Trabajo Corto: Algoritmos A y Min-max*

Informe de resultados

Inteligencia Artificial, GR 1

Prof. Ing. Kenneth Obando R.

Instituto Tecnológico de Costa Rica

Julián Rodríguez Sarmiento, c.2019047635

Samuel Valverde Arguedas, c.2022090162

II Semestre 2025

Índice

1. Introducción

El presente trabajo aborda la resolución de dos problemas clásicos mediante algoritmos de búsqueda:

1. **Peg Solitaire**, resuelto con el algoritmo A*.
2. **Lines and Boxes (Timbiriche)**, resuelto con Minimax con poda alpha-beta.

El objetivo fue diseñar e implementar las funciones principales de cada algoritmo, resolver los juegos de forma eficiente, diseñar interfaces de usuario simples y recolectar métricas de desempeño (tiempo, nodos expandidos, tasa de éxito) de los distintos algoritmos con parámetros variados. Asimismo, se buscó evidenciar cómo las heurísticas y técnicas de poda influyen directamente en la eficiencia de la búsqueda.

2. Implementación

2.1 Peg Solitaire (A*)

Representación del estado: el tablero se parametrizó como una cruz de tamaño n, generando un grid de (2n+1) × (2n+1). El centro comienza vacío siempre.

Movimientos: definidos como saltos ortogonales donde una ficha salta sobre otra hacia un espacio vacío.

Aplicación de movimientos: encapsulada en el método apply, que devuelve un nuevo estado inmutable.

Estado objetivo: quedar con una única ficha ubicada en el centro.

Heurísticas implementadas:

- **h_min_moves:** número de fichas restantes menos 1. Admisible y consistente, pero muy débil.
- **h_manhattan:** suma de las distancias Manhattan de cada ficha al centro. Más informativa y todavía admisible.
- **h_combo:** combinación de las dos anteriores. Mejora la eficiencia, pero no siempre es admisible.

Mejoras:

- Se introdujo un parámetro max_expansions para abortar búsquedas excesivamente largas.
- Se recolectaron métricas de cada corrida: tiempo, nodos expandidos, nodos generados, tamaño máximo de la frontera, éxito o fracaso, y si la búsqueda fue abortada.

Interfaz: se desarrolló un UI con Streamlit, donde el usuario puede visualizar el tablero y recorrer la solución paso a paso. Por limitaciones de tiempo de cómputo, el UI solo permite tableros pequeños (n=3).

2.2 Lines and Boxes (Minimax con poda alpha-beta)

Representación del estado: el tablero está diseñado en forma de "rowsXcols" pero en dimensiones de cajas y no de puntos para facilitar el manejo durante el juego, es una clase y los atributos más importantes son:

- "horizontal_edges" una matriz (rows+1 x cols) booleana para saber cuando un segmento está dibujado.

- Con la misma idea se tiene “vertical_edges” pero con rows x (cols+1).
- “box_owner” que le asigna cada caja al jugador que la completó.
- “player” que es el atributo para saber de quien es el turno.
- “scores” maneja el puntaje por cajas ganadas.
- “history” maneja el camino hacía el estado objetivo.

Movimientos: Se realizó una clase movimiento en la cual está la dirección de la línea dibujada ya sea horizontal o vertical, la posición en la que se realiza en la matriz, el jugador que la dibujó y si este movimiento completó una caja o no. Esta clase movimiento es la que se utiliza en el atributo de historial de la clase “board” y es una lista de movimientos.

Aplicación de movimientos: Se utilizan funciones para verificar si el movimiento es válido y se una la función “make_move” la cual va a recibir una dirección y una posición del tablero, al estar dentro de la clase “board” puede sacar el turno del jugador y el estado del tablero.

Estado objetivo: Que se dibujen todas las líneas posibles del tablero, en un tablero cuadrado sería la completitud de las cajas.

Heurísticas implementadas: Las que se implementaron fueron de acuerdo al juego y son simples, primero es una con la diferencia de puntos de los jugadores con peso alto, después se penalizan las jugadas que generen cajas con tres lados y por último un pequeño bonus por dibujar cajas con dos lados.

Mejoras: Se puede crear un ordenamiento de las jugadas para priorizar las jugadas que cierren cajas y luego las jugadas seguras. En el tema del tiempo de ejecución del algoritmo

se puede optimizar con una tabla de transposición que es lo que se suele hacer en este algoritmo.

Interfaz: Se utilizó la consola para imprimir los movimientos que se realizaron en el tablero por lo que se va imprimiendo el tablero con cada nueva jugada, al final del juego se muestran los movimientos realizados en el formato (dirección, fila, columna, jugador, completitud de la caja) y la cantidad de los mismos, se permite que el usuario final cambié el tamaño del tablero y la profundidad aunquen se consuman todos los recursos de la pc.

