



Universidad
del Caribe

2000

CANCUN, QUINTANA ROO, MÉXICO

CONOCIMIENTO Y CULTURA PARA EL DESARROLLO HUMANO

Desarrollo de un algoritmo eficiente para resolver un Sudoku utilizando una de las técnicas algorítmicas avanzadas: programación dinámica, divide y vencerás, o algoritmos voraces.

TECNICAS ALGORITMICAS

Luis Guillermo Mejenes Chacon
230300756 | 21 DE NOVIEMBRE DE 2024

Descripción del Problema:

Desarrolla un algoritmo eficiente para resolver un Sudoku utilizando una de las técnicas algorítmicas avanzadas: programación dinámica, divide y vencerás, o algoritmos voraces. Elige la técnica que consideres más adecuada y justifica tu selección en base a la estructura del problema y la complejidad computacional esperada.

Objetivos del Ejercicio:

- Determinar el método algorítmico óptimo entre programación dinámica, divide y vencerás, y algoritmos voraces para resolver un Sudoku.
- Aplicar la técnica seleccionada de forma eficiente, implementando el algoritmo en el lenguaje de programación de tu elección.
- Evaluar y reportar la complejidad computacional del algoritmo desarrollado.
- Entregar la solución al Sudoku resuelto y el tiempo de ejecución del algoritmo.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

1. Elección de Técnica:

De las técnicas evaluadas, backtracking se considera la más efectiva para resolver el problema del Sudoku. Este enfoque ofrece una solución directa y flexible que explora todas las combinaciones posibles de números en el tablero de manera sistemática, retrocediendo únicamente cuando se encuentra una contradicción con las reglas del juego. Su complejidad computacional en el peor de los casos es $O(9^n)$, donde n es el número de celdas vacías, lo que puede parecer elevado; sin embargo, las optimizaciones como heurísticas y propagación de restricciones reducen significativamente el espacio de búsqueda en la mayoría de los tableros prácticos. Esto lo convierte en una técnica eficiente para resolver tanto Sudokus simples como complejos.

En comparación, la programación dinámica no es adecuada debido a la falta de subproblemas independientes reutilizables, mientras que el enfoque de divide y vencerás falla porque las regiones del Sudoku no son independientes, ya que cada número colocado afecta múltiples filas, columnas y subcuadros. Además, el backtracking destaca por su facilidad de implementación y adaptación al problema, ya que permite validar las reglas del juego localmente y puede enriquecerse con optimizaciones específicas para mejorar su rendimiento. Por estas razones, el backtracking es la técnica más efectiva y práctica para abordar el problema del Sudoku.

2. Implementación:

https://github.com/JaceBeleren11/Sudoku_Solver

3. Evaluación y Análisis de Complejidad:

1. Backtracking (Algoritmos voraces)

Complejidad Temporal

El algoritmo de backtracking explora todas las configuraciones posibles del tablero. Si el tablero tiene $n = 9$ filas y columnas, y asumimos que hay k celdas vacías:

- En el peor caso, el algoritmo intenta 9 valores para cada celda vacía, generando una complejidad de $O(9^k)$

- Sin embargo, con heurísticas de validación como comprobar filas, columnas, y subcuadros, el espacio de búsqueda puede reducirse considerablemente.

Complejidad Espacial

El algoritmo utiliza:

- Un stack recursivo proporcional al número de celdas vacías, con una profundidad máxima de $O(k)$
- El tablero 9×9 ocupa $O(1)$, ya que su tamaño no cambia. Total: $O(k)$ adicional al espacio fijo del tablero.

2. Programación Dinámica

Complejidad Temporal

En teoría, la programación dinámica podría almacenar subproblemas como "soluciones parciales válidas" para evitar recalcular configuraciones ya evaluadas. Sin embargo, el número de estados posibles para un tablero completo es extremadamente grande:

- Para un tablero de 9×9 , hay 9^{81} configuraciones posibles, lo que hace que la complejidad temporal sea impráctica en términos de $O(9^{81})$.
- Incluso si se optimizan subproblemas, la dimensionalidad del problema hace que no sea aplicable en la práctica para un tablero completo.

Complejidad Espacial

El almacenamiento de subproblemas requiere un espacio proporcional al número de configuraciones parciales posibles:

- Esto equivale a $O(9^{81})$ en el peor caso, lo cual es inviable.

