TEC | Tecnológico de Costa Rica

Inteligencia Artificial - IC6200

TC1 - Algoritmos de Búsqueda - A* y MinMax

Profesor:

Kenneth Obando Rodríguez

Integrantes:

Leonardo Céspedes Tenorio - 2022080602

Kevin Chang Chang - 2022039050

Fecha de Entrega: 25 de Marzo

I Semestre 2025

A* - Peg Solitaire

Implementación:

Para la implementación del algoritmo de Búsqueda A* En el juego *Peg Solitaire* seguimos los siguientes pasos:

1. Definimos las variables globales como los movimientos posibles de una pieza guardadas en una lista de tuplas, la posición objetivo deseado de la última pieza del juego, una matriz representando el estado inicial del tablero y otra con el estado objetivo del tablero.
2. Programamos la lógica del juego.
    a. getMoves(): Analiza todos los posibles movimientos en un estado dado del tablero.
    b. applyMove(): Aplica un movimiento específico a un tablero dado y retorna el tablero resultante.
    c. isGoal(): Válida si el tablero actual es igual al tablero objetivo para saber si ya se llegó al objetivo.
3. Definimos la función heurística que predice los turnos restantes para llegar al estado objetivo.
4. Finalmente aplicamos el algoritmo de A*. Abrimos listas de nodos cerrados y abiertos e iteramos sobre ellas encontrando todos los movimientos posibles desde un estado dado y calculando el f() de cada posición. Escogemos el valor de f más óptimo y desarrollamos los posibles movimientos de ese estado. El proceso se repite hasta encontrar el objetivo o hasta haber navegado todos los estados sin haber encontrado la solución. (f(n) = g(n) + h(n)).
5. Se guardan los pasos a seguir, información sobre el tiempo de ejecución y el número de nodos abiertos y se presentan los datos.

Heurística:

Para la heurística decidimos tratar de predecir el número de turnos restantes para llegar al estado objetivo. Nuestro valor de g(n) representa la cantidad de turnos que se han jugado hasta el momento por lo que el estimado que planeamos utilizar en la heurística nos parece la más óptima y apropiada para este caso en específico.

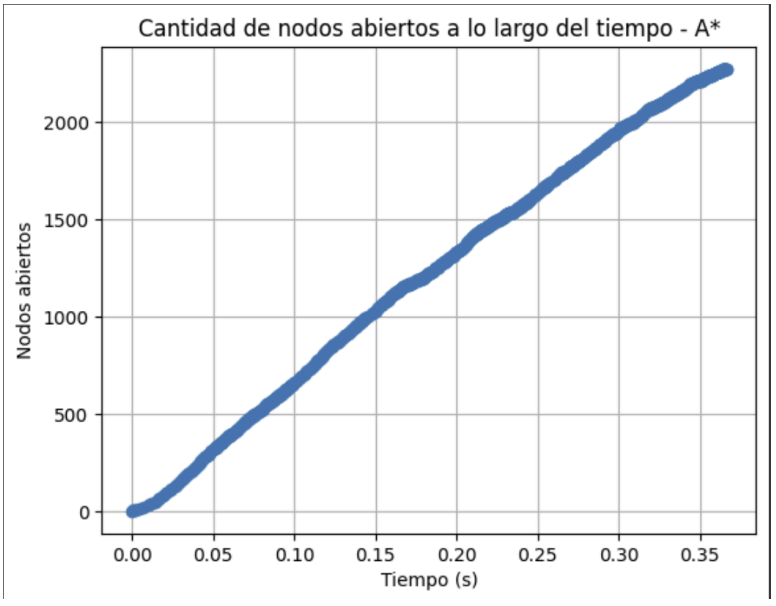
Para nuestra heurística, calculamos la distancia total de cada pieza en el tablero con la posición objetivo de la última pieza al ganar el juego (En este caso el centro del tablero). Calculamos utilizando la distancia de Manhattan para todas las piezas en un dado estado y calculamos la suma de todas estas distancias. Finalmente contamos el número de movimientos posibles en el estado dado y lo restamos a la suma anterior como una penalización. Esto nos da el estimado de turnos pendientes para llegar al estado objetivo.

```python
def heuristica(board):
    positions = np.argwhere(board == 1)

    if positions.size == 0:
        return float('inf')

    distances = np.sum(np.abs(positions - TARGET), axis=1)
    moves_penalty = len(getMoves(board))  # Penalizar estados con menos movimientos
    return np.sum(distances) - moves_penalty
```

Resultados Obtenidos:

La función sirve correctamente y analiza apropiadamente cada nodo y estado para poder hacer correctamente la búsqueda de la solución más óptima.

