Informe tablero de ajedrez

Grupo 07

**Integrantes**

Huamán, Cecilia

**Materia** Programación III

**Profesor** Omar Bianchimano

# Índice

[**Indice 2**](#_hvirpxxjfdid)

[**Introducción 3**](#_oed8mj1eroae)

[**Objetivo 3**](#_74g1pk2t5bcl)

[**Backtracking 3**](#_qob5t1g0cj1t)

[¿Cómo funciona? 3](#_r8sapnddv4h7)

[Puntos clave 4](#_7difv99vv4tt)

[Pseudocodigo base 4](#_totdsehoqmfy)

[Ilustración a modo de ejemplo 5](#_h00c41fzj8pj)

[Complejidad Temporal 5](#_yrw6x9auzzi2)

[**Branch and bound 6**](#_v8j464n74u54)

[¿Cómo funciona? 6](#_uf6zxhwrtgv2)

[Ramificación 7](#_o3w0rl7e7ml0)

[Poda 7](#_gt648os7mew2)

[Heurísticas 7](#_hveoq9ttx130)

[Otras Opciones de Poda 8](#_hqe45slyvhan)

[Ventajas clave 8](#_1g46kykvfmo5)

[Aplicaciones comunes 9](#_akrzo8kdt3i1)

[Pseudocódigo base 9](#_ccv3eu7658lo)

[Ilustración a modo de ejemplo 10](#_6sdinmmswt65)

[Complejidad Temporal 10](#_dx31drqxd16f)

[**Diferencia entre backtracking y B&B 11**](#_hie3o8qghh5x)

[**Caballo de ajedrez 11**](#_3kqko1s1c4ze)

[Descripción del problema 11](#_qwzalsjaenc3)

[Reglas 12](#_m6ds0y9wwp3y)

[Heurista de H. C. von Warnsdorff 12](#_xthg943z7p4)

[Pseudocodigo 14](#_h2zbb11ts3f1)

[Backtracking 14](#_sraqszvfhttu)

[Branch & Bound 15](#_nnp03j2u777v)

[Algoritmo 16](#_yebwqj3imaz3)

[Backtracking 16](#_ugdcxwjmmzuw)

[Branch & Bound 17](#_uk24bi91ptlz)

[Analisis de Costos 18](#_ne8cchgu6ikv)

[Backtracking 18](#_l9gzu79yyqc)

[Branch & Bound 19](#_od9unmwo5rfn)

[**Comparaciones 19**](#_qeno2brj0ypv)

[Cantidad de nodos explorados 20](#_isbbxeqhp1e9)

[Tiempo de ejecución 27](#_xs19c9xu9i4n)

[**Entregables 33**](#_bybz02i9mrcp)

[**Bibliografia 33**](#_oxg9mes1l0e2)

# Introducción

En el presente trabajo se aborda el clásico problema del caballo de ajedrez desde una perspectiva algorítmica, proponiendo una solución basada en la técnica de backtracking combinada con la aplicación de branch and bound.

El objetivo principal consiste en desarrollar e implementar un algoritmo eficiente que determine una secuencia de movimientos de un caballo en un tablero de ajedrez de tamaño dado, de manera que el caballo visite cada casilla exactamente una vez. A través de la exploración exhaustiva del espacio de búsqueda y la aplicación de estrategias de poda, se busca encontrar una solución óptima o, en su defecto, una solución factible en un tiempo de ejecución razonable.

# Objetivo

El objetivo es implementar y comparar diferentes enfoques algorítmicos para resolver el problema del recorrido del caballo en un tablero de ajedrez, utilizando las técnicas de Backtracking y Branch & Bound (B&B). El recorrido debe visitar todas las casillas del tablero exactamente una vez, en lo que se conoce como el problema del caballo.

# Backtracking

El backtracking es una técnica algorítmica de resolución de problemas que implica explorar todas las diferentes opciones de forma incremental y deshacerlas si conducen a un callejón sin salida (poda). Es como probar diferentes caminos en un laberinto, y si un camino no conduce a la salida, vuelves sobre tus pasos para probar otra ruta.

La representación del backtracking se realizar sobre grafos dirigidos acíclicos (construcción lógica de posibles alternativas a evaluar​).

## ¿Cómo funciona?

1. El algoritmo comienza con un estado inicial y toma una decisión.
2. Comprueba si la elección conduce a una solución válida o a un callejón sin salida.
3. Si la elección es válida, explora recursivamente más posibilidades.
4. Si se llega a un callejón sin salida, el algoritmo retrocede hasta el punto de elección anterior y prueba una opción diferente.
5. La solución del problema se presenta en una lista ordenada (X1, X2, …, Xn)

## Puntos clave

Backtracking explora todas las soluciones posibles de manera sistemática. Se implementa de forma recursiva, lo que la hace elegante y concisa debido a su estrategia de búsqueda exhaustiva.

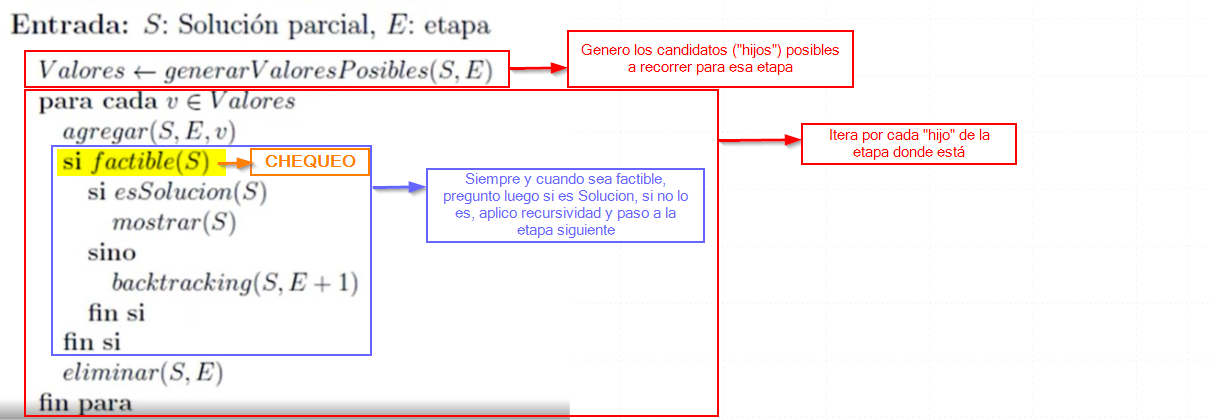
Se utiliza en varios dominios de resolución de problemas, incluidos:

* Sudoku, rompecabezas de 8 reinas, etc.
* Problema del viajante, Problema de la mochila, etc.
* Juegos: Ajedrez, tres en línea, etc.

En esencia, el backtracking es una herramienta poderosa para resolver problemas que implican explorar un gran espacio de búsqueda. La idea es encontrar la mejor solución posible en un momento determinado, por eso, se dice que este tipo de algoritmo es una búsqueda en profundidad, aun habiendo encontrado una solución, voy a seguir recorriendo todo el árbol

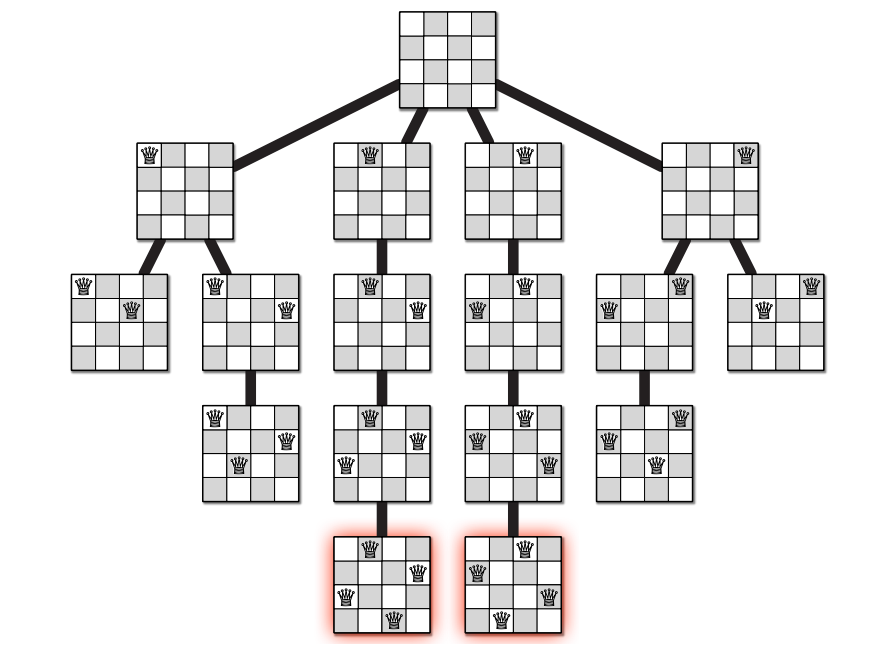
Durante la búsqueda, si se encuentra una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción (*“hijo”* si nos referimos a un árbol). Si no hay más alternativas la búsqueda falla, es decir que existen casos donde **no se encuentra una solución**. De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución (solución parcial en el caso de nodos interiores o solución total en el caso de los nodos hoja).

