Puzzles, Cubo de Rubik y Búsqueda en Juegos (Temas 3 y 4)

Tema 3: Puzzles y Cubo de Rubik (Búsqueda Heurística)

Introducción a los puzzles como problemas de búsqueda: En Inteligencia Artificial, muchos puzzles se pueden modelar como problemas de búsqueda en un *espacio de estados*. Un estado representa una configuración del puzzle, y las acciones llevan de un estado a otro hasta alcanzar el estado meta (por ejemplo, el puzzle resuelto). Estos problemas suelen tener un número norme de estados posibles, lo que impide una búsqueda exhaustiva por fuerza bruta. Un ejemplo lásico es el 8-puzzle (o 15- puzzle), donde cada estado es una permutación de las fichas; otro aún más complejo es el Cubo de Rubik 3×3. Este último puzzle tiene $\sim 4.3 \times 10^{19}$ estados posibles, una cantidad astronómica que hace inviable probar todas las combinaciones por pura fuerza bruta. De hecho, está comprobado que *cualquier* posición del cubo se puede resolver en 20 movimientos o menos, número conocido como *número de Dios* (en inglés *God's number*. alcanzar esta conclusión requirió un enorme esfuerzo computacional: se particionó el espacio de 4.3×10^{19} estados en subespacios usando simetrías del cubo, y con programas especializados se exploraron enormes cantidades de posiciones (del orden de mil millones por segundo) empleando alrededor de mil millones de segundos de tiempo de CPU cedido por Google. Este resultado ilustra tanto la dificultad del problema como la necesidad de técnicas de búsqueda heurística para tratar puzzles complejos.

Búsqueda con heurísticas y heurísticas admisibles

Para abordar puzzles de gran complejidad, la búsqueda heurística introduce conocimientos adicionales mediante una *función heurística* $h(n)$ que estima el costo o distancia desde un estado n hasta la solución. Una heurística típica para puzzles de fichas es la distancia Manhattan, que suma las distancias (en movimientos ortogonales) de cada ficha desde su posición objetivo. Otra podría ser contar cuántas fichas no están en su lugar (heurística de fichas mal colocadas). Se dice que una heurística es admisible si *nunca* sobrestima el costo real de alcanzar la meta desde un estado dado. Por ejemplo, la distancia Manhattan es admisible para el 8-puzzle, ya que cualquier movimiento real de una ficha es al menos un paso en la distancia Manhattan, por lo que esta siempre es menor o igual al número real de movimientos necesarios ahmedbegggaua.github.io . La admisibilidad garantiza que algoritmos como A* (A estrella) encuentren soluciones óptimas. Una heurística más informada (que da valores más altos más cercanos a la meta) suele producir una búsqueda más eficiente (*mayor poda* del árbol de búsqueda). Sin embargo, existe un compromiso entre la potencia de la heurística y el costo de calcularla. Por ejemplo, calcular la distancia Manhattan para un estado tiene costo $O(N)$ (siendo N el número de fichas), mientras que otra heurística más precisa podría requerir $O(N^{1.5})$ operaciones ahmedbegggaua.github.io . En general, *cuanto más precisa (elevada) es la heurística admisible, más nodos puede podar*, reduciendo el espacio explorado ahmedbegggaua.github.io . Pero a su vez suele ser más costosa de computar, así que conviene encontrar un equilibrio.

Uso de patrones y bases de datos heurísticas (Pattern Databases)

Una estrategia para obtener heurísticas *muy* informadas es precomputar distancias óptimas y almacenarlas en una tabla, técnica conocida como pattern database o tabla de patrones. Esto consiste en realizar una búsqueda completa desde el estado objetivo hacia todos los demás estados (usualmente mediante BFS, búsqueda en anchura) y guardar en una tabla T la distancia mínima de cada estado alcanzable al objetivo. Por ejemplo, para el 8-puzzle (cuyo estado meta siempre es el mismo ordenamiento de fichas), se puede calcular una tabla que asocie a cada configuración el número mínimo de movimientos para resolverla. Esa tabla actúa como una heurística perfecta $h^*(n)$, ya que por definición $h^*(n)$ es exactamente el costo real desde n hasta la meta . Usar esta heurística en A* llevaría a expandir *únicamente* los nodos que están en el camino óptimo (poda máxima). De hecho, en el 8-puzzle, un A* con la tabla de patrones adecuada puede resolver cualquier configuración expandiendo pocos nodos (solo 21 nodos en un ejemplo, frente a cientos sin heurística). Además, dicha tabla permite conocer la distribución de distancias óptimas: por ejemplo, se ha calculado que el 8-puzzle requiere como máximo 31 movimientos, y la mayoría de estados aleatorios están entre 20 y 25 movimientos de la solución. No obstante, las bases de datos de patrones requieren mucha memoria y crecen exponencialmente con el tamaño del problema. Para puzzles muy grandes como el Cubo de Rubik, una tabla que cubra *todos* los estados sería gigantesca (el espacio de Rubik es de 4.3×10^{19} posiciones). Por ello, no es factible almacenar $h^*(n)$ para cada estado del cubo completo ahmedbegggaua.github.io . En su lugar, se pueden emplear pattern databases *parciales*: por ejemplo, tablas para patrones específicos (como considerar solo las esquinas del cubo, luego solo las aristas, etc.) y sumar esas heurísticas. Estas aproximaciones siguen siendo admisibles (si se diseñan adecuadamente para no sobrestimar) y logran heurísticas muy potentes sin requerir almacenar todos los estados.

