TPO
Caballo De Ajedrez

Materia: Programación III

Profesor: Bianchimano Omar Nestor Cristian

Alumnos: Federico Bavio, Jordi Castro López, Franco Miño, Nicolás Vega

Parte 1: Backtracking básico.

Implementación utilizando la técnica backtracking para realizar el recorrido del caballo de ajedrez.

Descripción del algoritmo:

Datos de entrada: matriz NxN (representa el tablero de ajedrez), posX, posY, contador de movimientos.

Caso base: contador de movimientos igual al área de la matriz NxN, esto significa que el contador es igual al área de la matriz, dándonos que se recorrieron todas las casillas del tablero.

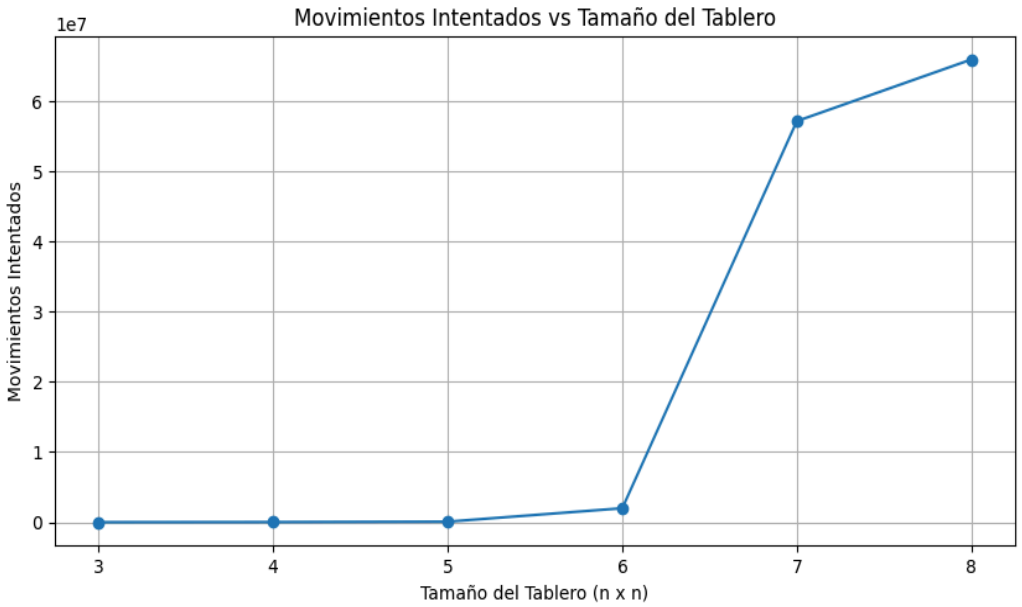
PSEUDOCÓDIGO:

```
funcion esValido(x, y, tablero):
    size = length(tablero)
    retornar 0 <= x < size and 0 <= y < size and tablero[x][y] == -1

funcion resolver_caballo_tour(tablero, x, y, contadorMov):
    size = length(tablero)
    si contadorMov == size * size: ← Caso Base
        retornar True

    movimientosPermitidos = {
        (2,1), (1,2), (-1, 2), (-2, 1),
        (-2,-1),(-1,-2),(1,-2),(2,-1)
    }

    para dX, dY en movimientosPermitidos:
        nuevo_x, nuevo_y = x + dX, y + dY
        si esValido(nuevo_x, nuevo_y, tablero):
            tablero[nuevo_x][nuevo_y] = contadorMov
            si resolver_caballo_tour(tablero, nuevo_x, nuevo_y, contadorMov+1):
                retornar True

            tablero[nuevo_x][nuevo_y] = -1 ← Backtracking, se vuelve para atras

    retornar False
```

Teniendo en cuenta un tablero de 5x5, sacamos la conclusión de que en posiciones iniciales de [x][y] %2!=0, no existe solución. Para las posiciones en los bordes que no tienen solución, realizan una cantidad de movimientos de 1.829.420, para las posiciones de adentro de la matriz, realizan 1.028.892.