## Pseudocódigo base



## Ilustración a modo de ejemplo

Este clásico rompecabezas nos plantea colocar estratégicamente N reinas en un tablero de ajedrez N×N sin que ninguna de ellas pueda atacarse entre sí. Seremos testigos de cómo el algoritmo de retroceso navega ingeniosamente a través de las posibilidades, creando una sinergia entre la exploración y la toma de decisiones que nos lleva a soluciones válidas.



## 

## Complejidad Temporal

La complejidad temporal del backtracking es altamente variable y depende en gran medida de la estructura del problema específico que se esté resolviendo. No existe una fórmula única para calcularla, ya que puede variar desde casos polinomiales hasta exponenciales.

Estadísticamente hablando, **tiende a O(n2) siendo n la cantidad de nodos del grafo**

# Branch and bound

Es una variante del backtracking mejorado sustancialmente. Se aplica mayoritariamente para resolver cuestiones o problemas de optimización, mayormente de minimización, pero también aplicable a problemas de maximización.​.

Su principal virtud es que en la mayoría de las implementaciones se pueden evitar combinaciones, estableciendo funciones de acotación (o poda) reduciendo el tiempo de ejecución.

Se suele interpretar como un árbol de soluciones, donde cada rama conduce a una posible solución posterior a la actual. La característica de esta técnica es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para «podar» esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.

## ¿Cómo funciona?

1. Bifurcación

El problema se divide en subproblemas más pequeños, creando una estructura en forma de árbol.

Cada nodo del árbol representa un subproblema.

Se comienza por el nodo que promete llegar a una solución más rápidamente​

El proceso de ramificación continúa hasta que cada subproblema es lo suficientemente

simple como para resolverse directamente o hasta que queda claro que no puede conducir a una solución óptima.

2. Delimitación

Para cada subproblema se calcula un valor cota. Este valor representa la mejor solución posible que se puede obtener de ese subproblema.

Las cotas se pueden determinar utilizando diversas técnicas, como la relajación, la heurística o el conocimiento de un problema específico.

Los subproblemas con cotas que son peores que la mejor solución actual se pueden eliminar, eliminando la exploración innecesaria.

3. Exploración

El algoritmo explora el árbol, priorizando subproblemas con límites prometedores.

A medida que avanza la búsqueda, el algoritmo puede encontrar mejores soluciones, actualizando la mejor solución actual.

Si se detecta un nodo que nunca conducirá a una buena solución, se lo elimina​ (poda).

## Ramificación

La ramificación con una cola de prioridad (representación de los “nodos vivos”) es una estrategia de búsqueda inteligente que evita explorar todas las posibles soluciones al azar. Al priorizar los nodos más prometedores, se reduce significativamente el tiempo de búsqueda y se aumenta la probabilidad de encontrar una solución rápidamente. Esta técnica es especialmente útil para problemas de gran escala donde una búsqueda exhaustiva sería inviable

## Poda

En la búsqueda de soluciones, se puede descartar (podar) ciertos nodos si se determina que no pueden conducir a una solución mejor que las ya encontradas. Esto se basa en cotas: un valor mínimo (cota inferior) y máximo (cota superior) que limita el rango de posibles soluciones a partir de un nodo.

A tener en cuenta:

* Si el mejor valor que se podría alcanzar desde ese nodo es peor que el valor de la mejor solución encontrada hasta el momento, se puede podar dicho nodo. A esto se lo conoce como “***Cota Inferior***”
* Si el peor valor desde donde estoy es mejor que el mejor valor que se puede alcanzar desde otro nodo, este otro nodo se puede podar. A esto se lo conoce como “***Cota Superior***”

## 

## Heurísticas

Algunas heurísticas son comúnmente usadas para acelerar el proceso. Como las variables se pueden procesar en cualquier orden, generalmente es más eficiente intentar ser lo más restrictivo posible con las primeras (esto es, las primeras con menores valores posibles). Este proceso poda el árbol de búsqueda antes de que se tome la decisión y se llame a la subrutina recursiva.

Algunas implementaciones muy sofisticadas usan una función de cotas, que examina si es posible encontrar una solución a partir de una solución parcial. Además, se comprueba si la solución parcial que falla puede incrementar significativamente la eficiencia del algoritmo. Para el uso de estas funciones de cota, se debe ser muy minucioso en su implementación de forma que sean poco costosas computacionalmente hablando, ya que lo más normal es que se ejecuten en para cada nodo del algoritmo.

Con el objetivo de mantener la solución actual con coste mínimo, los algoritmos de backtracking mantienen el coste de la mejor solución en una variable que va actualizándose con cada nueva mejor solución encontrada. Así, si una solución es peor que la que se acaba de encontrar, el algoritmo no actualizará la solución. De esta forma, devolverá siempre la mejor solución que haya encontrado.

## Otras Opciones de Poda

* **Poda por Distancia hacia el Centro del Tablero**: Esta estrategia prioriza los movimientos hacia el centro del tablero.
* **Poda por Casillas Visitadas Cercanas**: Este criterio evita movimientos hacia casillas que se encuentran rodeadas de posiciones ya visitadas.
* **Poda por Recursividad Controlada**: Limitar la profundidad de la recursión en ciertos casos (tableros muy grandes).
* **Poda de Movimiento Retornable**: En esta estrategia, se busca que el caballo siempre pueda regresar a áreas más abiertas del tablero sin quedar encerrado.
* **Poda basada en un Árbol de Decisión Probabilístico**: Para esto se utiliza un modelo probabilístico para determinar la "viabilidad" de cada movimiento.

## Ventajas clave

* Garantiza que se consideren todas las soluciones potenciales.
* Reduce el espacio de búsqueda eliminando ramas subóptimas.
* Se puede aplicar a una amplia gama de problemas de optimización.
* Garantiza encontrar la solución óptima, con tiempo y recursos suficientes.

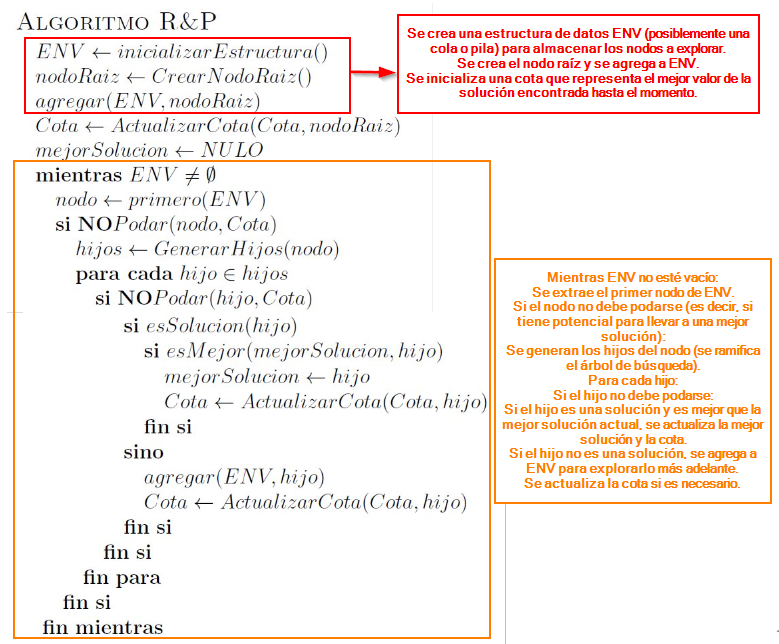
## Aplicaciones comunes

Resolución de problemas de optimización con variables enteras.

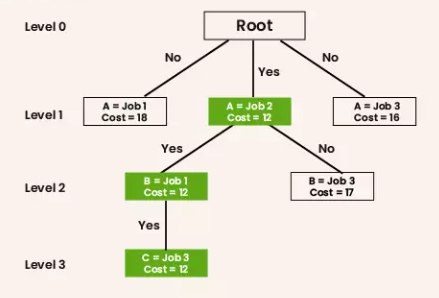
* Problema del viajante
* Problema de la mochila
* Programación de trabajos

Al combinar el poder de la ramificación y el límite, este algoritmo ofrece un enfoque sólido y eficiente para abordar desafíos complejos de optimización.