Búsqueda en profundidad iterativa e IDA*

Debido a la imposibilidad de explorar explícitamente todos los estados en puzzles complejos, la búsqueda en profundidad iterativa o iterative deepening es una técnica útil. Combina las ventajas de BFS (que garantiza encontrar la solución óptima) con las de DFS (menor uso de memoria). La idea es realizar búsquedas en profundidad acotadas: primero hasta profundidad 1, luego 2, y así sucesivamente, hasta hallar la solución. Aunque parece que repetir búsquedas desde cero incrementa el trabajo, resulta que el reexplorar nodos de niveles bajos es un costo pequeño comparado con no fijar límite de profundidad (como prueba, Richard Korf demostró que la búsqueda en profundidad iterativa es óptima en orden de magnitud de tiempo y espacio entre búsquedas exhaustivas sin información). Cuando incorporamos una heurística admisible en este esquema, surge el algoritmo IDA* (Iterative Deepening A*). En IDA*, en lugar de incrementar la profundidad por niveles fijos, se incrementa el límite de costo $f(n) = g(n) + h(n)$. Es decir, primero se busca soluciones con costo hasta, por ejemplo, $f_{\max} = 10$; si no se encuentra, se incrementa el umbral y se busca con costo hasta 11, y así sucesivamente. IDA* ha sido durante mucho tiempo el método por excelencia para resolver el Cubo de Rubik, combinado con heurísticas muy informadas (a menudo derivadas de pattern databases parciales). De hecho, antes de la demostración computacional de que el número de Dios es 20, algoritmos como el de Kociemba (usado por muchos solucionadores de cubo) se basaban en una búsqueda eurística bidireccional e iterativa para encontrar soluciones en pocos segundos. En resumen, Rubik's Cube se resolvió en la práctica con búsquedas IDA* eficientes y heurísticas potentes, mucho antes de que se confirmara teóricamente la cota de 20 movimientos. Esto ejemplifica cómo la búsqueda heurística permite enfrentar puzzles complejos reduciendo drásticamente los nodos explorados, a costa de un razonable esfuerzo computacional por nodo (cálculo de la heurística).

Entropía cruzada: qué es y cómo se aplica en juegos

Un concepto importante en aprendizaje automático y juegos es la entropía cruzada (*cross-entropy*). Para entenderla, primero recordemos que la entropía (en el sentido de teoría de la información) mide la incertidumbre promedio de una distribución de probabilidad. Por ejemplo, una moneda equilibrada (50% cara, 50% cruz) tiene entropía máxima en un bit por lanzamiento (incertidumbre máxima), mientras que una moneda trucada que casi siempre cae cara tendría menor entropía (menos incertidumbre en el resultado).

La entropía cruzada mide qué *tan bien una distribución de probabilidad Q *describe* o *se ajusta* a otra distribución verdadera P**. Matemáticamente, la entropía cruzada de P respecto a Q se define como:

$$H(P, Q) = -\sum_i P(i) \log Q(i),$$

donde la suma recorre todos los eventos o clases i. En otras palabras, es el promedio (ponderado por P) del logaritmo negativo de la probabilidad que Q asigna a los eventos. Si P es la distribución real (por ejemplo, la jugada correcta en un juego) y Q es la distribución que predice nuestro modelo o agente, entonces $H(P, Q)$ cuantifica la "distancia" o discrepancia entre ambas distribuciones.

Cuanto mayor sea la entropía cruzada, mayor es la diferencia (el modelo asignó poca probabilidad a eventos que en realidad ocurren, lo cual es malo); una entropía cruzada cero significaría que Q asigna probabilidad 1 a exactamente los mismos eventos que ocurren según P (modelo perfecto). En aprendizaje automático, especialmente en clasificación, la entropía cruzada es una función de pérdida muy utilizada: el modelo intenta minimizarla para que su distribución de probabilidad predicha Q se acerque lo más posible a la distribución real P. Por ejemplo, en un juego, si un agente predice las probabilidades de posibles movimientos y la distribución real (óptima) de movimientos difiere, la entropía cruzada penaliza esas diferencias y sirve como guía para ajustar el modelo.

Aplicación en problemas de juegos: en juegos y entornos de decisión, la entropía cruzada aparece de dos formas principales:

- *Como función de pérdida para entrenar agentes o modelos predictivos:* Supongamos que tenemos un agente de Pac-Man que usa una red neuronal para decidir movimientos. Podemos entrenarlo de forma *supervisada* con ejemplos de partidas óptimas (o con el propio algoritmo minimax) para que imite esas decisiones. En tal caso, para cada estado del juego, definimos P como la distribución *objetivo* concentrada en la acción óptima (por ejemplo, si la mejor acción es "ir Norte", entonces $P(\text{Norte}) = 1$ y $P(\text{Sur}) = P(\text{Este}) = P(\text{Oeste}) = 0$). La red produce una distribución Q sobre las cuatro acciones posibles. Para ajustar la red, se calcula la entropía cruzada $H(P, Q)$. Esta será $-\log Q(\text{acción óptima})$ (pues P vale 1 en la óptima y 0 en las demás). Si la red asignó alta probabilidad a la acción correcta, la entropía cruzada será baja; si asignó baja probabilidad, la entropía cruzada será alta (penalizando fuertemente al modelo). Mediante retropropagación, la red ajusta sus pesos para minimizar esta pérdida. Así, *minimizar la entropía cruzada equivale a "aprender" la estrategia óptima* en promedio. Este enfoque se usa, por ejemplo, en métodos de aprendizaje por imitación y en etapas iniciales de AlphaGo/AlphaZero donde se entrenan redes neuronales para predecir movimientos de expertos.
- *Como método de búsqueda aleatoria (Cross-Entropy Method):* Existe una técnica de optimización estocástica llamada método de entropía cruzada (*Cross-Entropy Method*, *CEM*), que se puede aplicar a juegos o puzzles difíciles. La idea del CEM es generar muchas soluciones aleatorias, seleccionar las mejores según una métrica (por ejemplo, la puntuación del juego o la cercanía a la solución del puzzle) y luego ajustar la distribución de generación hacia esas soluciones buenas, repitiendo el proceso iterativamente. El nombre proviene de que este ajuste se realiza minimizando la entropía cruzada entre la distribución actual y una distribución "ideal" concentrada en las mejores muestras.