Explicación del algoritmo: Este realiza un recorrido probando todas las combinaciones posibles con los movimientos permitidos (8 movimientos permitidos), borra el movimiento si no es posible colocar el caballo en ese lugar (backtracking).
Complejidad del algoritmo (O(8^{n^2})) en el peor caso.

Gráficos y resultados para diferentes tableros:



Vemos que para el caso de tableros 3x3 y 4x4 no existen soluciones en esos tamaños, la cantidad de movimientos es aproximadamente cero a comparación de los tamaños de 7x7 y 8x8 donde la cantidad de movimientos crece fuertemente.

La cantidad de operaciones del algoritmo crecen de una manera exponencial con respecto al cuadrado del tamaño del problema (n).
Tableros de n x n
n = 5: O(8^25) = 3.78e+22
n = 6: O(8^36) = 3.25e+32
n = 7: O(8^49) = 1.78e+44
n = 8: O(8^64) = 6.28e+57

Resultados de pruebas experimentales de cantidad de movimientos hasta encontrar una solución en función del tamaño del tablero (los movimientos fueron contados con una variable global puesta en la función recursiva la cual contaba la cantidad de movimientos que realiza el algoritmo, empezando en la posición (0,0)):

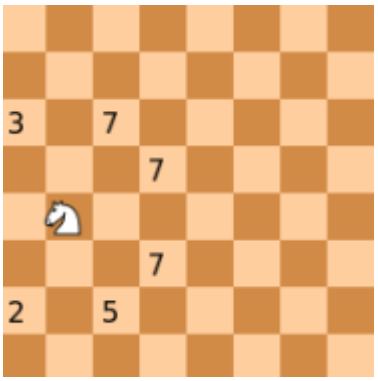
Tamaño de tablero	Cantidad de movimientos	Tiempo de ejecución
5x5	9217	0.028126717
6x6	248168	0.800505638
7x7	7359630	24.4580946
8x8	8250732	28.16437364

En pruebas experimentales de tableros 8x8 con diferentes posiciones iniciales el algoritmo es intratable ya que el árbol de recursión toma una profundidad muy densa en la búsqueda de alguna solución, por lo tanto los tiempos de ejecución crecen muchísimo.

Dado que los resultados son casi astronómicos, nos da indicios de que resolver un tablero de tamaño un poco más grande es inviable mediante la fuerza bruta, por lo tanto debemos aplicar técnicas de optimización como branch and bound.

Parte 2: Optimización con Branch & Bound (B&B).

Para realizar la optimización con Branch and bound debemos considerar la regla de Warnsdorff la cual se implementa como heurística para encontrar el recorrido óptimo, el caballo se mueve de manera que siempre avance hacia la casilla desde la que tendrá menos movimientos hacia adelante. En el algoritmo, este método se realiza tomando la posición en la cual se encuentra el caballo en ese momento y viendo los posibles movimientos futuros que tiene cada vecino a su alrededor, esto nos dará una lista que contendrá el número de futuros movimientos desde ese vecino.



Siempre vamos a elegir la que tenga menor cantidad de movimientos futuros; en este caso elegimos el 2 y posicionamos al caballo en esa casilla.

Explicación B&B en el algoritmo

Considerando la optimización de B&B, utilizando como heurística la regla de Warnsdorff, para el algoritmo. La complejidad se aproxima a O(N^2) porque al tomar el vecino con menor cantidad de movimientos próximos posibles, descarta las ramas que no conduzcan a una solución óptima (“poda implícita”)

Tomamos como **cota** a la mínima cantidad de movimientos futuros que tienen los vecinos desde esa casilla actual en la que se encuentra el caballo. Si el caballo se encuentra en una casilla donde los vecinos tienen pocas opciones de movimiento, puede ser más prometedor explorar esa rama, ya que tiene menos riesgos de quedar atrapado sin movimientos.
En este caso la poda por la cota inferior sería que si una rama requiere más movimientos que tu cota inferior actual para completarse, entonces podemos descartarla.
Este enfoque, especialmente con el análisis de los vecinos y el uso de cota inferior, ayuda a evitar ramas que probablemente no lleven a una solución óptima y, en cambio, se enfoca en soluciones más prometedoras según los movimientos del caballo en cada paso.