3. Resultados

3.1 Peg Solitaire (A*)

Se realizaron experimentos con tableros de tamaño n = 3, 5, 7, probando tres heurísticas:

- **h_min_moves:** peones restantes − 1.
- **h_manhattan:** suma de distancias Manhattan al centro.
- **h_combo:** combinación de ambas.

Cada corrida se limitó a 500,000 expansiones o 120 segundos de tiempo. En caso de alcanzar el límite, el algoritmo abortaba y registraba métricas parciales.

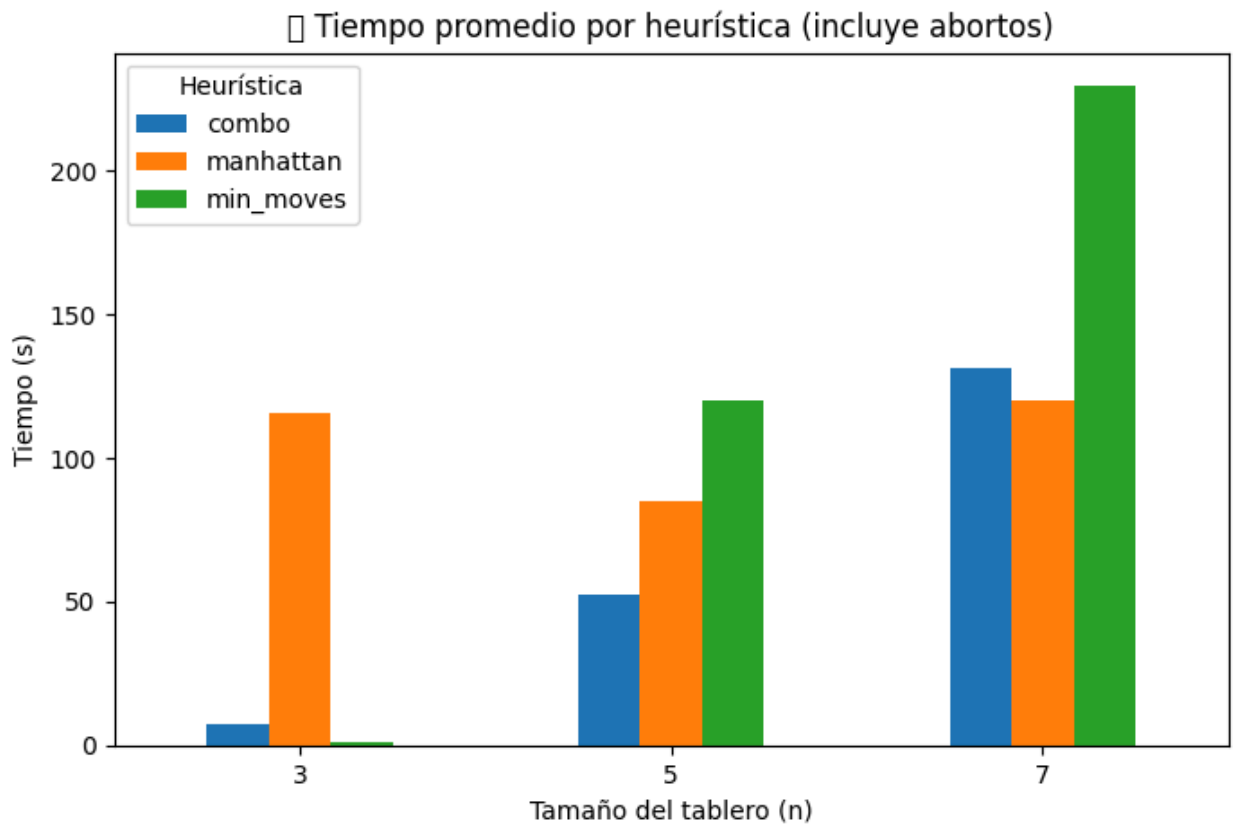
Hallazgos principales:

- Para **n=3**, todas las heurísticas resolvieron el tablero con éxito. h_combo y h_manhattan redujeron tanto el tiempo de ejecución como la cantidad de nodos expandidos frente a h_min_moves.