Intuitivamente, se comienza explorando al azar y progresivamente *se reduce la aleatoriedad centrándose en las áreas prometedoras del espacio de búsqueda*. Este método se ha utilizado en problemas de juegos combinatorios y optimización, y se menciona junto a otros heurísticos como recocido simulado y algoritmos genéticos. Por ejemplo, podría aplicarse a un puzzle como el 15-Puzzle: generar secuencias aleatorias de movimientos, quedarse con las que acercan más a la solución, y ajustar las probabilidades de los movimientos en cada posición, iterativamente, hasta encontrar la solución. Si bien el método de entropía cruzada no garantiza óptimo global, en la práctica ha demostrado ser efectivo en ciertos juegos y problemas de planificación.

En resumen, "entropía cruzada" en contextos de juegos suele referirse a medir discrepancias entre estrategias (o predicciones) y comportamientos óptimos, sirviendo para *entrenar agentes inteligentes*. Ya sea minimizando la entropía cruzada de un modelo respecto a jugadas ideales, o usando el método de entropía cruzada para explorar espacios de decisiones, este concepto conecta la teoría de la información con la resolución de problemas lúdicos.

Tema 4: Búsqueda Adversaria en Juegos (Minimax y Alfa-Beta)

Juegos de dos jugadores y búsqueda adversaria: Ahora pasamos a los juegos de adversarios (dos jugadores, cero-suma), como el ajedrez, las damas, o Pac-Man contra los fantasmas. A diferencia de los puzzles, aquí *el entorno incluye a un oponente* que busca activamente nuestra derrota. Formalmente, estos juegos cumplen: (1) Dos jugadores se alternan en turnos, con objetivos opuestos (lo que uno gana, el otro lo pierde). (2) Información perfecta y determinismo: ambos jugadores conocen las reglas, el estado actual y los posibles movimientos, y no hay azar involucrado en la evolución del estado. (3) Estados terminales claramente definidos donde el resultado es victoria para uno, derrota para otro, o a veces empate. Este marco permite modelar el problema como un árbol de juego: los nodos representan estados (configuraciones del juego), la raíz es el estado inicial, y los *hijos* de un nodo son los estados resultantes de aplicar cada movimiento legal. Los nodos de niveles pares corresponden a turnos del jugador "Max" (nosotros, que buscamos maximizar la puntuación o la probabilidad de ganar), y los niveles impares a turnos del jugador "Min" (el oponente, que busca minimizarnos la puntuación o hacernos perder).

Minimax: El algoritmo básico para decidir el mejor movimiento en este contexto es el Minimax. Este algoritmo explora el árbol de juego hasta cierta profundidad (o hasta estados terminales si es factible) y evalúa los estados finales con una *función de utilidad* o *evaluación*. A esos valores numéricos se les asigna un significado de qué tan bueno es el estado para el jugador Max (por ejemplo +1 gana, -1 pierde, 0 empate, o en juegos con puntaje, alguna estimación de la ventaja). Minimax entonces retropropaga esos valores desde las hojas hacia la raíz siguiendo estas reglas: en un nodo Max, se toma el máximo valor de entre sus hijos; en un nodo Min, se toma el mínimo. De esta manera, cada nodo interior recibe un valor que asume que a partir de ahí ambos jugadores jugarán óptimamente: Max tratando de maximizar el resultado y Min de minimizarlo. El movimiento óptimo para Max desde la raíz será aquel que conduce al hijo con el valor más alto (que representa el mejor resultado que Max puede forzar suponiendo que Min tratará de evitarlo). Análogamente, desde la perspectiva de Min, él seleccionaría en su turno la acción que lleva al valor más bajo (peor para Max, mejor para Min).

El algoritmo Minimax garantiza encontrar la estrategia óptima *si* explora exhaustivamente el árbol hasta estados terminales reales. Sin embargo, para juegos complejos como ajedrez, el número de estados crece exponencialmente con la profundidad (el *factor de ramificación* elevado al número de turnos simulados). Es inviable llegar siempre al final del juego, por lo que en la práctica Minimax se limita a cierta profundidad y usa una función de evaluación heurística para estimar el valor de los estados no terminales en la frontera explorada (por ejemplo, una evaluación de tablero en ajedrez que sume material, posición, etc.). A pesar de ello, incluso con profundidad limitada, la cantidad de nodos examinados por Minimax es grande: si cada posición tiene *b* movimientos legales en promedio (anchura o branching factor)

y analizamos *d* turnos hacia el futuro (profundidad), el número de nodos es $O(b^d)$.
Para mitigar este problema, se emplean técnicas de poda que evitan explorar ramas del árbol que no pueden influir en la decisión final.