Se considera de importancia destacar que, al usar una heurística de Warnsdorff se está priorizando siempre el menor de los vecinos en términos de cantidad de movimientos posibles. Este número es lo que define nuestra Cota. Por lo que no resulta aplicable el uso de Cota Superior o Cota Inferior, sino que nos basamos en un único número para realizar

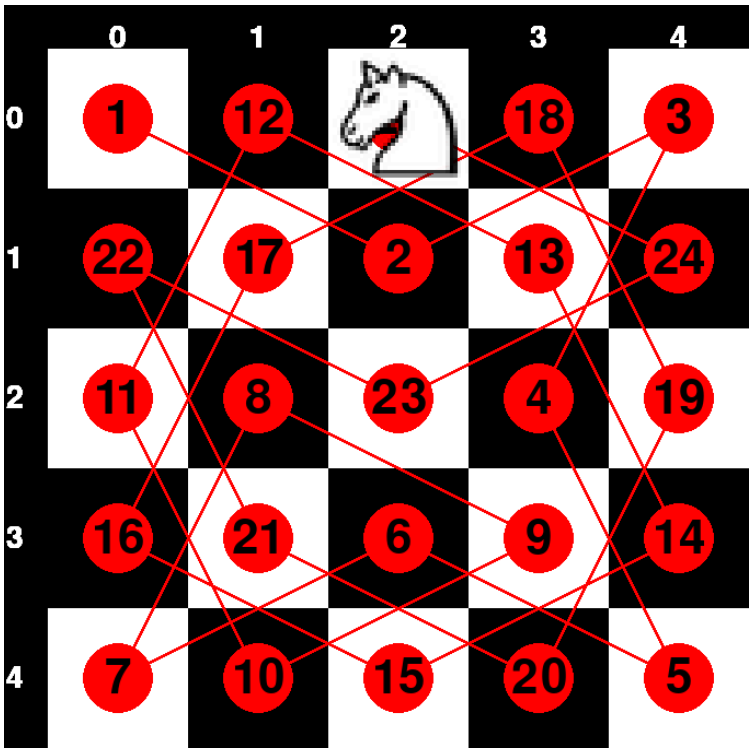
comparaciones. De todas maneras, a fines teóricos, tengamos en cuenta que nuestra Cota Superior podría ser definida como el número restante de cuadrantes del tablero que nos falta recorrer.

Empezando desde (0,0):

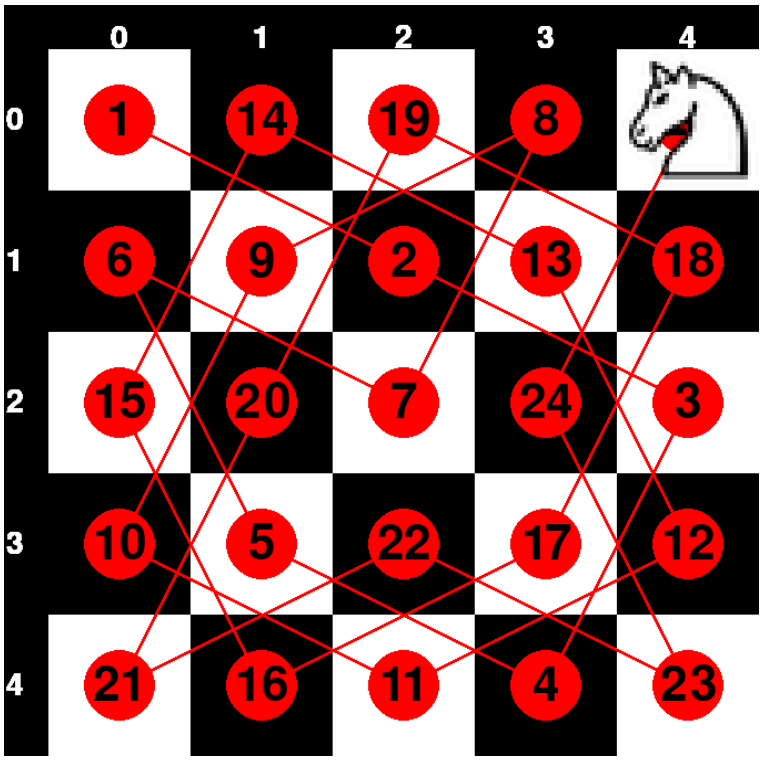
Tamaño de tablero	**Cantidad de movimientos**	**Tiempo de ejecución**
5x5	24	0.00000000000001
6x6	35	0.0010
7x7	48	0.0015
8x8	63	0.0010
10x10	99	0.0010

Parte 3: Comparación experimental.