## Pseudocódigo base



## Ilustración a modo de ejemplo



El método Branch & Bound sigue la técnica de dividir un problema en varias partes pequeñas de problemas. También conocido como TSP (Problema del viajero), consiste en encontrar la ruta más corta que visite cada ciudad de una lista exactamente una vez, y regrese al punto de partida.

## 

## Complejidad Temporal

Si bien en el peor caso, la complejidad temporal de Branch and Bound (B&B) puede ser similar al backtracking tradicional, este algoritmo ofrece ventajas significativas en el caso promedio.

Al emplear estrategias inteligentes de ramificación y poda, B&B logra descartar una gran cantidad de nodos irrelevantes, reduciendo drásticamente el espacio de búsqueda y mejorando considerablemente la eficiencia del proceso

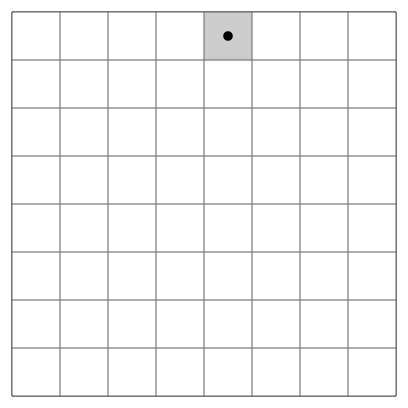
# Diferencia entre backtracking y B&B

* El Backtracking se utiliza para encontrar todas las posibles soluciones disponibles para un problema. Cuando se da cuenta de que ha tomado una mala decisión, deshace la última elección retrocediendo. Busca en el árbol de espacio de estados hasta que encuentra una solución para el problema.
* Branch-and-Bound se utiliza para resolver problemas de optimización. Cuando se da cuenta de que ya tiene una mejor solución óptima a la que conduce la el nodo, abandona ese nodo y realiza una búsqueda completa en el árbol de espacio de estados para obtener la solución óptima.

# Caballo de ajedrez

## Descripción del problema

Dado un tablero de ajedrez de tamaño N×N y una posición inicial para el caballo, el objetivo es encontrar un camino en el que el caballo visite cada casilla exactamente una vez. El caballo se mueve en forma de "L", es decir, dos casillas en una dirección (vertical u horizontal) y luego una casilla en una dirección perpendicular, o una casilla en una dirección y dos casillas en la perpendicular.

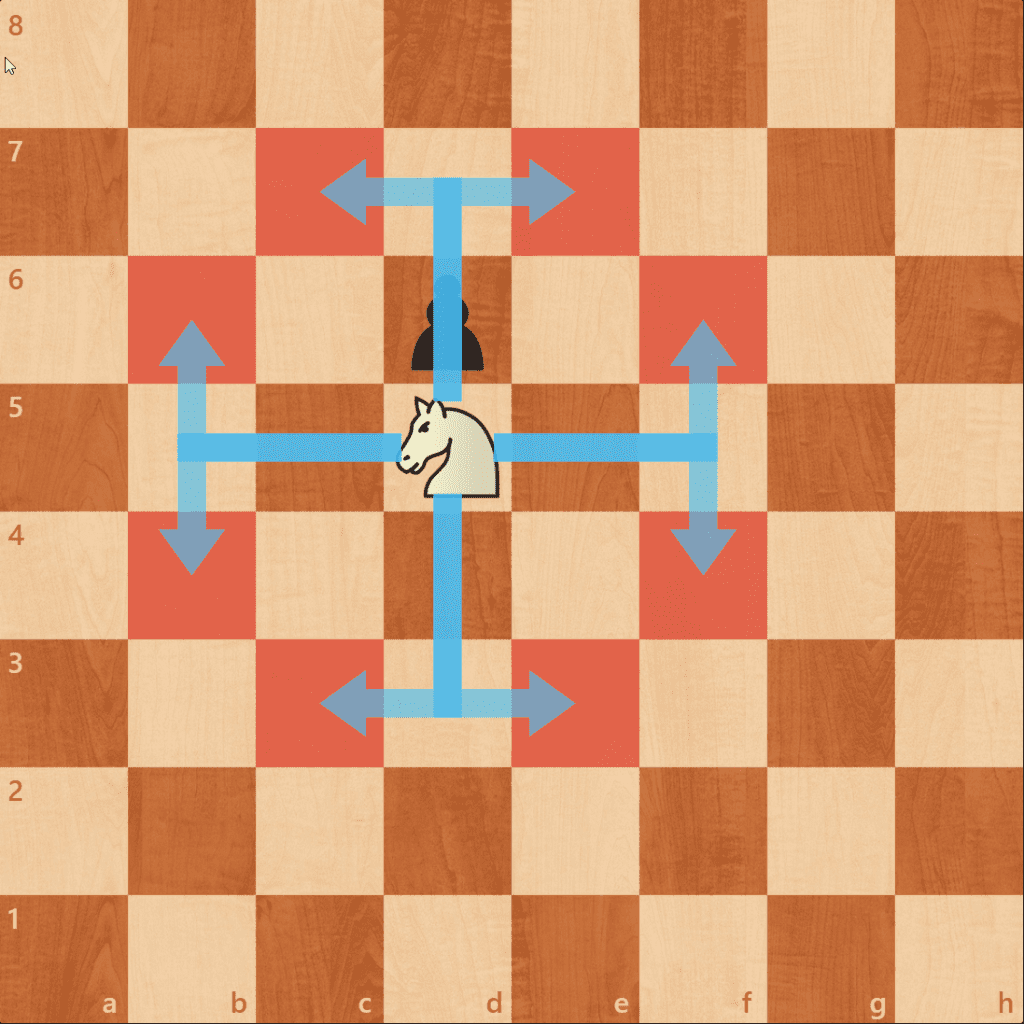


***Solución para 63 saltos de caballo***

***por las 64 casillas.***

## Reglas

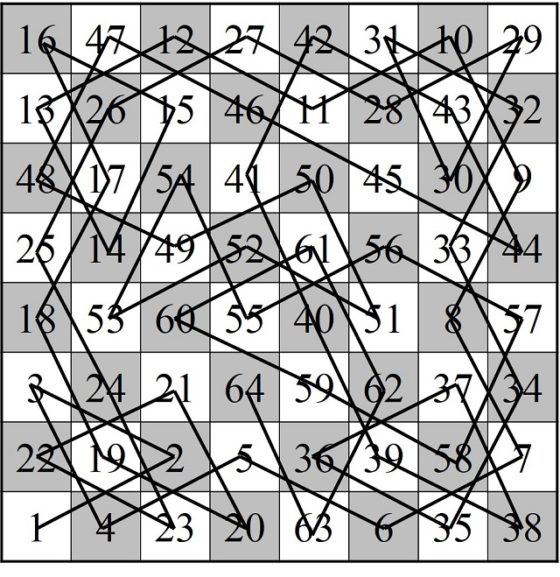
* El caballo se desplaza varias casillas con cada jugada, para este trabajo el caballo **visita cada casilla exactamente una vez**.
* Puede moverse una casilla hacia adelante o hacia atrás verticalmente y entonces otras dos a izquierda o derecha de manera horizontal
* O bien dos casillas hacia adelante o hacia atrás verticalmente y entonces una a izquierda o derecha de manera horizontal.
* Una forma bastante efectiva de recordar su movimiento es pensar en la forma de una "L".



## Heurista de H. C. von Warnsdorff

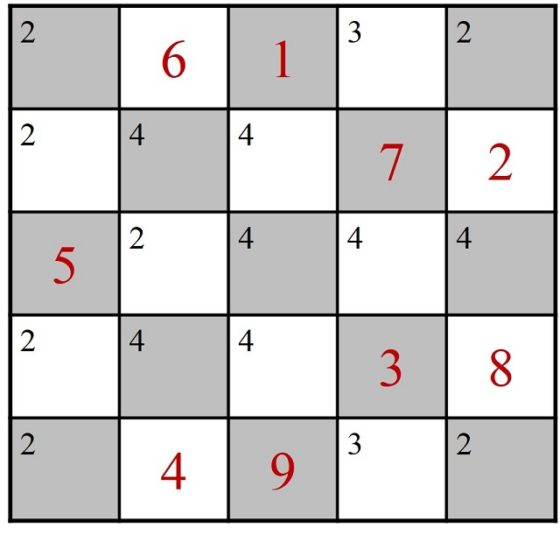
El alemán H. C. von Warnsdorff propuso un método sencillo y elegante de resolver el problema del recorrido del caballo, en su publicación titulada “*La solución más sencilla y general al salto del caballo (1823), que podéis encontrar en internet”*.

La técnica de H. C. von Warnsdorff para encontrar recorridos del caballo sobre el tablero 8 x 8 es la siguiente: Mueve el caballo a una casilla adyacente (es decir, a la que se llegue con el movimiento del caballo), que no haya sido visitada ya y que tenga el menor número de casillas adyacentes libres. Si se da un empate, elegir de forma arbitraria.