Poda Alfa-Beta: optimización de Minimax

La poda alfa-beta es una mejora sobre Minimax que *no altera el resultado final*, pero reduce drásticamente el número de nodos evaluados. La idea clave es evitar evaluar movimientos que sabemos que no serían escogidos por al menos uno de los jugadores. Alfa-Beta logra esto manteniendo dos valores durante la búsqueda en profundidad: α (alfa) y β (beta), que representan los *mejores valores encontrados hasta ahora* para Max y para Min respectivamente en la rama en exploración. En concreto, α es el valor mínimo garantizado para Max dada la rama explorada (una cota inferior de lo que Max puede forzar), y β es el valor máximo garantizado para Min (cota superior de lo que Min puede lograr evitar que Max supere) en.wikipedia.org . Inicialmente, $\alpha = -\infty$ y $\beta = +\infty$ (sin garantías aún). A medida que se explora el árbol en profundidad, estos límites se

actualizan:

- En un nodo Max, sus hijos se evalúan y α se actualiza con el mayor valor encontrado entre ellos (Max busca maximizar su resultado). Si en algún momento α alcanza o supera a β ($\alpha \geq \beta$), significa que el nodo Min "ancestro" no permitiría llegar hasta aquí porque Min ya tiene garantizado un resultado más bajo (β) en otra rama; por tanto, *se deja de explorar más hijos* (se produce una poda). En términos prácticos: *en un nodo Max, si se encuentra un hijo con valor ≥ β, no hace falta evaluar los demás hijos,* porque el jugador Min nunca dejaría a Max alcanzar al menos ese valor β (Min tendría opción mejor en otra rama).
- Simétricamente, en un nodo Min, se eligen los hijos con el menor valor y β se actualiza al mínimo valor encontrado. Si β desciende por debajo de α ($\beta \leq \alpha$), significa que el jugador Max "ancestro" no permitiría llegar a este nodo porque ya tiene garantizado un resultado más alto en otra rama (α); entonces *podamos* el resto de hijos. Es decir, *en un nodo Min, si se encuentra un hijo con valor ≤ α, se omite la exploración de los demás,* ya que Max tendría una alternativa mejor en otra rama y no permitiría que Min lo forzara a este camino peor.

De este modo, alfa-beta elimina ramas enteras del árbol sin perder exactitud en la decisión final. En el mejor de los casos, la poda alfa-beta reduce el número de nodos examinados de b^d a aproximadamente $b^{d/2}$, doblando efectivamente la profundidad de búsqueda con el mismo esfuerzo (por ejemplo, en ajedrez se suele decir que alfa-beta permite mirar dos veces más jugadas adelante que Minimax puro con el mismo tiempo). Esto ocurre cuando los movimientos se examinan en el orden más favorable (primero las mejores opciones que llevan a poda máxima). En el peor caso (orden de exploración muy desfavorable), alfa-beta evaluaría el mismo número de nodos que Minimax, pero *nunca más*. En general, con ordenación razonable de movimientos (por ejemplo, explorando primero las jugadas que la heurística sugiere como mejores), la poda ahorra muchísimo trabajo.

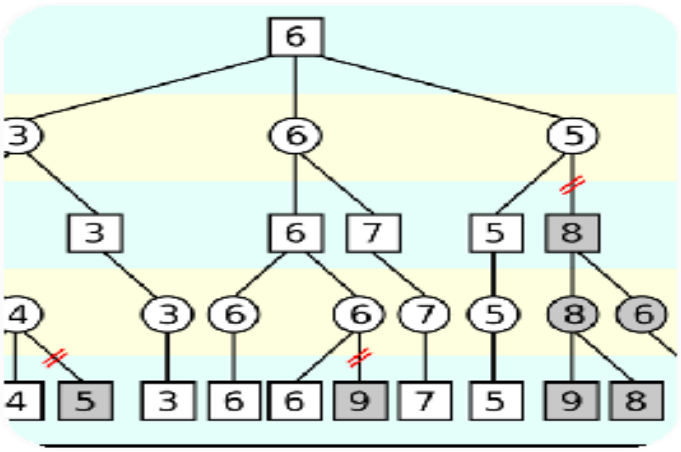

Figura: Ejemplo de árbol de juego con algoritmo minimax y poda alfa-beta aplicada. Los nodos sombreados en gris con una "X" roja representan ramas podadas (no se exploran) porque se determinó que no afectan al resultado óptimo. En este ejemplo, el valor propagado en la raíz es 6 (MAX busca maximizar). Gracias a la poda alfa-beta, varias hojas (nodos del último nivel) no necesitan ser evaluadas completamente, ya que se encontró que ciertas ramas MIN ya ofrecían valores peores (para MAX) de lo que MAX podía asegurar por alternativas.

En la figura, se observa cómo en los niveles MIN algunas ramas se cortan (por ejemplo, en la rama derecha, un valor alto encontrado para MAX hizo innecesario explorar otro hijo porque MIN ya tenía garantizado un resultado menor en la rama izquierda). El algoritmo recorre el árbol en profundidad: tan pronto como una rama no puede superar una alternativa ya encontrada, se descarta. A pesar de que en el dibujo el árbol es pequeño, en juegos reales la reducción es sustancial.

En resumen, poda alfa-beta mejora a Minimax filtrando movimientos subóptimos: *no afecta a la decisión final (el mejor movimiento sigue siendo el mismo), pero ahorra tiempo de cálculo*. Junto con técnicas de ordenación de movimientos (por ejemplo, probar primero capturas favorables en ajedrez, o movimientos que acercan a la meta en otros juegos) y memorias transposicionales (tablas que almacenan valores ya calculados para estados repetidos), la poda alfa-beta es esencial para los motores de juego modernos. En Pac-Man, por ejemplo, donde Pac-Man (MAX) juega contra varios fantasmas (MIN múltiples), la misma idea se extiende: los fantasmas pueden modelarse como un jugador Min cooperativo o como varios niveles MIN consecutivos en el árbol (uno por cada fantasma). El algoritmo alfa-beta se adapta manejando varios niveles MIN seguidos (el valor se minimiza sucesivamente por cada fantasma). La eficiencia sigue siendo muy superior a Minimax puro, permitiendo que Pac-Man *con AI* planifique varios movimientos por adelantado incluso con múltiples adversarios.