Caso experimental con tablero 5x5 probado con Branch And Bound:

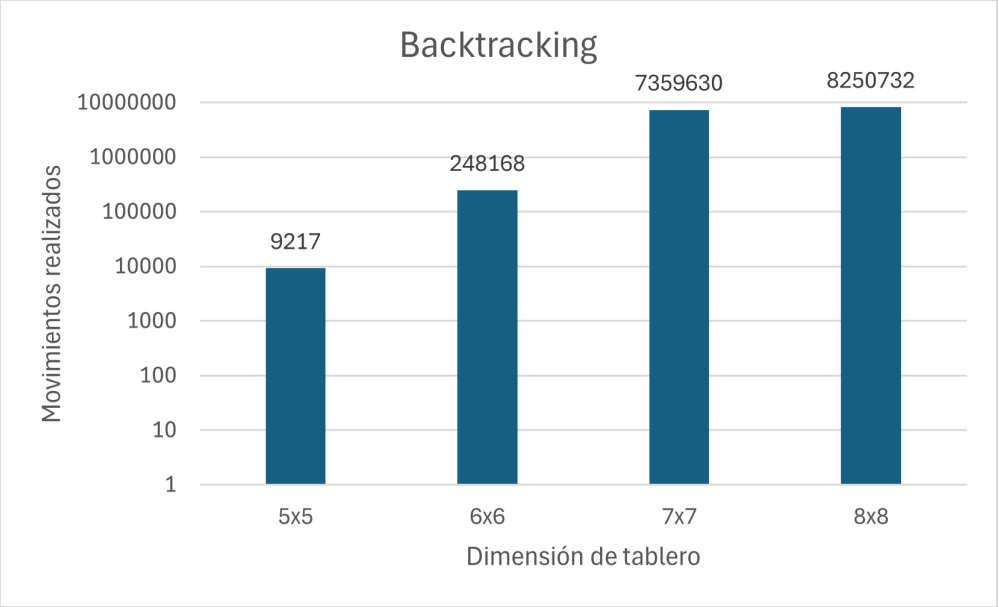


Caso experimental con el mismo tablero con backtracking básico:

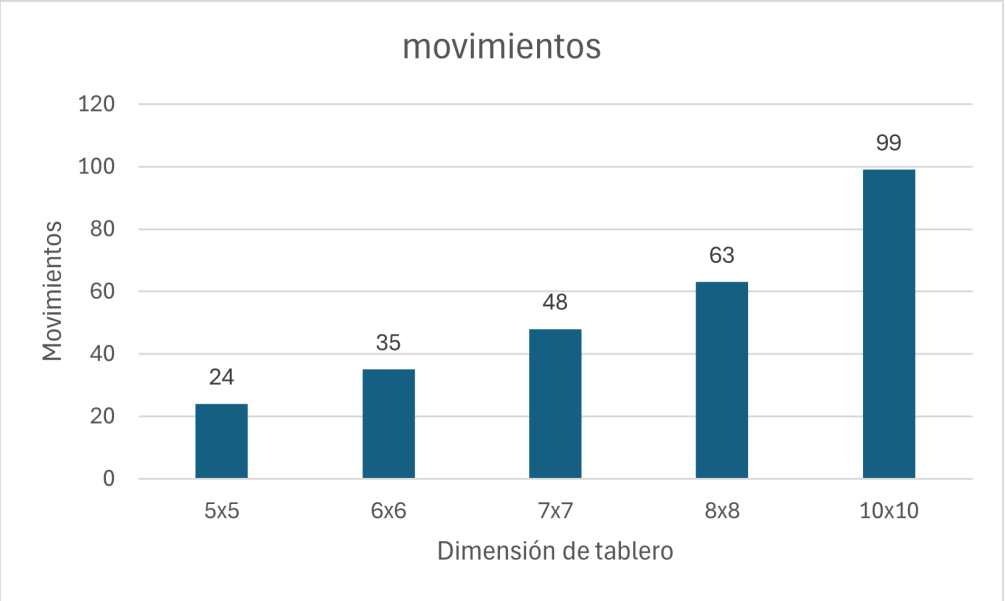


La principal diferencia observada son los diferentes caminos hamiltonianos resultantes en los dos tableros de misma dimensión.