Esta es una regla intuitiva, puesto que indica que hay que visitar primero las casillas con menos conexiones, que son aquellas que si no se recorren en ese momento quizás, por la falta de conexiones, no se pueda regresar después a pasar por ellas, que realmente se puede utilizar para todo tipo de tableros rectangulares, que es bastante eficaz a la hora de buscar algunos recorridos particulares del caballo, aunque no es válida para una búsqueda exhaustiva, por ejemplo, los recorridos de Euler o Al-Adli sobre el tablero clásico no verifican la regla de von Warnsdorff.

Sin embargo, el método no resuelve bien los casos en los que hay empates en el número de casillas adyacentes libres. Si se elige al azar la casilla en la que continuar, como sugiere Warnsdorff, puede ocurrir que el recorrido se pare en un momento intermedio y no pase por todas las casillas del tablero, como se muestra en el siguiente ejemplo 5 x 5.

Si utilizamos la regla de Warnsdorff para un tablero 5 x 5, empezando en la casilla del medio de la primera fila, el primer empate se produce al llegar a la casilla 9, que tiene dos opciones de continuidad, dos casillas con número de casillas adyacentes libres igual a 2, en el lateral y debajo de la casilla número 5.



## Pseudocódigo

### Backtracking

|  |
| --- |
| **def resolverTablero(n, pos\_x,pos\_y):**  Tablero = generarTablero(n) ***//genero matriz nxn con -1***  Inicializar movimientos\_x=[] ***//[2, 1, -1, -2, -2, -1, 1, 2]***  Inicializar movimientos\_y=[] ***//[1, 2, 2, 1, -1, -2, -2, -1]***  tablero[pos\_x][pos\_y]=0 ***//Inicializo posición inicial***  pos = 1 ***//primera posición***  Si pos recorrió todos los casilleros  Muestro el tablero  Si no los recorrió, chequeo el primero movimiento de los 8 que puede hacer un caballero en un escaque posible  ***//new\_x = curr\_x + move\_x[i]***  ***//new\_y = curr\_y + move\_y[i]***  Pregunto si es factible  ***//Si x e y se encuentran dentro de los límites del tablero***  ***//Y que el caballero no haya pasado por ese escaque***  Si lo es, muevo el caballero a la nueva posición.  ***//board[new\_x][new\_y] = pos***  Hago recursividad desde el nuevo escaque y chequeo movimientos posibles a partir de ahí  ***//if(factible(n, board, new\_x, new\_y, move\_x, move\_y, pos+1)):***  Si no lo es, vuelvo un movimiento atrás y pruebo desde el próximo movimiento posible de los 8 mencionados anteriormente.  ***//board[new\_x][new\_y] = -1***  ***//i++***  Si no tengo más movimientos imprimo que no hay solución |
|  |

### Branch & Bound

|  |
| --- |
| **// Generar tablero nxn inicializado con -1**  **generarTablero(n)**  **// Los 8 movimientos posibles del caballo en el eje x e y**  **movimientos\_x = [2, 1, -1, -2, -2, -1, 1, 2]**  **movimientos\_y = [1, 2, 2, 1, -1, -2, -2, -1]**  **// Colocar la posición inicial del caballo**  **tablero[pos\_x][pos\_y] = 0**  **// Primera posición**  **pos = 1**  **// Si el Caballo recorrió todos los escaques Imprimo el tablero**  **si pos == n \* n:**  **imprimirTablero**  **// Si no los recorrió, Genero una cola de prioridad con los movimientos factibles del caballo**  **(dentro de los límites del tablero y no haya pasado por ese escaque).**  **Siendo los óptimos los más cercanos a los bordes.**  **//Si la cola de prioridad posee mas o igual de 4 movimientos, mi cota superior será de 2.**  **//Si no la cola mantendrá su tamaño**  **Muevo el Caballo a la posición más óptima.**  **Hago recursividad desde el nuevo escaque y chequeo nuevamente**  **los movimientos más óptimos posibles.**    **//Si ese movimiento es un camino sin solución, vuelvo un movimiento atrás y pruebo desde el próximo**  **movimiento posible de la cola de prioridad.**  **board[new\_x][new\_y] = -1**  **//Si no tengo más movimientos imprimo que no hay solución** |

## 

## Algoritmo

### Backtracking

|  |
| --- |
| **def isSafe(x, y, board, size):**  return 0 <= x < size and 0 <= y < size and board[x][y]==-1  **def printSolution(n, board):**  for i in range(n):  for j in range(n):  print(board[i][j], end=' ')  print()  **def solveKT(n, pos\_x, pos\_y):**  **# Inicialización Matriz**  board = [[-1 for \_ in range(n)] for \_ in range(n)]  **# Movimientos posibles**  move\_x = [2, 1, -1, -2, -2, -1, 1, 2]  move\_y = [1, 2, 2, 1, -1, -2, -2, -1]  board[x\_position][y\_position] = 0  pos = 1  if not solveKTUtil(n, board, x\_position, y\_position, move\_x, move\_y, pos):  print("Solution does not exist")  else:  printSolution(n, board)  **def solveKTUtil(n, board, curr\_x, curr\_y, move\_x, move\_y, pos):**  if pos == n\*\*2:  return True  for i in range(8):  new\_x = curr\_x + move\_x[i]  new\_y = curr\_y + move\_y[i]    if isSafe(new\_x, new\_y, board, n):  board[new\_x][new\_y] = pos  if solveKTUtil(n, board, new\_x, new\_y, move\_x, move\_y, pos + 1):  return True  board[new\_x][new\_y] = -1    return False |

### Branch & Bound

|  |
| --- |
| **def solveKT(n, pos\_x, pos\_y):**  **# Inicialización Matriz**  **board = [[-1 for \_ in range(n)] for \_ in range(n)]**    **# Movimientos posibles**  **move\_x = [2, 1, -1, -2, -2, -1, 1, 2]**  **move\_y = [1, 2, 2, 1, -1, -2, -2, -1]**    **# Posicion Inicial**  **board[x\_position][y\_position] = 0**  **# Contador**  **pos = 1**    **# Pregunto si es Solucion**  **if not solveKTUtil(n, board , x\_position, y\_position, move\_x , move\_y , pos):**  **print("Solution does not exist")**  **else:**  **printSolution(n, board )**    **def solveKTUtil(n, board , curr\_x, curr\_y, move\_x, move\_y, pos):**  **if pos == n\*\*2:**  **return True**    **for i in range(8):**  **# Movimiento posible en X e Y**  **new\_x = curr\_x + move\_x[i]**  **new\_y = curr\_y + move\_y[i]**    **#Pregunto si es válido**  **if isSafe(new\_x, new\_y, board, n):**  **board[new\_x][new\_y] = pos**  **#Recursividad con nueva posicion**  **if solveKTUtil(n, board, new\_x, new\_y, move\_x, move\_y, pos + 1):**  **return True**    **#Si no es válido, vuelvo a marcar -1**  **board[new\_x][new\_y] = -1**    **#Si no hay solución, retorno False**  **return False**    **def isSafe(x, y, board, size):**  **#X e Y se encuentran dentro de los limites**  **#Esta casilla no fue visitada**  **return 0 <= x < size and 0 <= y < size and board[x][y] == -1**  **</code></pre>**  **</div>**  **</section>**  **<section>**  **<h4>Algoritmo Branch & Bound</h4>**  **<div class="code-container">**  **<pre class="language-python">**  **<code data-line-numbers="|1-17|13|19-31|24|33-48|42|50-55|33-48|44-46|48|57-60|48|19-31|26-31|15-17|68-73|">**  **def solveKT(n):**  **start\_time = time.time()**  **board = [[-1 for i in range(n)]for i in range(n)]**  **move\_x = [2, 1, -1, -2, -2, -1, 1, 2]**  **move\_y = [1, 2, 2, 1, -1, -2, -2, -1]**  **board[x\_position][y\_position] = 0**  **pos = 1**  **if(not solveKTUtil(n, board, x\_position, y\_position, move\_x, move\_y, pos, bkalg)):**  **print("Solution does not exist")**  **else:**  **print(f"--- {time.time() - start\_time} seconds ---")**  **printSolution(n, board)**    **def solveKTUtil(n, board, curr\_x, curr\_y, move\_x, move\_y, pos):**  **if(pos == n\*\*2):**  **return True**    **cola\_prioridad = branch(curr\_x, curr\_y, n, board, move\_x, move\_y)**  **for \_, new\_x, new\_y in cola\_prioridad:**  **board[new\_x][new\_y] = pos**  **if(solveKTUtil(n, board, new\_x, new\_y, move\_x, move\_y, pos+1, bkalg)):**  **return True**  **board[new\_x][new\_y] = -1**  **return False**  **def branch(curr\_x, curr\_y, n, board, move\_x, move\_y):**    **priority\_queue = []**  **for i in range(8):**  **new\_x = curr\_x + move\_x[i]**  **new\_y = curr\_y + move\_y[i]**  **if isSafe(new\_x, new\_y, board, size=len(board)):**  **# Guarda los movimientos con sus distancias mas cercanas del centro**  **distance = distanciaHaciaBordes(new\_x, new\_y, n)**  **priority\_queue.append((distance, new\_x, new\_y))**    **# Ordena los movimientos priorizando los mas cercanos a los bordes**  **priority\_queue.sort(reverse=True, key=lambda move: move[0])**    **return bound(priority\_queue)**  **def distanciaHaciaBordes(x, y, n):**  **center = (n - 1) / 2**  **dx = x - center**  **dy = y - center**  **distance = (dx \* dx + dy \* dy) \*\* 0.5**  **return distance**  **def bound(moves):**  **if len(moves)>=4:**  **moves=moves[:1]**  **return moves**  **def isSafe(x, y, board):**  **if 0 <= x < len(board) and 0 <= y < len(board) and board[x][y] == -1:**  **return True**  **return False**  **def printSolution(n, board):**  **for i in range(n):**  **for j in range(n):**  **print(board[i][j], end=' ')**  **print()** |