Ejercicios Resueltos

Ejercicio 1: Cálculo de entropía cruzada en un juego binario

Enunciado: Supongamos que estamos diseñando un agente para jugar un juego sencillo donde solo hay dos resultados posibles al final de la partida: Ganar o Perder. Queremos que el agente estime antes de jugar la probabilidad de ganar. En un cierto estado del juego, el agente asigna una probabilidad $p(\text{Ganar}) = 0.8$ (y por tanto $p(\text{Perder}) = 0.2$). Imaginemos que, en realidad, la acción óptima en ese estado garantiza la victoria (es decir, *si juega bien, debería ganar con probabilidad 1*). Calcula la entropía cruzada entre la distribución predicha por el agente y la distribución real del resultado, e interpreta su significado. Luego, considera qué ocurriría con la entropía cruzada si el agente subestima drásticamente sus chances, por ejemplo asignando $p(\text{Ganar}) = 0.3$.

Resolución: Primero, definamos las distribuciones:

- Distribución real P: Dado que la estrategia óptima lleva a una victoria segura, la distribución verdadera de resultado es $P(\text{Ganar}) = 1$ y $P(\text{Perder}) = 0$. (Esto equivale a un one-hot en "Ganar".)
- Distribución predicha Q: El agente cree $Q(\text{Ganar}) = 0.8$, $Q(\text{Perder}) = 0.2$

La fórmula de entropía cruzada para distribuciones binarias $P = (p,\ 1-p)$ y $Q = (q,\ 1-q)$ es:

$$H(P, Q) = -[p \log q + (1-p) \log(1-q)].$$

En nuestro caso, $p = 1$ (la probabilidad real de ganar es 1), así que la fórmula se simplifica a:

$$H(P, Q) = -[1 \cdot \log Q(\text{Ganar}) + 0 \cdot \log Q(\text{Perder})] = -\log(0.8).$$

Calculamos este valor: $-\log(0.8)$. Asumiendo logaritmo natural (base e), $\log(0.8) \approx -0.2231$, así que $-\log(0.8) \approx 0.2231$. Si utilizáramos log base 2 (bits), $-\log_2(0.8) \approx 0.3219$ bits. Interpretación: La entropía cruzada 0.2231 (en nats) es el "coste de sorpresa" de que, en realidad, el resultado es ganar con certeza cuando el agente no estaba completamente seguro. Es relativamente baja porque el agente asignó un 80% de probabilidad a ganar, lo cual es bastante cercano a la realidad (100%). En esencia, la entropía cruzada aquí nos está diciendo que el agente *todavía cometió algo de error (no estaba 100% seguro de la victoria), pero no demasiado.*

Ahora, si el agente hubiese predicho $Q'(\text{Ganar}) = 0.3$ y $Q'(\text{Perder}) = 0.7$ en la misma situación (subestimando sus chances dramáticamente), la entropía cruzada sería:

$$H(P, Q') = -[1 \cdot \log(0.3) + 0 \cdot \log(0.7)] = -\log(0.3).$$

Aquí $-\log(0.3)$ en base e es aproximadamente 1.2046. Es mucho mayor que 0.2231. Interpretación: Una entropía cruzada más grande indica que la distribución del agente está mucho más alejada de la realidad. En este caso, el agente asignó solo 30% a ganar cuando en verdad era 100%, así que su predicción fue muy mala. La entropía cruzada alta penaliza fuertemente este error. Si estuviésemos entrenando al agente con entropía cruzada como función de pérdida, una pérdida de ~1.20 vs ~0.22 indica que el segundo modelo (el que daba 80% a ganar) es mucho mejor que el primero (que daba 30%). En términos informativos, significa que usar el "código" del modelo para los resultados verdaderos sería ineficiente, requiriendo ~1.20 nats en promedio, frente a ~0.22 nats cuando el modelo es más preciso.

En resumen, la entropía cruzada en este ejercicio cuantifica el grado de error en la estimación de la probabilidad de ganar. Con $p = 0.8$ la entropía cruzada es baja (el agente casi acierta); con $p = 0.3$ la entropía cruzada es alta (el agente se "equivoca" mucho en su predicción). Esto demuestra cómo la entropía cruzada sirve para medir la calidad de las predicciones de un agente en un j.bin

Ejercicio 2: Entropía cruzada en distribución de movimientos de un juego

Enunciado: Considera un estado de un juego con tres movimientos posibles $\{A, B, C\}$. Un agente de IA predice las siguientes probabilidades para cada movimiento: $Q(A) = 0.6$, $Q(B) = 0.3$, $Q(C) = 0.1$. Sin embargo, supongamos que un experto (o un algoritmo de búsqueda exhaustiva) nos dice que en ese estado el movimiento óptimo es B con probabilidad 1 de llevar a la mejor outcome (es decir, B es claramente mejor que A o C). Calcula la entropía cruzada entre la distribución predicha por el agente y la distribución "real" (óptima), e interpreta el resultado. Luego, analiza cómo cambiaría ese valor si el agente hubiera predicho mejor dando más peso al movimiento B.

