TEC | Tecnológico de Costa Rica

TC01: Búsquedas.

Diego Alonso Mora Montes - 2022104866

Omar Alexis Madrigal Rodriguez - 2020059280

Profesor: Kenneth Obando

Inteligencia Artificial

Escuela de Ingeniería en Computación

Instituto Tecnológico de Costa Rica

Cartago, Costa Rica

Video:

https://youtu.be/6WiXhu3pdrU

A*: Peg Solitaire

Implementación:

Mediante el uso del algoritmo A*, cuyo objetivo es encontrar el camino óptimo desde el estado inicial del tablero hasta la meta o el estado final, se resolvió el juego de Peg Solitaire mediante la siguiente implementación:

Tablero:

El tablero se implementó como un arreglo de Numpy 7x7 de estilo Ingles en donde:

- -1 = posición fuera del tablero
- 0 = posicion vacia
- 1 = posición con ficha

Generación de movimientos:

La función get_possible_moves(board) recorre todas las fichas del tablero y genera movimientos legales. Un movimiento es legal si:

1. Una ficha salta sobre otra ubicada a la izquierda, derecha, arriba o abajo de la ficha que se está evaluando.
2. La celda de destino está vacía.

Aplicación de movimientos:

La función apply_move(board, move) recibe un estado del tablero y un movimiento, y devuelve un nuevo tablero en el que la ficha inicial se movió, la ficha intermedia se eliminó y la ficha final se colocó en el destino.

Lista abierta y lista cerrada:

- La lista abierta se implementó como una cola de prioridad usando heapq, que siempre selecciona el nodo con menor f(n) para expandir.
- La lista cerrada se implementó como un diccionario que guarda el costo g más bajo alcanzado para cada tablero.

Algoritmo A*:

Cada estado en la búsqueda contiene:

- g: costo acumulado desde el inicio (número de movimientos).
- h: heurística (estimación de movimientos restantes).
- f = g + h: función de evaluación usada para ordenar la lista abierta.
- parent: referencia al nodo padre, lo que permite reconstruir el camino solución al llegar al objetivo.

Heurística Utilizada:

La definición de la heurística que se utilizo es la siguiente:

$$h(n) = \text{fichas restantes} - 1$$

Esto nos indica el numero minimo de movimientos necesarios para alcanzar el estado objetivo de una sola ficha en el medio del tablero.

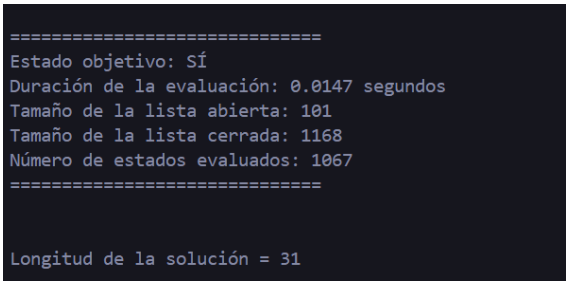
$$g(n) = g(n-1) + 1$$

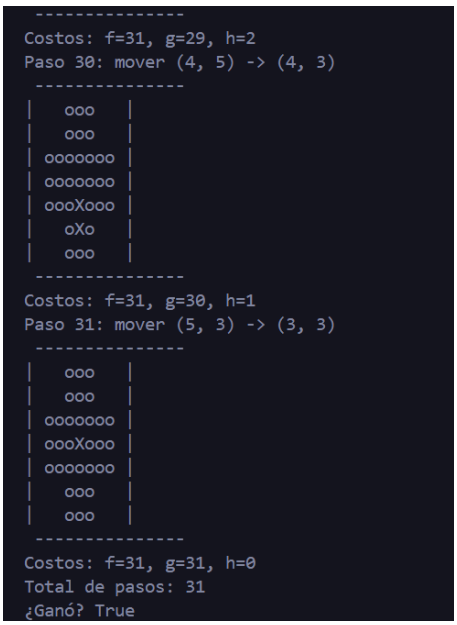
Esto nos indica cuantos estados tuvimos que atravesar para poder llegar al estado objetivo.

Resultados:

El sistema muestra:

1. Si se alcanzó el estado objetivo
2. La duración de la evaluación
3. El tamaño final de la lista abierta
4. El tamaño final de la lista cerrada
5. Número de estados evaluados
6. La longitud de la solución
7. Paso a paso hacia la solución:
    a. Cada movimiento aplicado en el formato (fila_1, columna_1) → (fila_2, columna_2).
    b. El tablero resultante tras cada movimiento.
    c. Los valores de f, g, h asociados a cada estado.
    d. El número total de movimientos y la verificación de si se alcanzó el objetivo.