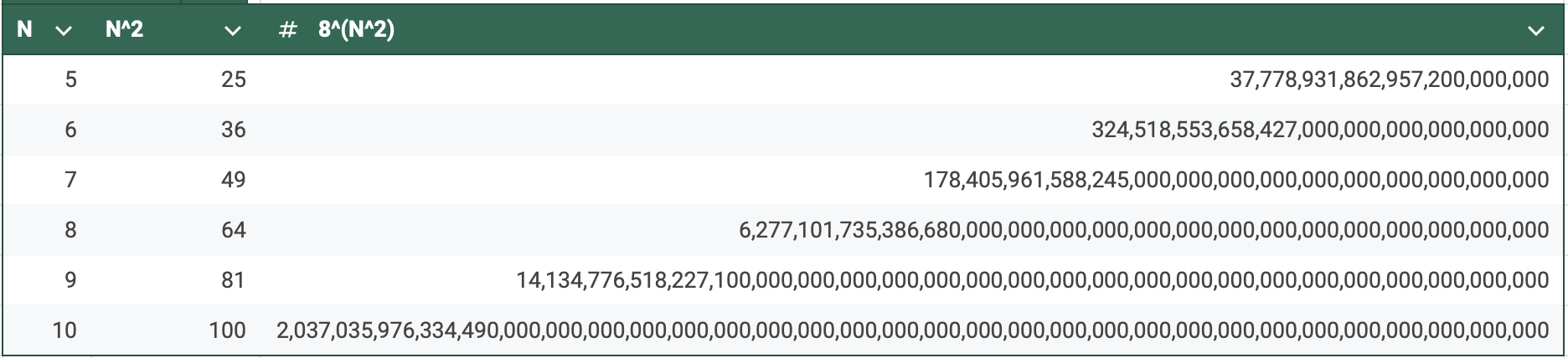
## Análisis de Costos

### Backtracking

La complejidad temporal de este problema utilizando Backtracking normalmente

se analiza cómo **O(8^(N^2))** - **Exponencial**, donde:

* N es el tamaño del tablero de ajedrez. Un tablero de ajedrez (N x N) tiene N^2 casillas.
* Movimientos máximos por casilla: en cada casilla, el caballero puede realizar como máximo 8 movimientos posibles.



### Branch & Bound

En el peor caso, la complejidad temporal de la regla de Warnsdorff es **O(3^(N^2)) - Cuadrático**, donde:

* **N^2** es el número de casillas del tablero
* **3** es el máximo número de movimientos posibles del caballo luego de realizar la poda



# Comparaciones

Se realizaron algunas comparaciones entre las técnicas Backtracking y Branch And Bound con el objetivo de visualizar la cantidad de nodos explorados y el tiempo de ejecución de cada uno para diferentes tamaños de tablero.

Se utilizó Google Colab para la realización de las comparaciones. Este ofrece gratuitamente una computadora con 12gb de RAM y una CPU con 2 núcleos.

* [Colab File con Comparaciones y Métricas](https://colab.research.google.com/drive/1eNmOtows-W0upOWBcSgDQovIdotYWTo8?usp=sharing)

Con las limitaciones técnicas anteriormente dichas, la implementación para obtener los diferentes casos fue:

* **Timeout**: Un timeout de 30 segundos para cada proceso del Recorrido del Caballo
* **Contador de nodos explorados**: Un contador que refleja la cantidad de nodos que explora el caballo para obtener una solución
* **Tiempo de ejecución**: Diferencia entre el tiempo en que se inició el algoritmo y finalizó
* **Ejecución en paralelo**: Un algoritmo que permite ejecutar otros algoritmos en paralelo y no secuencialmente
* **Generador de casos**: Un algoritmo que devuelve las diferentes posiciones del tablero según su tamaño

Hubo tamaños específicos del tablero de backtracking que la implementación para obtener los datos de comparación para todos los casos del tablero, hacia que ocupe toda la RAM y por lo tanto corte la ejecución.

La solución que se encontró fue obtener datos parciales, siendo estos los resultados de cada fila del tablero ejecutándose en paralelo cada columna de la fila, y por cada fila, secuencialmente. Además, se guardaron los resultados en un archivo para un trabajo ágil.

Con esto se redujo considerablemente la cantidad de ejecuciones en paralelo.

## Cantidad de nodos explorados

A continuación, se visualizan diferentes tableros, primero con los datos para cada técnica, y luego la comparación.

Cada casillero del tablero representa la posición inicial del caballo, el color verde representa que en esa posición tuvo solución, el color rojo representa que en esa posición no tuvo solución o que se sufrió un timeout, pero ya en la comparativa se ve que el color amarillo es ciertamente por este timeout mencionado, y si es rojo simplemente no hubo solución, como es el caso de 5x5.

El valor dentro del casillero es diferente para los tableros por cada técnica y la comparación. El valor por cada técnica representa la cantidad de nodos que necesitó explorar para llegar a una solución.

El valor en la comparación es un conjunto de valores. El primero representa la diferencia de nodos explorados entre los nodos de branch and bound y backtracking. Por lo que, si el valor es positivo, branch and bound llegó a una solución con menos nodos explorados que backtracking. El segundo representa el porcentaje en eficiencia entre los nodos de branch and bound y backtracking.