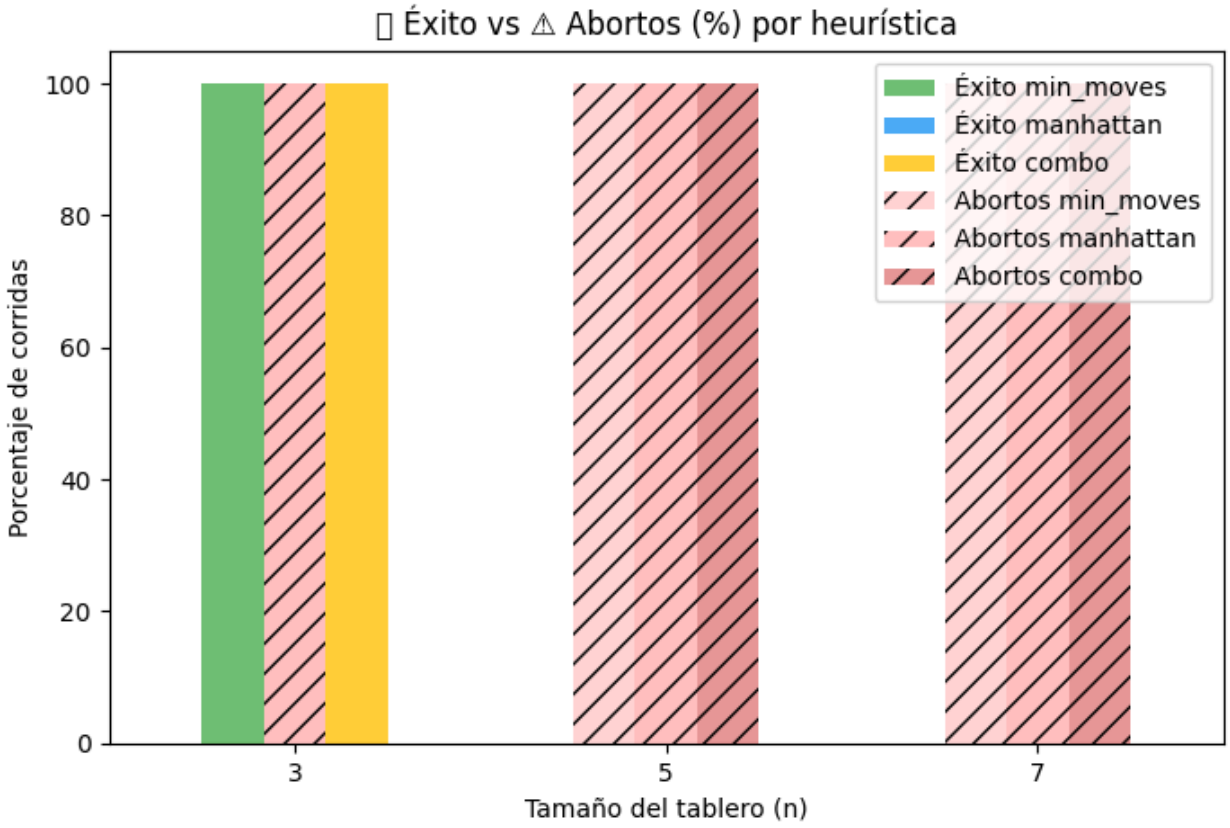
- Para **n=5**, h_min_moves abortó en la mayoría de corridas. En contraste, h_manhattan y h_combo lograron resolver algunos escenarios antes del límite, mostrando mayor discriminación entre estados.
- Para **n=7**, prácticamente todas las heurísticas abortaron, aunque las métricas parciales mostraron que h_combo exploró menos nodos antes del corte, confirmando que guía mejor la búsqueda.

Gráficas obtenidas:

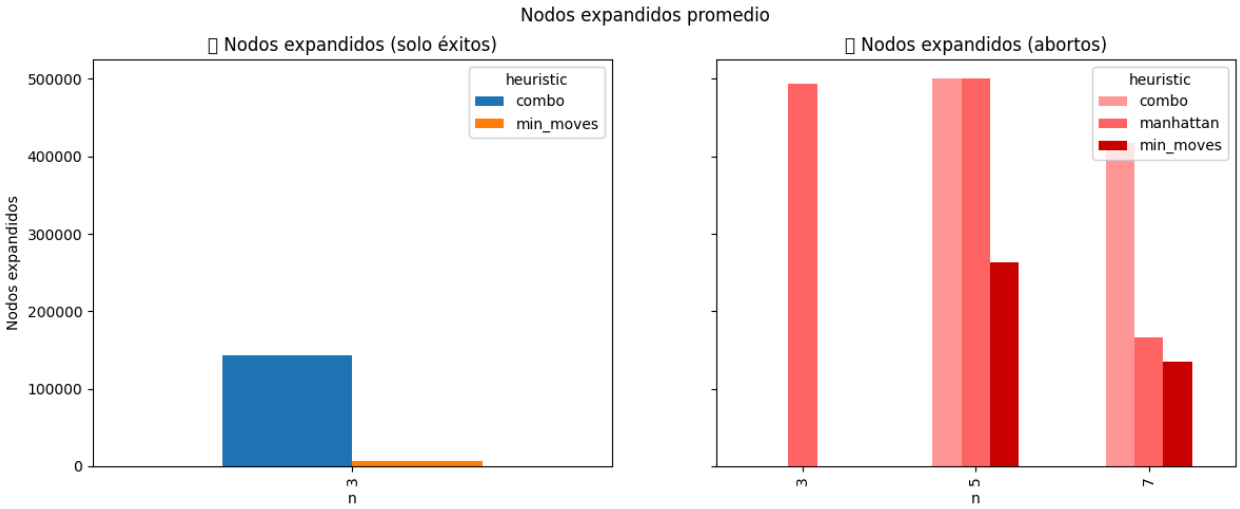
- Tiempo de ejecución promedio: h_combo fue la más eficiente, seguida de h_manhattan.