Min-Max: Lines and Boxes o Timbiriche

Implementación:

El proyecto se estructuró de manera **modular**, siguiendo principios de separación de responsabilidades. La capa base es el motor del juego, responsable de representar el tablero, validar movimientos y calcular puntajes. Sobre esta capa se implementaron distintos tipos de jugadores:

- Jugador aleatorio: actúa como referencia, eligiendo movimientos sin ninguna estrategia.

- Jugador Minimax: aplica un algoritmo de búsqueda adversaria con profundidad limitada, utilizando una función heurística para evaluar posiciones intermedias.

El algoritmo Minimax es un enfoque clásico en inteligencia artificial para juegos de dos jugadores con información perfecta. La idea consiste en recorrer el árbol de estados posibles hasta cierta profundidad y asignar un valor a cada posición. Los nodos de tipo MAX buscan maximizar la ganancia, mientras que los de tipo MIN intentan minimizarla, simulando el comportamiento de un oponente racional.

En esta implementación, además de seleccionar la jugada óptima, se registraron métricas como el número de **nodos explorados** y los **tiempos de ejecución**, lo que permite analizar empíricamente el rendimiento y la escalabilidad del algoritmo.

Evaluación heurística:

Explorar exhaustivamente todos los movimientos es computacionalmente prohibitivo incluso en tableros pequeños, debido a la explosión combinatoria inherente a este tipo de juegos. Por ello, la calidad del algoritmo depende fuertemente de la heurística de evaluación.

La función heurística diseñada integra múltiples factores:

1. Diferencia de puntajes: asigna un valor proporcional a la ventaja en el marcador.

2. Cuadros incompletos: se analizan las casillas con 2 o 3 lados ocupados. Si un cuadro tiene 3 lados, la decisión de cerrarlo o evitar entregarlo se penaliza/recompensa fuertemente.

3. Movimientos disponibles: se premian posiciones con mayor flexibilidad, ya que reducen el riesgo de quedar forzado.

4. Control del turno: se favorece mantener la iniciativa y encadenar jugadas, lo cual es estratégico en juegos como Dots and Boxes.

La combinación ponderada de estos factores logró un equilibrio entre un estilo ofensivo (maximizar puntos) y defensivo (evitar entregar cuadros), produciendo un comportamiento más cercano a un jugador humano estratégico.

Función Implementada:

```python
def evaluate_position(self):
    """Evalúa el estado del tablero con una heurística."""
    if self.is_game_over():
        return (self.scores[0] - self.scores[1]) * 100

    # Diferencia de puntos
    score = (self.scores[0] - self.scores[1]) * 50

    # Cuadros casi completados
    for r in range(self.size):
        for c in range(self.size):
            if not self.completed_boxes[r][c]:
                sides = self.count_box_sides(r, c)
                if sides == 3:
                    score += 20 if self.current_player == 0 else -20
                elif sides == 2:
                    score += 6 if self.current_player == 0 else -6

    # Movimientos disponibles
    score += len(self.get_possible_moves()) * 2

    # Control de turno
    if self.move_history and self.move_history[-1][3] == self.current_player:
        score += 3 if self.current_player == 0 else -3
    return score
```

Resultados:

Se realizaron pruebas en tableros de 3x3 enfrentando al jugador **Minimax** contra el jugador **aleatorio**.

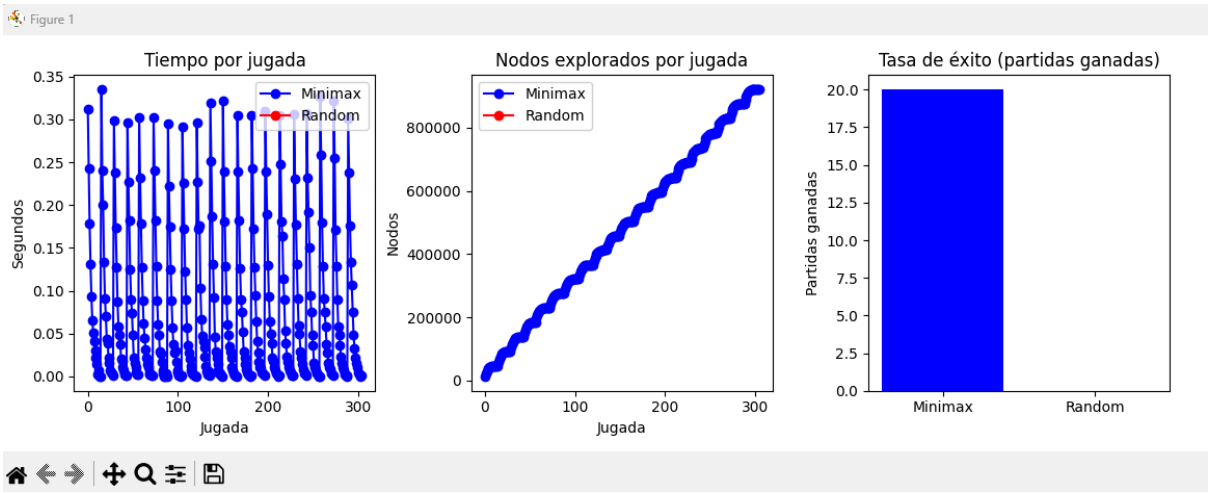
- **Tasa de victorias:** Minimax superó consistentemente al jugador aleatorio, mostrando clara ventaja.