Tablero 5x5 con Backtracking



Tablero 5x5 con Branch & Bound



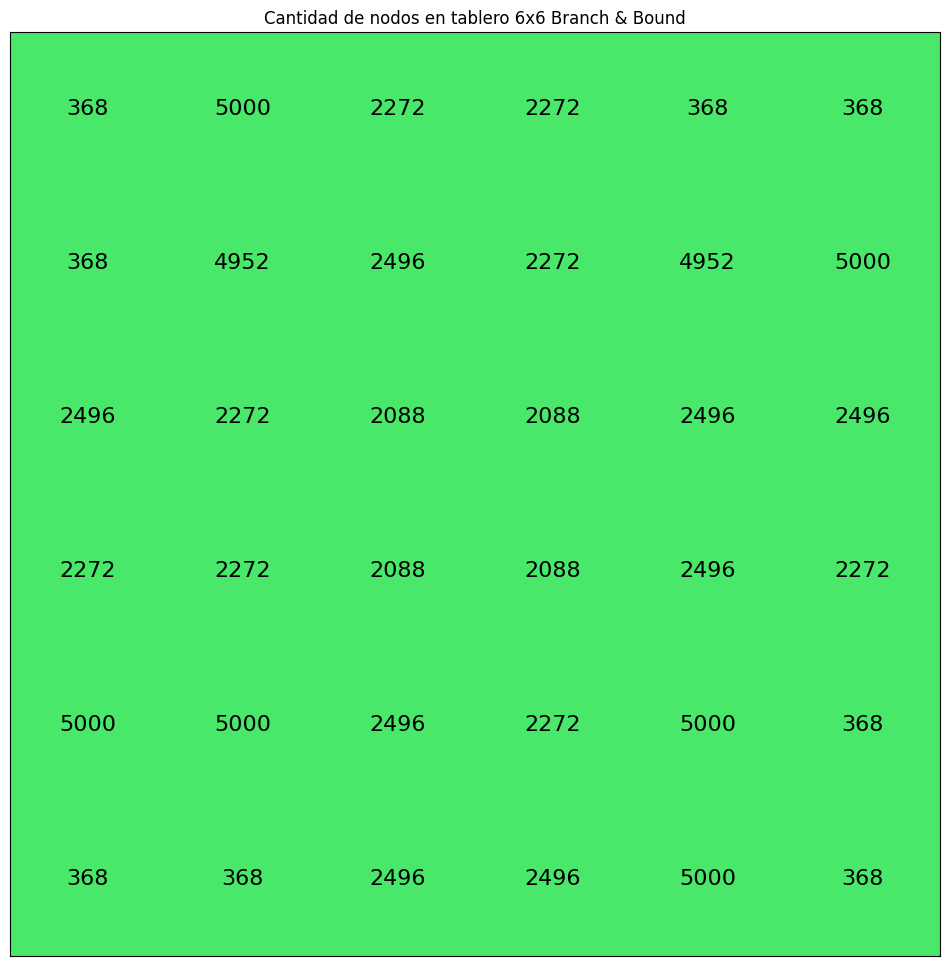
Tablero 5x5 comparativo



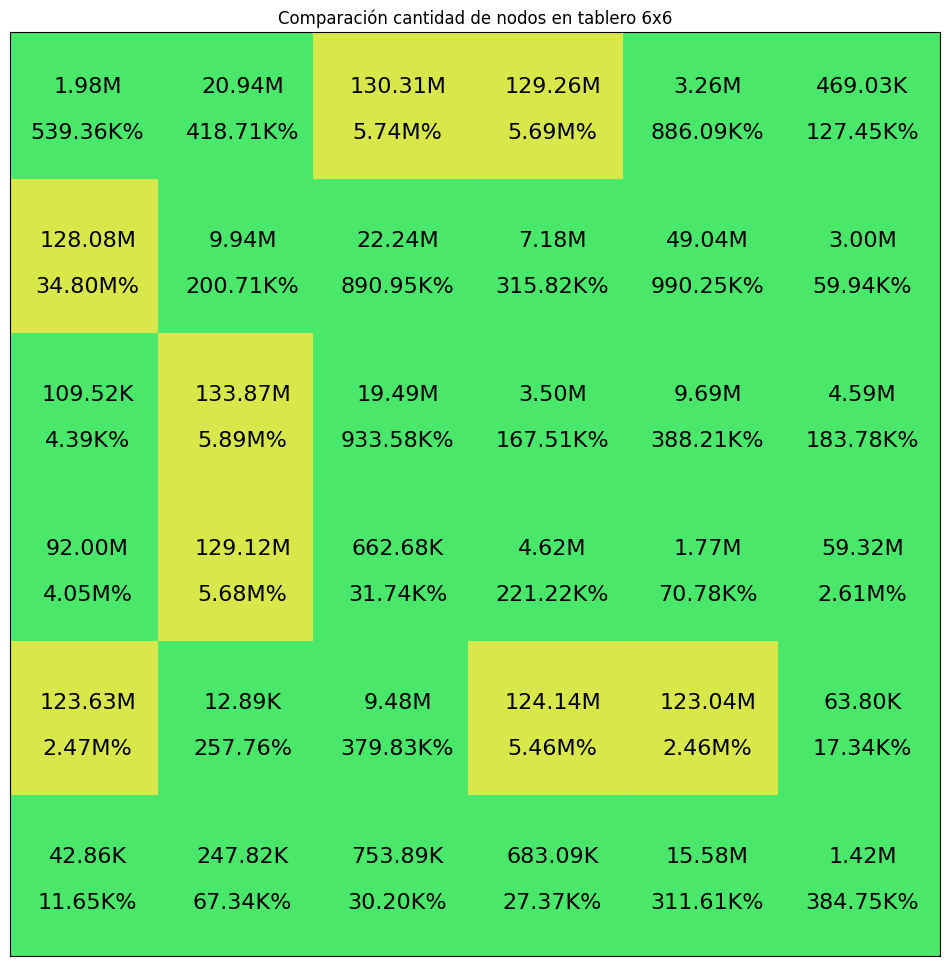
Tablero 6x6 con Backtracking



Tablero 6x6 con Branch & Bound



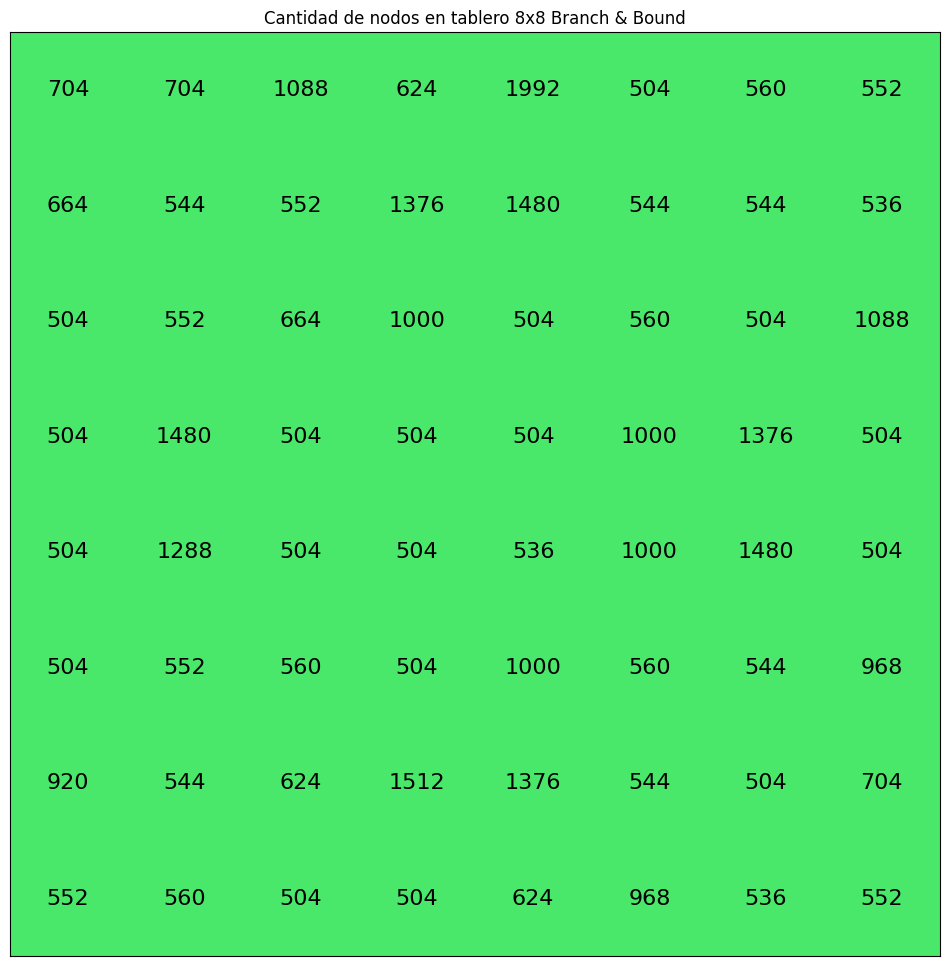
Tablero 6x6 comparativo



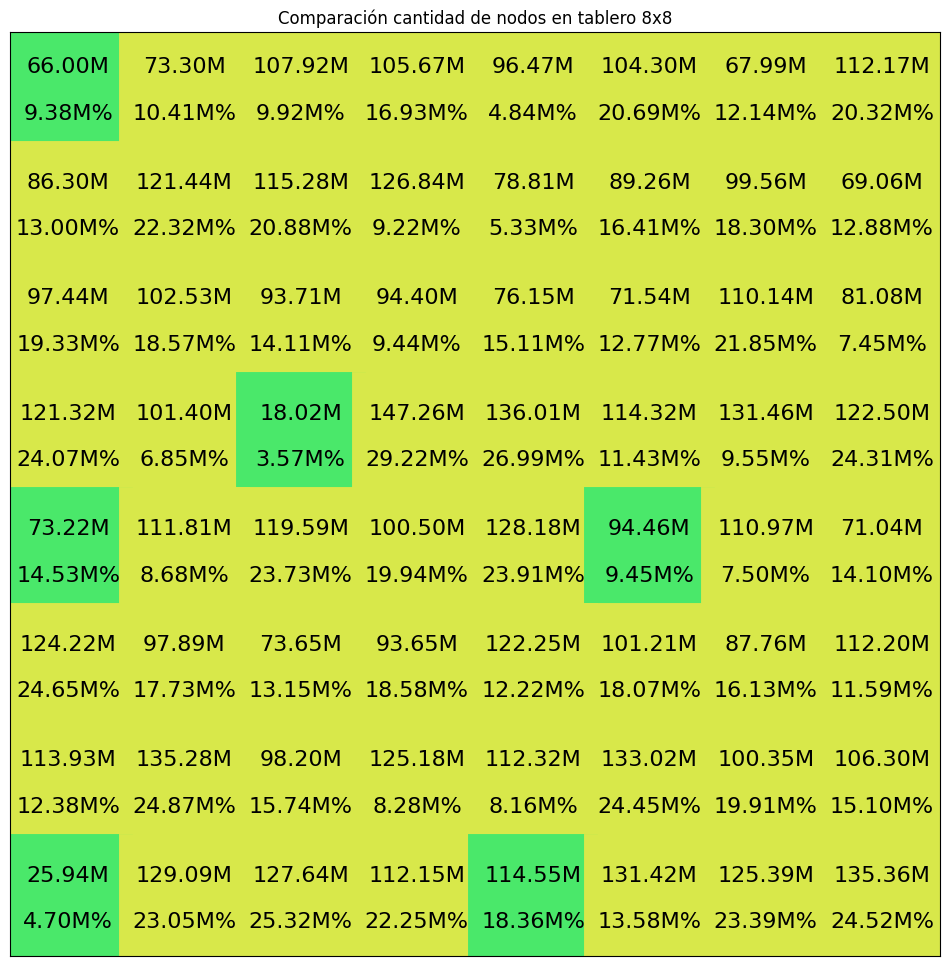
Tablero 8x8 con Backtracking



Tablero 8x8 con Branch & Bound



Tablero 8x8 comparativo



## Tiempo de ejecución

A continuación, se visualizan diferentes gráficos que demuestran el tiempo de ejecución de cada técnica, junto a un tablero final comparativo