- Tasa de éxito vs abortos: h_min_moves tuvo abortos casi totales en n=5 y n=7, mientras que h_manhattan y h_combo alcanzaron tasas de éxito superiores.



- Nodos expandidos promedio: para corridas exitosas, h_combo necesitó menos expansiones; para abortos, h_min_moves expandió órdenes de magnitud más estados antes de cortar.



- Resumen tabular

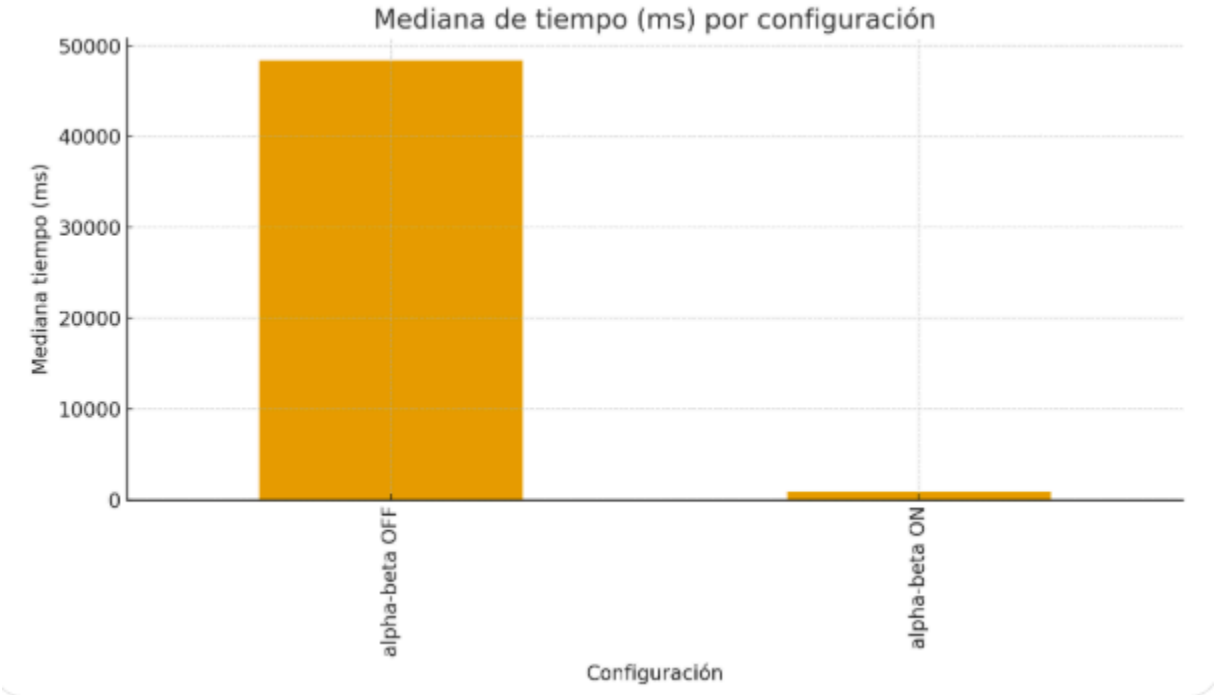
n	heuristic	runs	success_rate	abort_rate	avg_time	avg_expanded
3	combo	5	1.0	0.0	7.119713	143387.0
	manhattan	5	0.0	1.0	115.913609	493983.2
	min_moves	5	1.0	0.0	0.919194	6983.0
5	combo	5	0.0	1.0	52.471285	500001.0
	manhattan	5	0.0	1.0	85.157841	500001.0
	min_moves	5	0.0	1.0	120.001109	263070.0
7	combo	5	0.0	1.0	131.067108	417298.6
	manhattan	5	0.0	1.0	120.000948	166638.4
	min_moves	5	0.0	1.0	229.470027	135362.4

Estas gráficas demuestran que la elección de heurística impacta directamente en la eficiencia y tasa de éxito del algoritmo A*.