Resolución: Primero definamos las distribuciones de probabilidad sobre $\{A, B, C\}$:

- Distribución real P: Dado que B es el movimiento óptimo seguro, podemos representarlo como $P(B) = 1$ y $P(A) = P(C) = 0$. (Toda la probabilidad está en B; en otras palabras, la política perfecta elegiría B con un 100% de probabilidad en este estado).
- Distribución del agente Q: $Q(A) = 0.6$, $Q(B) = 0.3$, $Q(C) = 0.1$. El agente está dando prioridad al movimiento A, pero según el experto eso es un error porque B es mejor.

La entropía cruzada $H(P, Q)$ con P concentrada en B se reduce a $-\log Q(B)$ (similar al ejercicio anterior pero con tres acciones en lugar de dos). Esto es porque $P(B) = 1$ y los demás términos con $P(A)$ y $P(C)$ son cero. Así:

$$H(P, Q) = -[1 \cdot \log Q(B) + 0 \cdot \log Q(A) + 0 \cdot \log Q(C)] = -\log(0.3).$$

Calculamos $-\log(0.3)$ (base natural): ya vimos en el ejercicio 1 que $-\ln(0.3) \approx 1.2046$. En términos de bits, sería $-\log_2(0.3) \approx 1.737$ bits. Este valor es la entropía cruzada entre la política del agente y la política óptima. Interpretación: Es bastante alto, indicando que el agente asignó baja probabilidad (0.3) al movimiento correcto B. El agente está "desperdiciando" probabilidad en A (0.6) y algo en C (0.1), mientras que debería haber puesto toda la probabilidad en B.

Ahora, ¿qué pasaría si el agente hubiese predicho mejor, por ejemplo $Q'(A) = 0.1$, $Q'(B) = 0.8$, $Q'(C) = 0.1$ (dando 80% al movimiento B)? Entonces la entropía cruzada sería:

$$H(P, Q') = -\log Q'(B) = -\log(0.8).$$

Como antes, $-\log(0.8) \approx 0.2231$ en base e. Esto es mucho menor que 1.2046. Significa que Q' está mucho más cerca de P que Q original. En este caso, el agente habría predicho correctamente con alta confianza el movimiento óptimo B, y la entropía cruzada refleja ese pequeño error (aún no es perfecto porque no puso 100% en B, pero 0.8 es bastante cercano).

Conclusión: En la primera situación (0.3 a B), la entropía cruzada ~1.20 indica un error significativo en la estrategia del agente: su distribución de movimientos difiere bastante de la óptima. En la segunda situación (0.8 a B), la entropía cruzada ~0.22 indica un error mucho menor. Esto cuantifica formalmente la intuición de que en la primera predicción el agente "no entendió" cuál era la mejor jugada, mientras que en la segunda casi lo entendió. En entrenamiento de IA para juegos, uno ajustaría los parámetros del agente para pasar de la primera distribución a la segunda, precisamente reduciendo la entropía cruzada mediante aprendizaje.

Ejercicio 3: Traza del algoritmo alfa-beta (orden óptimo de búsqueda)

Enunciado: Dado el siguiente árbol de juego sencillo, simular el algoritmo minimax con poda alfa-beta indicando los valores de α y β que se van actualizando y qué ramas se podan. El árbol de juego (valores estáticos en las hojas) es:

```scss
Max (nodo raíz)
├── Min (nodo L)
│   ├── [hoja valor = 5]
│   └── [hoja valor = 8]
└── Min (nodo R)
    ├── [hoja valor = 4]
    └── [hoja valor = 7]
```

Copiar

Supongamos que el algoritmo explora primero la rama izquierda (nodo L) antes que la derecha. Mostrar paso a paso cómo funciona la poda alfa-beta en este caso, e indicar el valor final escogido por la raíz y qué nodos hoja no fueron necesarios evaluar.

Resolución: Vamos a recorrer el árbol con una búsqueda en profundidad controlada por los límites α y β. El orden de exploración dado es: primero la rama L (todas sus hojas), luego la rama R (hasta donde sea necesario).

Inicializamos $\alpha = -\infty$ y $\beta = +\infty$ en la raíz antes de comenzar.

1. Nodo raíz (Max): $\alpha = -\infty$, $\beta = +\infty$. Comenzamos explorando su primer hijo (rama L).
2. Nodo L (Min): Al entrar en este nodo, le pasamos los valores actuales: $\alpha = -\infty$ (heredado de arriba, mejor valor para Max conocido) y $\beta = +\infty$ (heredado también, mejor valor para Min conocido hasta ahora). Este es un nodo Min, significa que su objetivo es minimizar el resultado. Exploramos sus hijos en orden:
   - Primer hijo de L: hoja con valor 5.

     Al evaluar esta hoja, obtenemos un valor = 5. Volvemos al nodo L: como L es Minimizador, actualiza su valor β con el mínimo encontrado hasta ahora. Inicialmente $\beta = +\infty$, ahora $\beta = \min(+\infty, 5) = 5$. También, observamos que este valor 5 es propagado hacia arriba como posible resultado para Max si escoge L. En el nodo L: α sigue siendo $-\infty$ (no se utiliza α en un Min directamente para actualizar, α es cota de Max, β es cota de Min). Tras este hijo, para nodo L tenemos: $\alpha = -\infty$, $\beta = 5$ (valor mínimo provisional). No se produce poda aún porque la condición de poda en nodo Min es $\beta \leq \alpha$, lo que aquí es $5 \leq -\infty$ (falso). Continuamos con el siguiente hijo.
   - Segundo hijo de L: hoja con valor 8.