El eje x de los gráficos representan las posiciones del tablero donde inició el caballo y el eje y representa el tiempo de ejecución en encontrar una solución.

Gráfico 5x5 con Backtracking

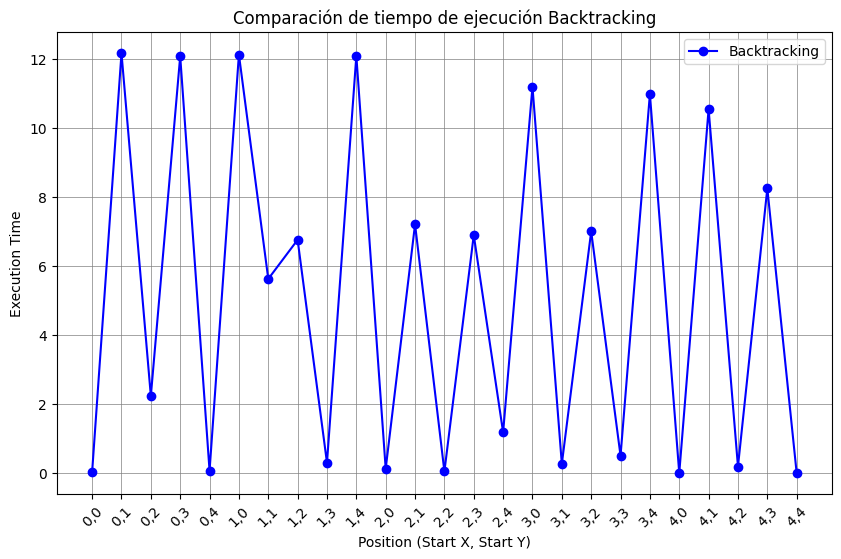
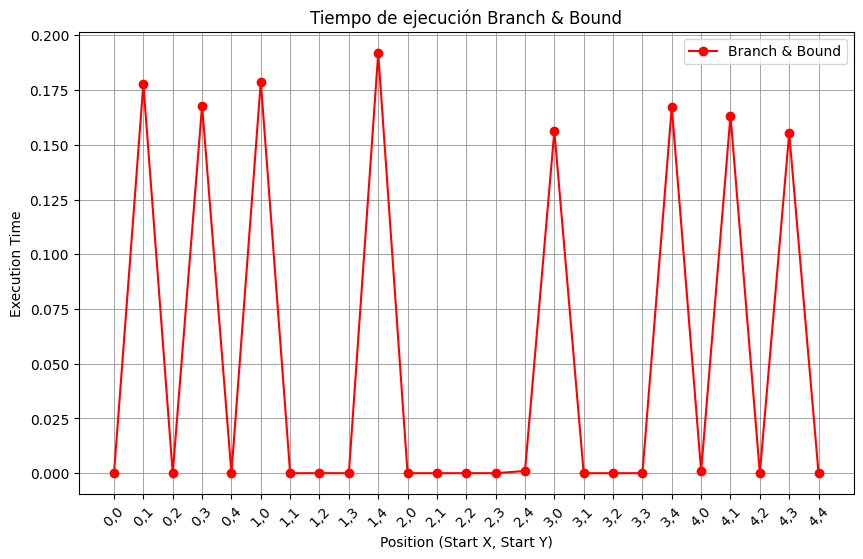


Gráfico 5x5 con Branch & Bound



Tablero 5x5 comparativa tiempo



Gráfico 6x6 con Backtracking

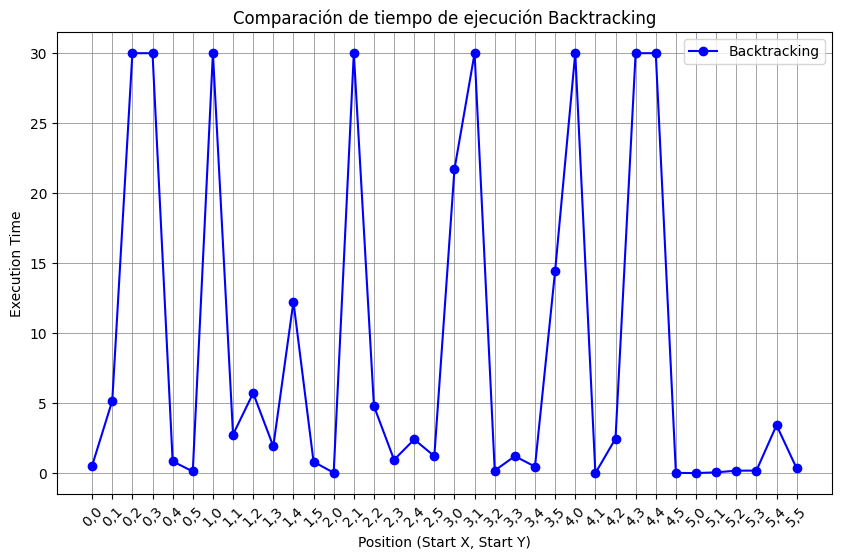
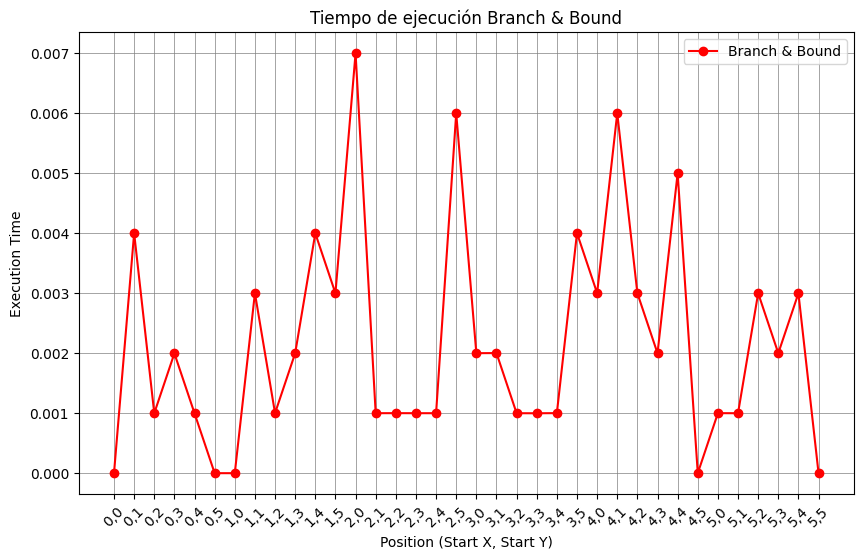