Resultados backtracking básico



Resultados branch and bound



Podemos observar que la cantidad de movimientos se reduce exponencialmente cuando se utiliza branch and bound en lugar de backtracking básico, esto afecta considerablemente el árbol de recursión el cual se hace demasiado profundo en tableros de 10x10 utilizando backtracking básico.

Comparación experimental con diferentes podas

El algoritmo de búsqueda branch and bound sigue una exploración recursiva de las posibles rutas del caballo, y en cada paso intenta avanzar hacia las casillas con menos opciones de movimiento posibles (según la heurística de Warnsdorff). Al limitar el número de opciones que el algoritmo puede explorar, es decir, pasar de 8 vecinos a 2 vecinos con la menor cantidad de movimientos posibles, se hace una exploración más dirigida y menos exhaustiva, lo que puede dar lugar a soluciones más rápidas en algunos casos.

Impacto del slice [:2]:
Experimentando con diferentes slices desde 2,3,4,5,6,7,8 el mejor slice para reducir la cantidad de nodos prometedores fue 2. Con respecto a un slice de 1 existen casos particulares en el cual el algoritmo no encontraba soluciones, en cambio, con un slice de 2 si las encontraba.

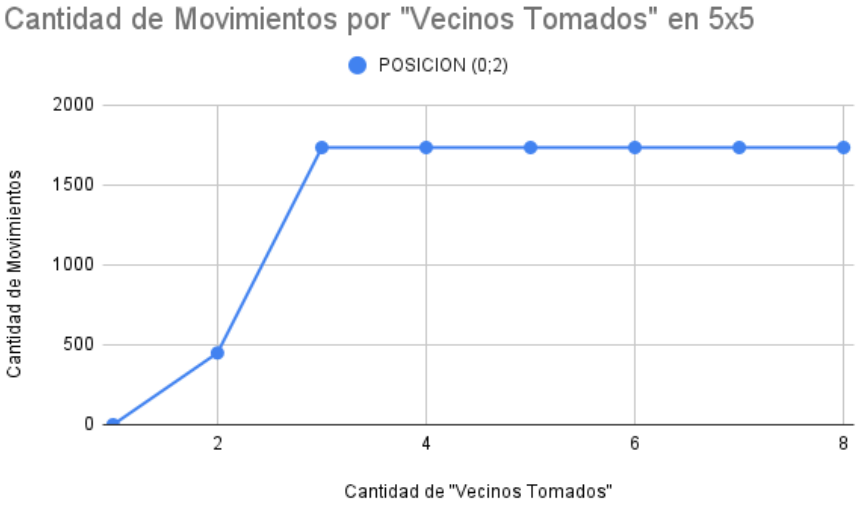
Reducción de la exploración: El algoritmo ahora solo evalúa las dos opciones más prometedoras en lugar de todas las disponibles. Esto puede llevar a un camino que, aunque más restringido, sea más eficiente y termine encontrando una solución en menos pasos.

Menos retrocesos: Dado que las opciones de movimiento son limitadas, es posible que el algoritmo realice menos retrocesos (backtracking), ya que se evalúan menos caminos y se evita pasar por caminos que resultan en estancamiento.

Posibilidad de poda más temprana: Al considerar solo los dos vecinos más prometedores, el árbol de búsqueda es podado más rápido, lo que significa que se descartan más rápidamente las rutas sin solución.