     Evaluamos esta hoja, valor = 8. Regresando a nodo L, actualizamos $\beta = \min(\text{valor actual } \beta, 8) = \min(5, 8) = 5$. Es decir, el nodo L ya tenía un valor mínimo provisional de 5 tras el primer hijo, y el segundo hijo tiene valor 8, así que no lo reduce: el mínimo sigue siendo 5. Condiciones de poda: comprobamos $\beta \leq \alpha$ en L $\rightarrow 5 \leq -\infty$ sigue falso, por lo que no había poda dentro de L. Hemos terminado de explorar los hijos de L.

   Después de ambos hijos, el valor del nodo L (Min) es $\beta = 5$ (el mínimo de $\{5,8\}$). Este valor es retornado hacia el nodo raíz.

3. Volviendo al nodo raíz (Max): Hemos obtenido un valor 5 de su hijo izquierdo

   (L). Ahora el nodo raíz, siendo Max, actualiza su α con el máximo encontrado: α = max(-∞, 5) = 5. Este 5 significa: "si el jugador Max elige la rama L, puede asegurar un resultado de 5". Ahora α=5 en la raíz, β sigue siendo +∞ (en la raíz, β representa la mejor garantizado para Min, pero en la raíz no hay un oponente arriba, así que podemos considerar β = +∞ aún).

   - Decisión de poda en raíz: En un nodo Max la poda ocurre si α ≥ β. Aquí $\alpha = 5$, $\beta = +\infty$, claramente $5 \geq +\infty$ es falso. Por tanto, seguimos. Ahora procedemos a explorar el segundo hijo del raíz: rama R.

4. Nodo R (Min): Llamamos al segundo hijo, nodo R, que es de tipo Min (jugada del oponente). Al entrar, heredamos los límites actuales del padre: α = 5 (¡ojo! la raíz tenía α=5 ahora) y β = +∞ (la raíz tenía β=+∞). Entonces en nodo R comenzamos con α=5, β=+∞. Como es Min, intentará minimizar. Exploramos sus hijos:

   - Primer hijo de R: hoja con valor 4.

     Evaluamos este nodo hoja, valor = 4. Volvemos a R: actualizamos β = min(+∞, 4) = 4. Ahora en nodo R tenemos α = 5 (heredado) y β = 4 (valor mínimo provisional encontrado). *Aquí comprobamos condición de poda:* en nodo R (Min) se poda si $\beta \leq \alpha$. Efectivamente, ahora β = 4 y α = 5, así que $4 \leq 5$ es verdadero. Esto indica que el nodo R (como Minimizador) ya encontró una rama (este primer hijo) que da un valor 4 a Max. Desde la perspectiva de la raíz (Max), recuerde que α=5 era el valor que Max podía obtener y está garantizado por escoger L. Ahora estamos evaluando la opción R; vemos que Min (en R) podría forzar a Max a obtener 4 o menos (de hecho 4) en esa rama. Como Max ya tiene garantizado 5 eligiendo L, nunca se arriesgaría a ir por R si R le va a dar 4 como mucho. Por lo tanto, no es necesario explorar el resto de hijos de R, porque el jugador Max no elegirá R sabiendo que la rama L le asegura 5. Se activa la poda alfa-beta: en nodo R, $\beta \leq \alpha \rightarrow$ rama R corta la exploración. No examinamos el segundo hijo de R.

   - Segundo hijo de R: (hoja valor 7) No se explora debido a la poda anterior. Queda marcado como podado, ya que no hace falta conocer su valor para tomar la decisión óptima.

   Tras la poda, el valor del nodo R se considera 4 (β = 4, basado solo en el primer hijo evaluado).

5. Nodo raíz (Max) concluye: El nodo R devolvió un valor 4 (aunque no completo, ese 4 ya era suficiente para descartar R). El nodo raíz toma el máximo entre su opción L (que valía 5) y R (que valdría 4). max(5, 4) = 5. Así, el valor final en la raíz es 5, y corresponde a la acción L. El algoritmo Minimax con poda alfa-beta recomendaría elegir la rama L, con resultado 5.

Resultado: El algoritmo determinó que la mejor acción es la rama L, con valor 5, y poda la evaluación del segundo hijo de R (valor 7) porque ya se sabía que la rama R no superaría lo obtenido por L. En total, se evaluaron 3 hojas en vez de las 4 totales gracias a la poda. Con el orden de exploración dado (primero L, luego R), la poda ocurrió lo antes posible en R, ahorrando la mitad del trabajo en la rama derecha.

Verificación manual: Sin poda, minimax habría evaluado todas las hojas:

- Rama L producía valor min(5,8) = 5.
- Rama R producía valor min(4,7) = 4.
- max elegiría max(5,4) = 5 por L.

  La poda simplemente evitó evaluar el 7 porque ya sabíamos que $R \leq 4 < 5 = L$. El algoritmo llega al mismo resultado (elegir L) pero con menos evaluaciones.

Ejercicio 4: Importancia del orden de exploración en alfa-beta

Enunciado: Considera el mismo árbol del ejercicio 3, pero ahora analicemos qué ocurriría si el orden de exploración de los hijos en la raíz fuera el opuesto: es decir, el algoritmo examina primero la rama R y después la rama L. Simula el algoritmo alfa- beta en ese caso y compara la cantidad de nodos evaluados respecto al caso anterior.
¿Se poda alguna rama? ¿Cuál es el valor final?