Lo que se ha observado es que para solucionar el tablero presentado en el enunciado con la última pieza en el medio del tablero como objetivo, el algoritmo ha completado esto de forma rápida en 25 pasos. El siguiente gráfico muestra una relación lineal entre el tiempo y el número de nodos abiertos lo cual nos muestra que se encuentra la solución de forma directa e y casi inmediata (No explora mucho y encuentra la solución rápido).



El problema que hemos visto es que con otros estados iniciales y objetivos, el programa ha durado más tiempo encontrando la respuesta, especialmente porque no todas las combinaciones de estado iniciales y objetivos tienen una solución por lo que el programa termina haciendo una búsqueda completa de todos los posibles movimientos y estados del tablero.

Concluimos que nuestro algoritmo es capaz de solucionar un problema que se sepa que tiene una solución finita. Tuvimos suerte al utilizar un conjunto de estados que tiene una solución que nuestro algoritmo encuentra rápido, pero con otros conjuntos el algoritmos puede llegar a durar más tiempo ya que tiene que explorar más estados (nodos) y llega a durar mucho más si dicho conjunto no tiene una solución del todo.

Dots and Boxes: MiniMax

Implementación:

Para la implementación del algoritmo de búsqueda Minimax en el juego *Dots and Boxes* seguimos los siguientes pasos:

1. Definimos la estructura que se utilizará para manejar los estados de juego junto a toda su información relacionada. Se decidió utilizar un diccionario con los siguientes datos:
    a. *horizontalLines*: Una matriz que representa cada una de las líneas horizontales existentes en el tablero. Cada valor de la matriz empieza con un valor de 0, esto se irá cambiando cada vez que un jugador realice una acción y se asignará un 1 o 2 dependiendo de cuál jugador "dibuja" esa línea.
    b. *verticalLines:* Una matriz que representa cada una de las líneas verticales existentes en el tablero. Cada valor de la matriz empieza con un valor de 0, esto se irá cambiando cada vez que un jugador realice una acción y se asignará un 1 o 2 dependiendo de cuál jugador "dibuja" esa línea.
    c. *boxes:* Una matriz que representa cada una de las cajas presentes en el tablero, posee el mismo funcionamiento que las líneas.
    d. *scores:* Se utiliza un diccionario que como llave posee el jugador (0 y 1) y como valor la cantidad de cajas que posee el jugador en cada estado del juego.

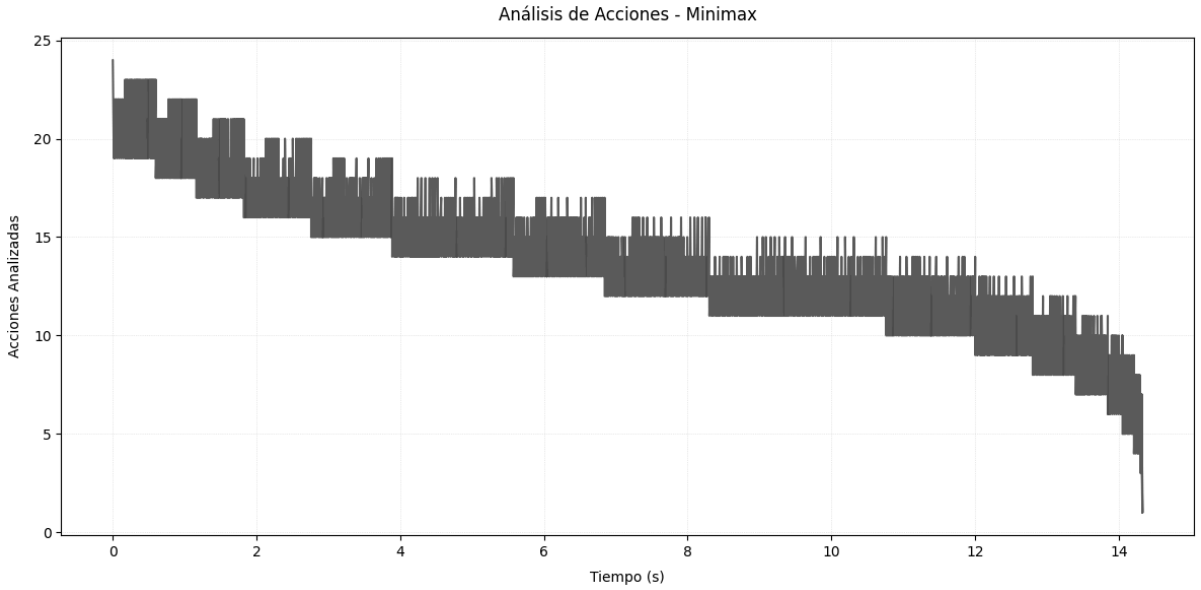
e. *maximizingPlayer:* Bool utilizado para llevar registrado quién es el jugador actual del estado.
f. *totalLines:* Corresponde a un integer que almacena la cantidad total de líneas en el tablero.
g. *linesDrawn:* Corresponde a un integer que almacena la cantidad total de líneas que han sido dibujadas. Este elemento funciona junto a *totalLines* para determinar si el juego ya acabó.
h. *size:* Almacena las dimensiones del tablero en términos de cajas. Por ejemplo, si *size* tiene un valor de 3, significa que el tablero es de 3x3 cajas.