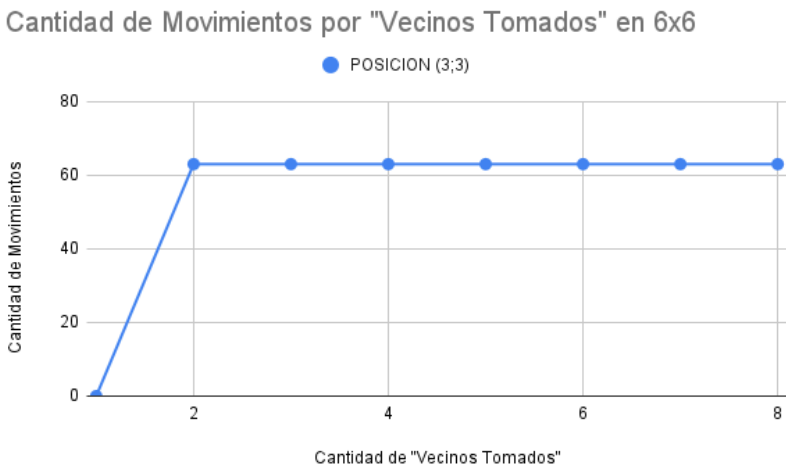
Tableros de 5x5

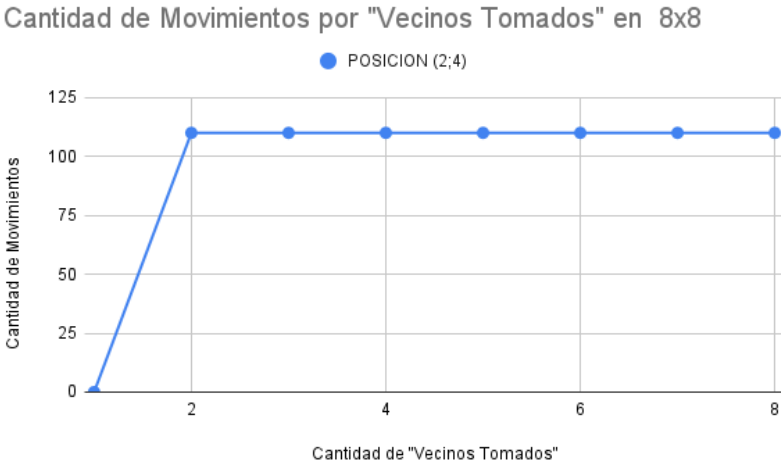
En este tablero podemos ver la mejora al tomar 2 vecinos en un **punto conflictivo**
Punto Conflictivo: Es un punto que al tomar el primer vecino, no daria solucion



Tableros de 6x6 y 8x8

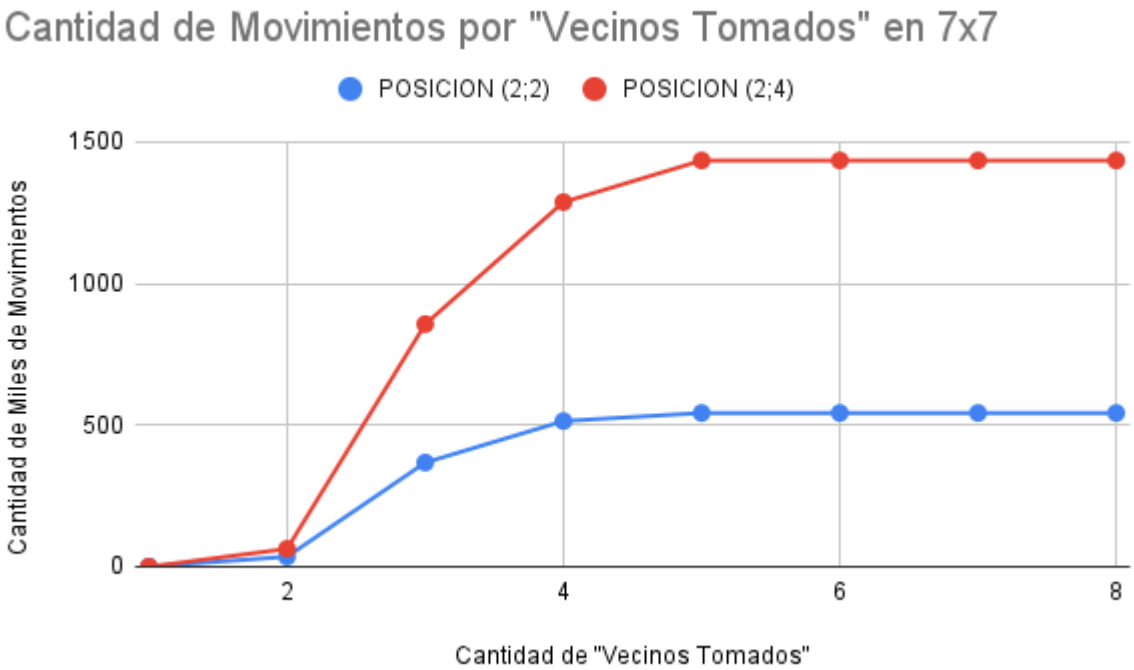
Estos tableros son iguales y la gran mayoría de los tableros pares (N par) se comportan igual

