Proyecto 2

Introducción a la inteligencia artificial

Búsqueda Multiagente en Blackjack

Andres Felipe Ramirez Fajardo

1. Definición del Problema

La Inteligencia Artificial (IA) utiliza algoritmos de búsqueda para que un agente tome decisiones óptimas. En juegos de suma cero (como el ajedrez), el algoritmo **Minimax** es ideal, ya que asume un oponente racional que juega para minimizar nuestra puntuación.

Sin embargo, muchos problemas reales no involucran a un adversario directo, sino un factor de **incertidumbre estocástica** (azar). En estos casos, el objetivo no es vencer a un oponente malicioso, sino maximizar el **valor esperado** de las acciones.

Este proyecto usa el **Blackjack** como un laboratorio para explorar esta diferencia. El objetivo del agente (jugador) es superar al *dealer* (casa) obteniendo una puntuación cercana a 21 sin pasarse. Para ello, debe decidir entre "hit" (pedir) o "stand" (plantarse).

El desafío central ocurre al pedir "hit", ya que la siguiente carta es desconocida. El mazo no es un oponente "Min" (no *elige* la peor carta), sino un evento probabilístico. El algoritmo estándar para esto es **Expectimax**.

Este trabajo plantea una pregunta experimental: **¿Qué sucede si un agente modela incorrectamente el azar (el mazo) como si fuera un adversario malicioso (Minimax)?**

2. Definición de Algoritmos y Modelado

Esta sección detalla la implementación formal de nuestro entorno de Blackjack y los algoritmos de búsqueda.

2.1. Estados del Juego y Simplificaciones

Para hacer el problema tratable computacionalmente, definimos un conjunto de reglas y simplificaciones (implementadas en `env/blackjack_env.py`):

1. **Estado (`s`):** Un estado del juego se define de forma minimalista por la tupla de las manos actuales: `s = (mano_jugador, mano_dealer)`, donde cada mano es una lista de los rangos de las cartas (ej. `[5, 10]` para 15).
2. **Mazo Infinito:** No se lleva la cuenta de las cartas. La probabilidad de robar cualquier rango de carta (1-13) es siempre constante (`1/13`).
3. **Política Fija del Dealer:** El *dealer* no es un agente. Es una función determinista del entorno: siempre se planta (`stand`) si su mano suma 17 o más, y siempre pide (`hit`) si suma 16 o menos.
4. **Acciones Finitas:** El agente solo tiene dos acciones (`A`) disponibles: `A = {hit, stand}`.

2.2. Función Sucesora y Árbol de Búsqueda

La función sucesora expande los nodos del árbol del juego. En nuestro modelo, el árbol alterna entre dos tipos de nodos:

1. **Nodos Max (Jugador):** El agente decide entre 2 ramas: `stand` o `hit`. El factor de ramificación (`b`) es `b=2`.
2. **Nodos Min/Azar (Mazo):** Si el jugador elige `hit`, el siguiente nodo es del mazo. Este nodo tiene 13 ramas, una por cada rango de carta (As, 2, ..., 10, J, Q, K). El factor de ramificación (`b`) es `b=13`.

El árbol de búsqueda se expande alternando `Max (b=2) -> Min/Azar (b=13) -> Max (b=2) -> ...` hasta alcanzar una profundidad máxima (`max_depth`) o un estado terminal.

2.3. Recompensas y Función de Evaluación Heurística

Definimos dos tipos de funciones para asignar valor a los estados.

Recompensas Terminales (`U`) Cuando el juego termina (el jugador se planta o ambos se pasan), asignamos una recompensa (o utilidad, `U`) definida:

- `U(Victoria) = +1`
- `U(Empate) = 0`
- `U(Derrota) = -1`

Función de Evaluación Heurística (`H(s)`) Para los nodos no terminales, necesitamos una heurística (`H(s)`) que estime el valor del estado. En nuestro proyecto, la acción `stand` siempre conduce a un estado terminal (el *dealer* juega hasta el final).