Resolución: Usaremos nuevamente el árbol:

```scss
Max (raíz)
├── Min (L)
│   ├── [5]
│   └── [8]
└── Min (R)
    ├── [4]
    └── [7]
```

Esta vez el orden de búsqueda es: primero rama R, luego rama L. Sigamos los pasos:

1. Nodo raíz (Max): inicial $\alpha = -\infty$, $\beta = +\infty$. Explora primero hijo R.
2. Nodo R (Min): entra con $\alpha = -\infty$, $\beta = +\infty$ (heredados de la raíz). Explora sus hijos:
   - Primer hijo de R: hoja valor 4.

     Evalúa a 4. Actualiza $\beta = \min(+\infty,4) = 4$. Ahora en R: $\alpha = -\infty$, $\beta = 4$. Comprueba poda: condición en Min es $\beta \leq \alpha \rightarrow 4 \leq -\infty$ falso. No poda aún. Sigue al siguiente hijo.
   - Segundo hijo de R: hoja valor 7.

     Evalúa a 7. Actualiza $\beta = \min(4,7) = 4$ (el mínimo entre 4 y 7 sigue siendo 4). En R: $\alpha = -\infty$, $\beta = 4$. Poda: chequea $4 \leq -\infty$, falso. Ya no hay más hijos. Termina nodo R con valor $\beta = 4$.

   El nodo R devuelve 4 al padre (Max).
3. Nodo raíz (Max): recibe valor 4 de R. Actualiza $\alpha = \max(-\infty, 4) = 4$. (Max sabe que al menos puede obtener 4 si elige R). Ahora procede a explorar el siguiente hijo, L, con los límites actualizados $\alpha = 4$, $\beta = +\infty$.
4. Nodo L (Min): entra con $\alpha = 4$, $\beta = +\infty$ (heredados de raíz). Explora hijos:
   - Primer hijo de L: hoja valor 5.

     Evalúa a 5. Actualiza $\beta = \min(+\infty,5) = 5$. Ahora en L: $\alpha = 4$, $\beta = 5$. Comprueba poda: condición Min $\beta \leq \alpha \rightarrow 5 \leq 4$ es falsa, así que no poda. Continúa.
   - Segundo hijo de L: hoja valor 8.

     Evalúa a 8. Actualiza $\beta = \min(5,8) = 5$ (el menor sigue siendo 5). En L: $\alpha = 4$, $\beta = 5$. Comprueba poda: $5 \leq 4$ falso. Terminan los hijos.

El nodo L devuelve valor 5 al padre.

5. Nodo raíz (Max): recibe valor 5 de L. Actualiza $\alpha = \max(4, 5) = 5$. Ahora $\alpha = 5$, $\beta = +\infty$ en la raíz. (Ya exploramos ambos hijos, realmente β en la raíz nunca se usó porque no hay nivel arriba).

Decisión final: Max compara sus opciones: rama R daba 4, rama L da 5, así que escoge L con valor 5. El resultado óptimo sigue siendo valor 5 por la rama L (igual que antes). Cantidad de nodos evaluados: En este recorrido, el algoritmo evaluó todas las hojas: 4, 7, 5 y 8. ¿Hubo poda? No, no se produjo ninguna poda en este caso. ¿Por qué? Porque exploramos primero R, que resultó ser peor rama que L. Veamos: tras R, α del Max era 4. Cuando fuimos a L (Min), ninguna hoja de L produjo un valor ≤ 4 que permitiera podar; de hecho el primer hijo de L dio 5, y ahí en L teníamos $\alpha=4$, $\beta=5$, la poda habría ocurrido si $\beta\,(5) \leq \alpha\,(4)$, lo cual no es cierto. Terminamos explorando ambos hijos de L. Irónicamente, aunque *al final* L tenía valor $5 > \alpha$, la poda en nodo Max ocurre solo cuando un hijo *supera* β, pero en la raíz $\beta=+\infty$ todo el tiempo, así que nunca se podó nada.

Comparación: En el ejercicio 3 (orden L luego R) se evaluaron 3 hojas gracias a una poda temprana en R. En este ejercicio 4 (orden R luego L) se evaluaron las 4 hojas completas, sin podas. Esto muestra claramente la importancia del orden:

- Cuando exploramos primero la rama "buena" para Max (L con valor alto), establecimos un α alto temprano ($\alpha=5$) y al ir luego por R, pudimos cortar esa rama en cuanto encontramos algo peor que 5.
- En cambio, cuando exploramos primero la rama "mala" (R con valor bajo), α quedó en 4. Luego en L, aunque encontramos 5 (mejor), eso no permite podar retrospectivamente R que ya exploramos entera. Dentro de L no pudimos podar porque su peor caso 5 nunca fue $\leq \alpha\,(4)$ hasta terminar todos sus hijos. Solo al final actualizamos α a 5.

En resumen, el orden de exploración influye en la eficiencia de alfa-beta pero no en el resultado. Ambos órdenes dan como resultado final valor 5 (mejor movimiento L). Sin embargo, con orden óptimo (explorar primero ramas que Max sospecha que son buenas), la poda evitó trabajo (no se examinó una hoja). Con orden adverso, no se ahorró nada. En problemas grandes, un buen orden puede ahorrar exponencialmente más trabajo, por eso en la práctica se usan heurísticas para ordenar movimientos (por ejemplo, en ajedrez expandir primero capturas o jugadas que parecen prometedoras).

Conclusión: En este árbol, el algoritmo alfa-beta con orden (L luego R) evaluó 3 hojas gracias a 1 poda, mientras que con orden (R luego L) evaluó 4 hojas (sin podas). El resultado elegido (Max = 5 por L) fue el mismo. Esto ejemplifica cómo alfa-beta siempre encuentra la jugada óptima, pero su rendimiento (nodos analizados) mejora drásticamente si las mejores jugadas se consideran primero.