiné la fórmula de selección UCB1 en el método select_child. Agregué un término de desempate determinista: -abs(c.move - 3). Matemáticamente, esto significa que si dos nodos tienen valores UCB1 idénticos o muy similares, el algoritmo elegirá siempre el que esté físicamente más cerca de la columna central (índice 3). Esto actúa como una heurística de guía suave que acelera la convergencia sin romper la propiedad de exploración del MCTS.

Evidencia de Código

- **Archivo:** policy.py
- **Método:** get_valid_moves y lambda en select_child.
- **Commit:** refactor: implement-center-bias-selection https://github.com/Diegosuar/Proyecto-Final/commit/2e887bcd67d16e49e571f6062d632214f6f47dc4

2. Reflexión y Análisis de la Solución

Logros Alcanzados:La vectorización con NumPy fue el factor determinante para que nuestro agente pudiera competir en el entorno de tiempo real de Gradescope. Pasamos de sufrir *timeouts* constantes a tener un margen de seguridad holgado, incluso ejecutando lógica heurística adicional. Por otro lado, el sesgo central demostró ser efectivo en el *early-game*, permitiendo al agente construir estructuras de ataque sólidas (como el "7-trap") mucho antes que un agente aleatorio.

Limitaciones Técnicas:A pesar de las mejoras, el *overhead* de instanciación de objetos en Python sigue siendo un problema. Cada nodo del árbol crea una nueva instancia de GameLogic, lo que dispara el consumo de memoria y activa frecuentemente el *Garbage Collector*, causando micro-pausas en la búsqueda.