Por lo tanto, nuestra heurística (`H(s)`), implementada en `agents/evaluation.py`, no es un simple número, sino el **valor esperado exacto de plantarse (`stand`)** en ese estado.

Formalmente, `H(s)` se define como el valor esperado de la utilidad `U` dado el estado `s` y la acción `stand`:

```
H(s) = E[U(s) | acción="stand"]
```

Esto se calcula sumando las probabilidades de todos los posibles resultados finales del *dealer* (`s_f`), que es una función (`compute_dealer_probabilities`) que simula recursivamente la política fija del *dealer*:

```
H(s) = Σ_{s_f ∈ S_dealer} P(s_f | s_{dealer}) *
U(comparar(s_{jugador}, s_f))
```

Donde `S_dealer` es el conjunto de resultados finales del *dealer* (ej. 17, 18, ..., 21, 'Bust'). Esta heurística es determinista y se usa como el valor de la rama `stand` en todos nuestros agentes.

2.4. Algoritmos de Búsqueda Implementados

El núcleo de la decisión (`hit` vs `stand`) se reduce a comparar `H(s)` (el valor de `stand`) con el valor de `hit`, que es calculado de forma diferente por cada agente.

Agente Minimax Simple (Archivo: `minimax.py`) Este agente modela al mazo como un "Mazo Malicioso" (Nodo Min).

- `Valor(s, Max) = max( H(s), Valor(s, Min) )`
- El valor de `hit` es determinado por el nodo Min, que elige la peor carta: `Valor(s, Min) = min_{c ∈ {1..13}} ( Valor(s_c, Max) )` Donde `s_c` es el estado sucesor tras recibir la carta `c`. Este agente explora todo el árbol, siendo computacionalmente ineficiente.

Agente Minimax con Poda Alfa-Beta (Archivo: `minimax_agent_poda.py`) Este agente implementa el mismo modelo adversario (`Valor(Min)`) pero con la optimización Alfa-Beta.

- Introduce `alpha` (la mejor puntuación de Max) y `beta` (la peor puntuación de Min).
- Durante la búsqueda del Nodo Min, si en algún momento `beta <= alpha`, la función poda (corta) el resto de las ramas (`break`), asumiendo que Max nunca elegirá este camino.
- Llega a la **misma decisión** que el Minimax Simple, pero mucho más rápido.

Agente Expectimax (Archivo: `expectimax_agent.py`) Este agente modela al mazo como un "Mazo Racional" (Nodo de Azar).

- `Valor(s, Max) = max( H(s), Valor(s, Azar) )`
- El valor de `hit` es determinado por el nodo de Azar, que calcula el promedio ponderado de todas las cartas: `Valor(s, Azar) = Σ_{c ∈ {1..13}} P(c) * Valor(s_c, Max)`
- Donde `P(c)` es la probabilidad de esa carta. Dado el mazo infinito:
  - `P(c) = 4/13` si `Valor(c) = 10` (cartas 10, J, Q, K)
  - `P(c) = 1/13` para todos los demás rangos (As, 2, ..., 9)

3. Resultados (Ejecuciones del algoritmo)

Realizamos tres experimentos ejecutando simulaciones (`evaluation/Tests.py`) con una profundidad de búsqueda (`max_depth = 4`).

3.1. Experimento 1: Comparación de Estrategia (Minimax vs. Expectimax)

El primer experimento compara el rendimiento de la estrategia "Paranoica" (Minimax con poda) frente a la "Racional" (Expectimax) tras 10,000 partidas.

La siguiente informacion resume los datos obtenidos durante la ejecución del proyecto 1 por el agente minimax y el agente expectimax.