Gráfico 6x6 con Branch & Bound



Tablero 6x6 comparativo



Gráfico 8x8 con Backtracking

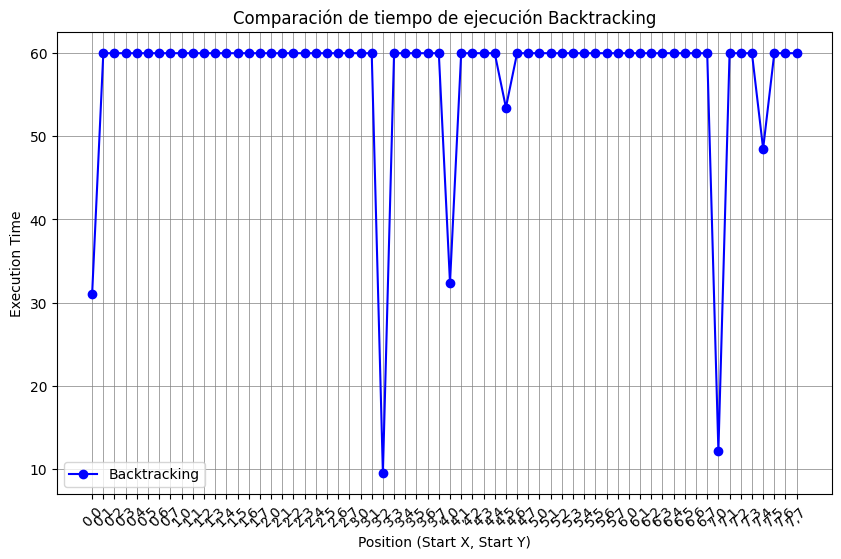
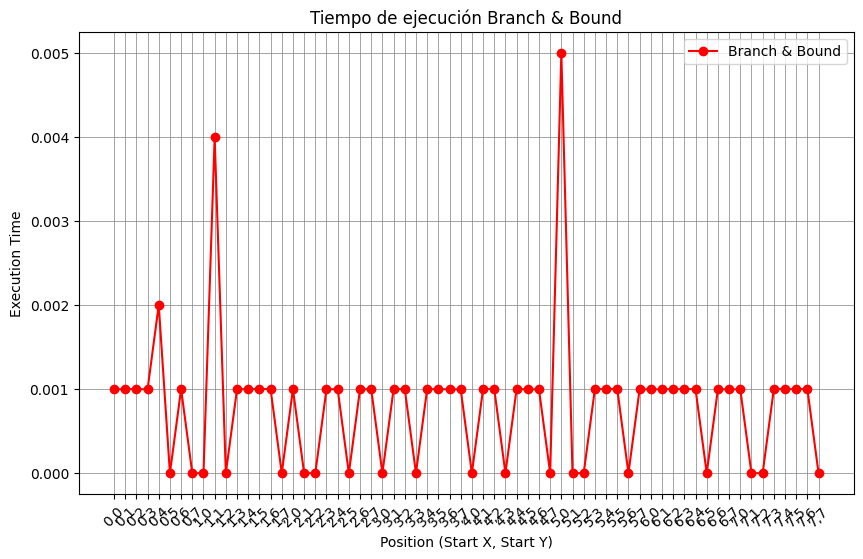
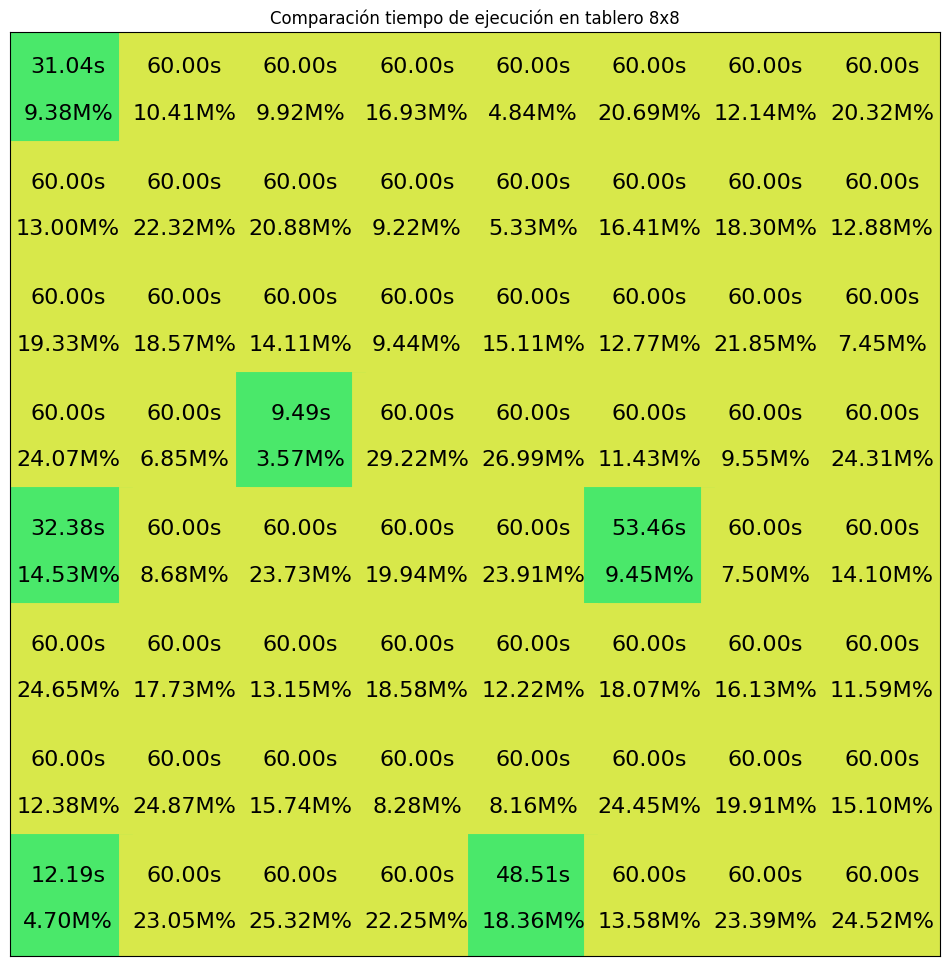


Gráfico 8x8 con Branch & Bound



Tablero 8x8 comparativo



A pesar de que a medida que va aumentando el tamaño del tablero, aumentan los casos de Backtracking con timeout, se puede observar que la gran mayoría de los casos la técnica de Branch & Bound es mucho más eficiente tanto en cantidad de nodos como en tiempo de ejecución.

Para el tablero de mayor tamaño analizado (8x8), en Backtracking el mejor tiempo que obtuvimos fue de aproximadamente 10 segundos. Para el caso de Branch & Bound con el mismo tamaño de tablero, el peor tiempo que obtuvimos fue aproximadamente de 0,005 segundos

Hasta ahora en ninguno de los casos Bactracking fue superior a Branch and Bound. Todo esto debido a que la heurística aplicada fue pensada más que nada en términos de encontrar la primera solución.

En conclusión, Branch & Bound es más eficiente en todos los casos que Backtracking para los casos evaluados y también dejandonos ver que en caules backtrancking hubiese llegado a una solución de cualquier forma pero los recursos dados no se dejó demostrar.

# Entregables

* [Análisis y Comparaciones de los algoritmos](https://colab.research.google.com/drive/1eNmOtows-W0upOWBcSgDQovIdotYWTo8?usp=sharing)
* [Codigo - GitHub](https://github.com/CeciliaHuaman/UADE_G3_Caballo_Ajedrez)

# Bibliografia

* *The Knight’s tour problem*. (2011, July 14). GeeksforGeeks. [The Knight's tour problem - GeeksforGeeks](https://www.geeksforgeeks.org/the-knights-tour-problem/)
* *Warnsdorff’s algorithm for Knight’s tour problem*. (2017, March 28). GeeksforGeeks. [Warnsdorff's algorithm for Knight’s tour problem - GeeksforGeeks](https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/)
* ‌Wikipedia Contributors. (2021, May 30). *Knight’s tour*. Wikipedia; Wikimedia Foundation. [Knight's tour - Wikipedia](https://en.wikipedia.org/wiki/Knight%27s_tour)
* Weisstein, E. W. (n.d.). *Knight Graph*. Mathworld.wolfram.com. [Knight Graph -- from Wolfram MathWorld](https://mathworld.wolfram.com/KnightGraph.html)
* Ortiz, Carlos. (2024). *algoritmo de Warnsdorff*. Scribd. [Algoritmo de Warnsdorff | PDF | Ajedrez | Juegos competitivos](https://es.scribd.com/document/657147038/algoritmo-de-Warnsdorff)
* de, C. (2006, May 9). *método de diseño de algoritmos*. Wikipedia.org; Wikimedia Foundation, Inc. [Ramificación y poda - Wikipedia, la enciclopedia libre](https://es.wikipedia.org/wiki/Ramificaci%C3%B3n_y_poda)
* de, C. (2005, November 11). *estrategia para encontrar soluciones a problemas que satisfacen restricciones*. Wikipedia.org; Wikimedia Foundation, Inc. [Backtracking - Wikipedia, la enciclopedia libre](https://es.wikipedia.org/wiki/Vuelta_atr%C3%A1s)
* Jesús. (2023, August 19). *Explaining the backtracking algorithm - Jesús - Medium*. Medium. [Explaining the backtracking algorithm | by Jesús | Medium](https://medium.com/@jdgb.projects/explaning-the-backtracking-algorithm-a1c9d1159430)