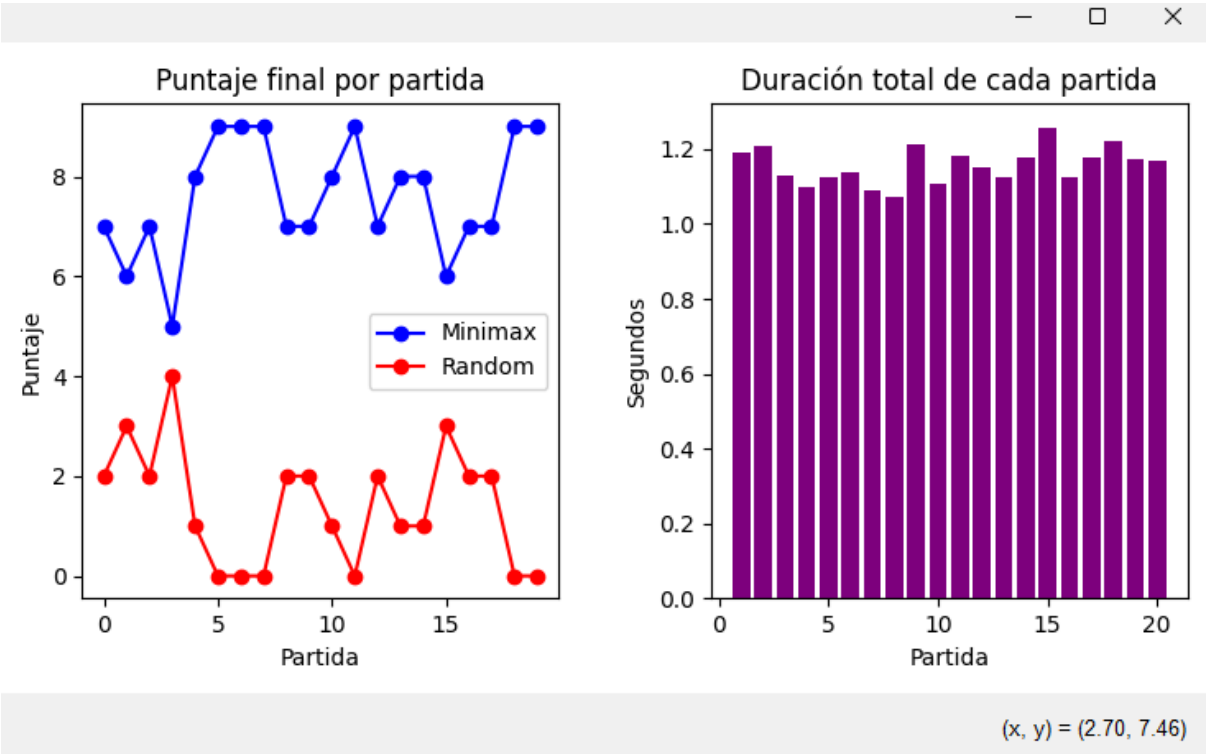
- **Influencia de la profundidad:** aumentar la profundidad mejoró la calidad de las decisiones, pero el tiempo de cómputo creció de manera no lineal debido al incremento de nodos explorados. Esto confirma la naturaleza exponencial del problema.

- **Métricas recolectadas:** se graficaron estadísticas de tiempos por jugada, nodos explorados, duración de las partidas y puntajes finales, ofreciendo una visión cuantitativa del desempeño.

Este comportamiento confirma la limitación natural del algoritmo frente a la explosión combinatoria del juego.En conclusión, los resultados validan que un Minimax con heurística adaptada es suficiente para jugar de forma competitiva en tableros pequeños, mostrando una clara superioridad frente a estrategias aleatorias.

Para complementar este análisis, se incluirán dos imágenes con los gráficos obtenidos al ejecutar Minimax con una profundidad de 3 en un tablero de 3x3 durante 20 partidas. Estos gráficos muestran la evolución de métricas clave como tiempos de ejecución, nodos explorados y resultados de las partidas, ofreciendo una representación visual del rendimiento del algoritmo.




Puntaje final por partida
Puntaje
Partida
Minimax
Random
0
2
4
6
8
0
5
10
15
Duración total de cada partida
Segundos
Partida
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
5
10
15
20
(x, y) = (2.70, 7.46)