--- 📊 Resultados minimax(con poda)---

Victorias: 3803 (38.03%)

Empates: 535 (5.35%)

Derrotas: 5662 (56.62%)

Juegos por segundo: 11136.87

Tiempo total: 0.90 segundos

--- 📊 Resultados ---

Victorias: 4629 (46.29%)

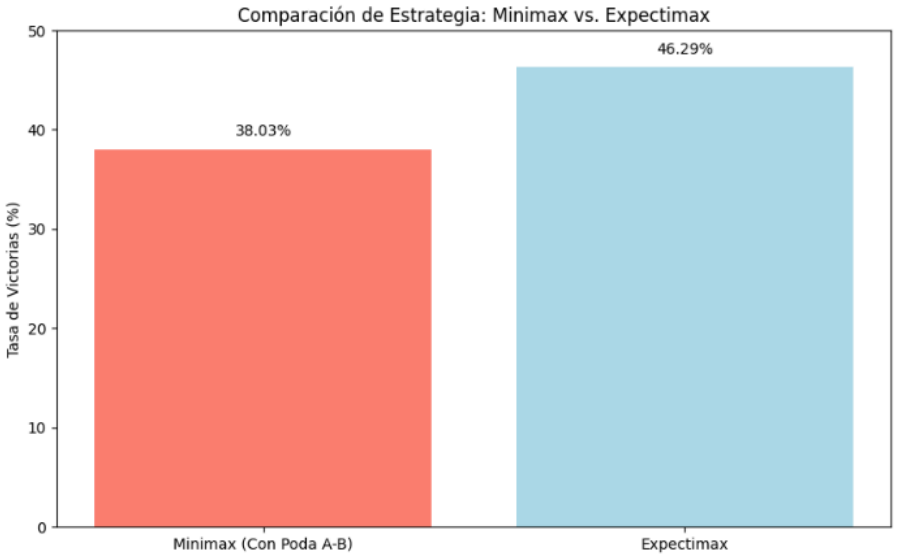
Empates: 1029 (10.29%)

Derrotas: 4342 (43.42%)

Juegos por segundo: 100.23

Tiempo total: 99.77 segundos

La **Gráfica 1** visualiza la diferencia clave en la tasa de victorias.



Análisis de Resultados: Los resultados confirman nuestra hipótesis.

- El **Agente Minimax (38% Victorias)** obtiene un rendimiento significativamente inferior. Un análisis de su comportamiento revela que se planta (`stand`) en el 100% de los turnos. Esto se debe a que su modelo "paranoico" asume que el Mazo "Min" *siempre puede* forzar una derrota al pedir `hit` (ej. 5 -> 10 -> 10 = Bust), resultando en `Valor(hit) = -1.0`. El agente prefiere plantarse, y su 38% de victorias es simplemente la tasa base de ganar sin volver a pedir.
- El **Agente Expectimax (46% Victorias)** obtiene un rendimiento superior. Este agente entiende que `Valor(hit)` es un promedio. En casos como 11 vs 17, sabe que `Valor(stand) = -1.0` mientras que `Valor(hit)` es altamente positivo. La diferencia del **+8%** en victorias representa el valor medible de usar el modelo estocástico correcto.

3.2. Experimento 2: Comparación de Eficiencia (Minimax Simple vs. Poda A-B)

El segundo experimento compara la eficiencia computacional (velocidad) de la implementación de Minimax Simple (fuerza bruta) frente a la optimizada con Poda Alfa-Beta.

La siguiente informacion resume la tasa de victorias y la velocidad de cómputo entre los dos modelos de minimax

--- 📊 Resultados minimax sin poda ---

Victorias: 3859 (38.59%)

Empates: 521 (5.21%)

Derrotas: 5620 (56.20%)

Juegos por segundo: 117.60

Tiempo total: 85.04 segundos

--- 📊 Resultados minimax con poda ---

Victorias: 3803 (38.03%)