2. Se define la lógica del juego, lo cual incluye las siguientes funciones:
   a. *isTerminal(state):* Esta función compara las líneas totales con las líneas dibujadas para determinar si el juego ya acabó.
   b. *utility(state, player):* Esta función permite calcular un valor de utilidad para un estado de juego. Esto se realiza restando la puntuación del *player* menos la puntuación del oponente.
   c. *getActions(state):* Retorna todos los movimientos posibles, lo cual equivale a todas las líneas verticales y horizontales que no han sido asignadas a ningún jugador.
   d. *applyAction(state, action, player=None):* Función utilizada para aplicar un movimiento sobre el estado actual. Luego de cambiar el valor en la matriz correspondiente, se revisa si se completó alguna caja tras dibujar la línea. Retorna el nuevo estado y un bool que representa si se completó o no una caja.
   e. *checkBox(state, i, j):* Función utilizada tras aplicar un movimiento para verificar si se completó una caja. Se verifican aquellas cajas en las que una de las aristas corresponde a la línea dibujada
3. Se crea el la función minimax, la cual será la principal función del algoritmo. Primeramente se valida si la profundidad es 0 o el juego ya terminó, lo cual corresponde a los casos base de la función. Seguidamente, dependiendo del jugador actual se define una variable *value*, la cual servirá para posteriormente determinar cuál es el mejor movimiento. Se obtienen todas las acciones del tablero actual e itera sobre estas, aplicando cada una de estas para ir viendo cuál posee mejor valor y también ir haciendo *pruning* con alpha y beta y así evitar recorrer secciones innecesarias del árbol que se irá formando. Finalmente, la función retorna el valor y la mejor acción posible.

4. Finalmente, se implementa una función para obtener el mejor movimiento posible para el jugador actual, la cual utiliza el método minimax para obtenerlo.

Resultados Obtenidos:

El algoritmo funciona correctamente y explora todos los movimientos posibles en el estado actual del juego. A continuación se muestra un gráfico de la cantidad de jugadas analizadas a lo largo de la ejecución del programa:



Como se puede observar, la cantidad de movimientos analizados va decreciendo, lo cual tiene sentido en el contexto de minimax, pues conforme se va avanzando en el juego hay menos espacios disponibles para hacer una línea.

Algo curioso es que no siempre se elige la mejor opción posible, esto probablemente es debido a la profundidad predefinida para mejorar el performance. Al inicio se estableció el parámetro como 4, pero se daban más casos de estos donde la mejor opción no era elegida. Por eso, se decidió cambiar a 6 y con esto se redujo la cantidad de 'errores' cometidos sin aumentar considerablemente el tiempo de ejecución.

En conclusión, el algoritmo funciona muy bien y se adapta correctamente a la estructura elegida para la representación de los estados. Es bastante eficiente y si se desea reducir la cantidad de 'errores' tomados durante las jugadas podría aumentarse el valor de la profundidad.

Enlace a cuaderno de Jupyter:

https://colab.research.google.com/drive/1EJfFkT86bpejtiW630bp4FQTDM26280r?usp=sharing