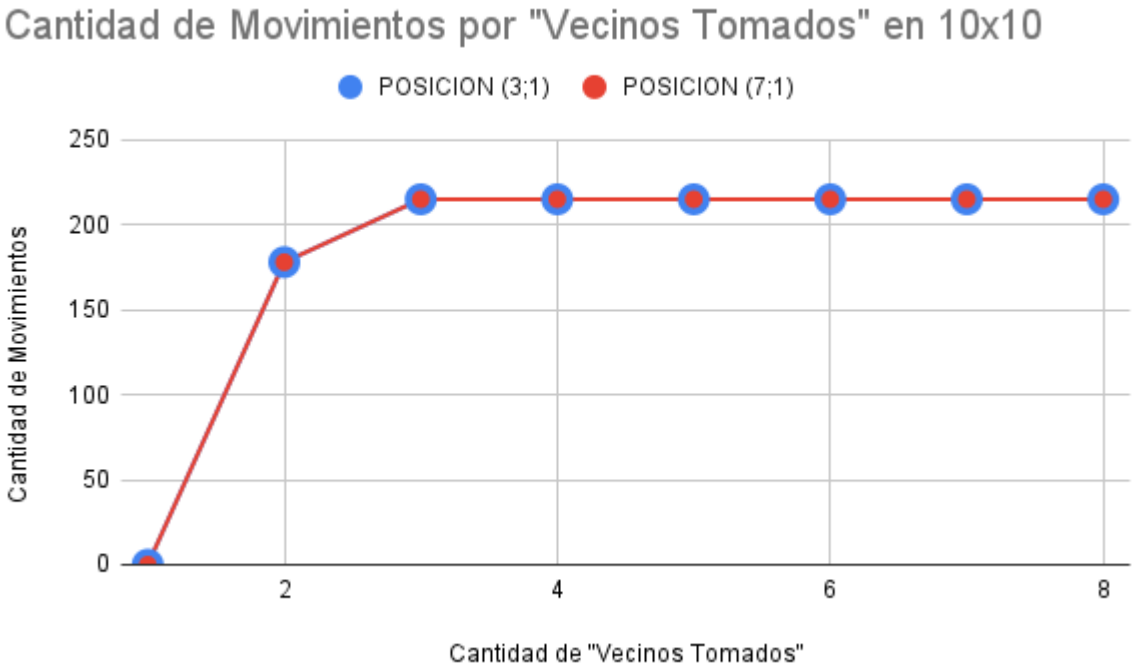




Tableros de 7x7 y 10x10

Estos tableros presentan más de un punto conflictivo





El tablero de 10x10 al ser par, se comporta igual sin importar qué cantidad de vecinos se tome

Tablero 9x9: Este tablero no presenta puntos conflictivos por lo que no fue graficado

Reflexión

La técnica de Backtracking sirve para llegar a una primera solución de una problemática, ya que utiliza fuerza bruta para probar combinaciones de una secuencia de acciones, caminos o posiciones que pueden conducir a la solución. Esta técnica consume muchos recursos lógicos (memoria) lo cual la hace ineficiente, en cambio, B&B busca los nodos vivos mediante la técnica de “ramificación”, utilizando una heurística, para reducir la cantidad de posibles acciones, caminos o posiciones a intentar. Además realiza una “poda” de los nodos que al evaluarlos no llegan a la solución, reduciendo así la profundidad del árbol y por consiguiente su complejidad temporal.