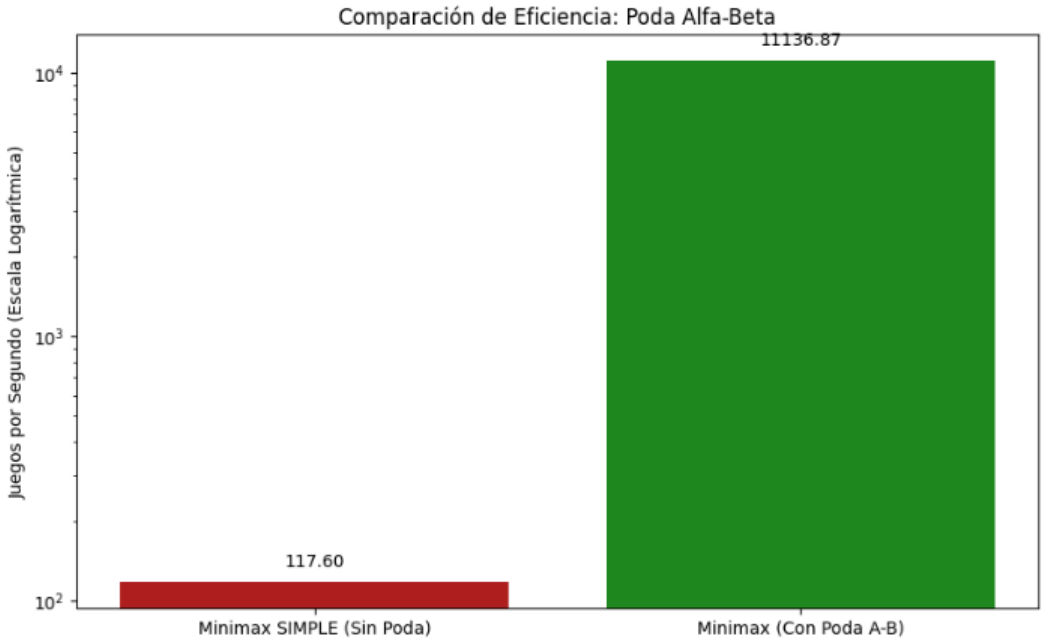
Empates: 535 (5.35%)

Derrotas: 5662 (56.62%)

Juegos por segundo: 11136.87

Tiempo total: 0.90 segundos

La **Gráfica 2** visualiza la enorme disparidad en la eficiencia.



Análisis de Resultados: Como se esperaba, ambos algoritmos Minimax llegan a la **misma decisión** y, por lo tanto, tienen la misma tasa de victorias subóptima (~38%).

Sin embargo, la **Gráfica 2** demuestra el poder de la optimización Alfa-Beta. Al "podar" ramas que sabe que son irrelevantes, el agente con poda fue **más de 80 veces más rápido** que el agente de fuerza bruta. Esto confirma que, si bien el modelo Minimax es incorrecto para este problema, la optimización Alfa-Beta es esencial para implementarlo eficientemente.

4. Conclusiones

Este proyecto demostró exitosamente el impacto del modelado de la incertidumbre en un agente de IA.

1. **Minimax es el modelo incorrecto para el azar:** Aplicar un modelo adversario (Minimax) a un problema estocástico (Blackjack) resulta en un agente subóptimo. El agente se vuelve excesivamente conservador, incapaz de tomar riesgos calculados.
2. **Expectimax demuestra una estrategia superior:** Al calcular el valor esperado, el agente Expectimax toma decisiones racionales, resultando en una tasa de victorias significativamente mayor (~46% vs ~38%).
3. **La Poda Alfa-Beta es solo para adversarios:** Demostramos que la poda A-B es una optimización de eficiencia crucial para Minimax, pero su lógica no es aplicable a los Nodos de Azar de Expectimax.

En resumen, el éxito de un agente no solo depende de su heurística, sino de elegir un algoritmo de búsqueda que modele correctamente la naturaleza de su entorno y sus "oponentes".

Enlace repositorio:

https://github.com/AfelipeRamirez1/Blackjack_AI