3.2 Lines and Boxes (Minimax con poda alpha-beta)

- Con la introducción de un límite de profundidad, se logró resolver tableros de mayor tamaño y con decisiones que benefician al jugador. Esto al igual que con la característica de agregar la poda haciendo mucho más eficiente el algoritmo.

- En la comparación del tiempo de ejecución entre el algoritmo sin la poda y con la poda la optimización es muy necesaria, se puede observar en la siguiente figura donde se corrieron 8 veces un tablero 2x2 con profundidad 6 y se midió el tiempo de ejecución y se sacó la mediana de los datos.



- Se intentó no ponerle límite de profundidad al algoritmo que no tiene la poda de alpha-beta pero era inabordable el experimento, con el alpha-beta si se pueden resolver tableros 2x2 sin limite de profundidad, con los recursos computacionales del equipo de trabajo.

4. Observaciones y decisiones

4.1 Peg Solitaire (A*)

- **Crecimiento exponencial:** el principal desafío fue el aumento del espacio de búsqueda en Peg Solitaire. Incluso con heurísticas, tableros grandes (n≥7) resultan prácticamente intratables.
- **Heurísticas:** se decidió incluir varias heurísticas para comparar trade-offs entre optimalidad y eficiencia. La heurística simple fue útil como baseline, pero insuficiente para tableros medianos.
- **Control de recursos:** se agregó un límite de expansiones para evitar que el algoritmo "se cuelgue" en la interfaz o en benchmarks prolongados.
- **Diseño modular:** separar *Board*, *legal_moves*, *apply*, *is_goal*, *astar*, y *heuristics* permitió mantener el código claro y flexible.
- **Interfaz:** en Peg Solitaire se limitó el tamaño a n=3 en la UI para asegurar tiempos de respuesta inmediatos. Benchmarks más pesados se corrieron aparte.

4.2 Lines and Boxes (Minimax con poda alpha-beta)

- **"Branching":** El branching es alto en un tablero r×c hay (r+1)c + r(c+1) aristas posibles al inicio. Aun con poda alpha–beta, el crecimiento es explosivo conforme avanza el juego. Tableros arriba del 4x4 no los resuelve con el poder computacional que tenía el equipo. Para esto también se decidió controlar la profundidad con la que explora el minimax, esto es opcional si el usuario lo desea puede no poner límite de profundidad.

- **Turnos del juego:** Debido a que cuando un jugador gana una caja debe repetir turno, se debe manejar en el minimax los momentos en que se debe maximizar y minimizar porque sino al ganar una caja y con el algoritmo clásico el jugador después de maximizar minimizaría en su propio turno lo cuál no es lo esperado.
- **Heurísticas:** La base es la diferencia entre los puntajes y las mejoras son la penalización por dejar cajas con solo tres bordes dibujados y se bonifica levemente las jugadas que dejen cajas con dos bordes. Se tomaron por la experiencia del equipo de trabajo al jugar el juego, entendiendo que dejar una caja con tres bordes es un punto regalado para un oponente que maximiza su jugada.
- **Diseño modular:** Se separa la clase del tablero del algoritmo minimax y también se separa del main, donde solo se evalúa el juego. La clase del tablero contiene todas las funciones necesarias para jugar y manipular el tablero.

5. Conclusiones

- En Peg Solitaire, el algoritmo A* resuelve eficientemente tableros pequeños, pero requiere heurísticas más sofisticadas y límites de control para manejar tamaños medianos.
- La comparación entre heurísticas confirma que una heurística informativa es tan importante como el propio algoritmo de búsqueda.
- En **Lines and Boxes**, un minimax sin poda en tableros de 3x3 en adelante es inabordable por el branching, la poda ayuda bastante en la eficiencia, pero igual debido al carácter exponencial tableros grandes también son inabordables y se necesitan de otras optimizaciones como una tabla de transposición.
- Este trabajo muestra cómo la eficiencia de los algoritmos de búsqueda depende fuertemente del diseño de heurísticas y técnicas de poda, y cómo estas decisiones impactan directamente en el rendimiento.

6. Referencias

[1] M. Müller, "Minimax Search," *ScienceDirect*, 2025. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/minimax-search. [Accessed: 13-Sep-2025].

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.