3. Divide y Vencerás

Complejidad Temporal

Este enfoque no se aplica directamente al Sudoku clásico, ya que no existe una manera obvia de dividir el problema en subproblemas independientes. Sin embargo, si el problema pudiera dividirse (por ejemplo, resolviendo bloques 3x3 independientemente):

- Cada subproblema tendría su propia complejidad de $O(9^k)$ y combinar soluciones sería costoso debido a la validación cruzada.
- En general, la complejidad del peor caso sigue siendo $O(9^k)$, similar al backtracking, pero con sobrecarga adicional para validar la interacción entre subproblemas.

Complejidad Espacial

El espacio requerido sería el mismo que el del backtracking:

- $O(k)$ para el stack recursivo en la resolución de subproblemas.

4. Reporte Final:

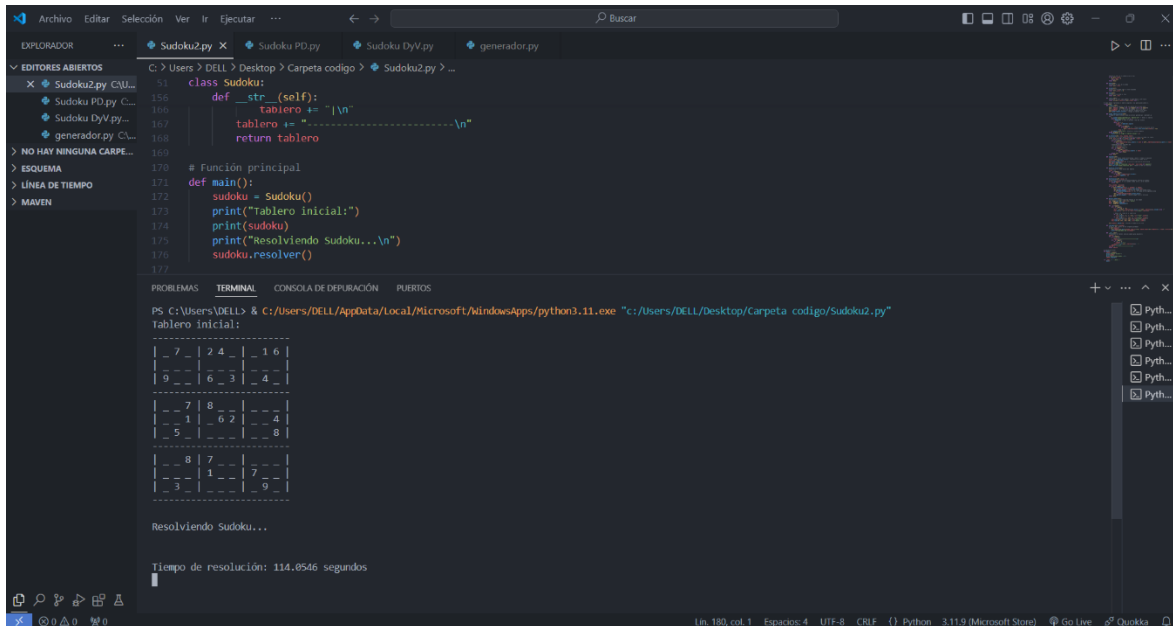
1. Justificación de la Técnica Seleccionada: Backtracking

El **backtracking** fue elegido como la técnica principal para resolver el problema de Sudoku debido a su capacidad para explorar todas las posibles configuraciones válidas de manera sistemática. Su simplicidad de implementación y su adaptabilidad lo convierten en la mejor opción para problemas con restricciones como el Sudoku. Además, permite incorporar verificaciones en tiempo real para reducir el espacio de búsqueda y evitar configuraciones inválidas.

2. Análisis de Complejidad Computacional

- **Complejidad Temporal:** En el peor caso, el backtracking tiene una complejidad de $O(9^k)$, donde k es el número de celdas vacías. Sin embargo, gracias a las validaciones (filas, columnas, y subcuadros), el espacio de búsqueda se reduce significativamente en la práctica.
- **Complejidad Espacial:** Requiere un stack recursivo proporcional al número de celdas vacías, con una complejidad espacial de $O(k)$. El tablero ocupa espacio constante ($O(1)$), ya que su tamaño no cambia.

3. Resultados Obtenidos



```
class Sudoku:
    def __str__(self):
        tablero += "\n"
        tablero += "-----\n"
        return tablero

# Función principal
def main():
    sudoku = Sudoku()
    print("Tablero inicial:")
    print(sudoku)
    print("Resolviendo Sudoku...\n")
    sudoku.resolver()

if __name__ == '__main__':
    main()
```

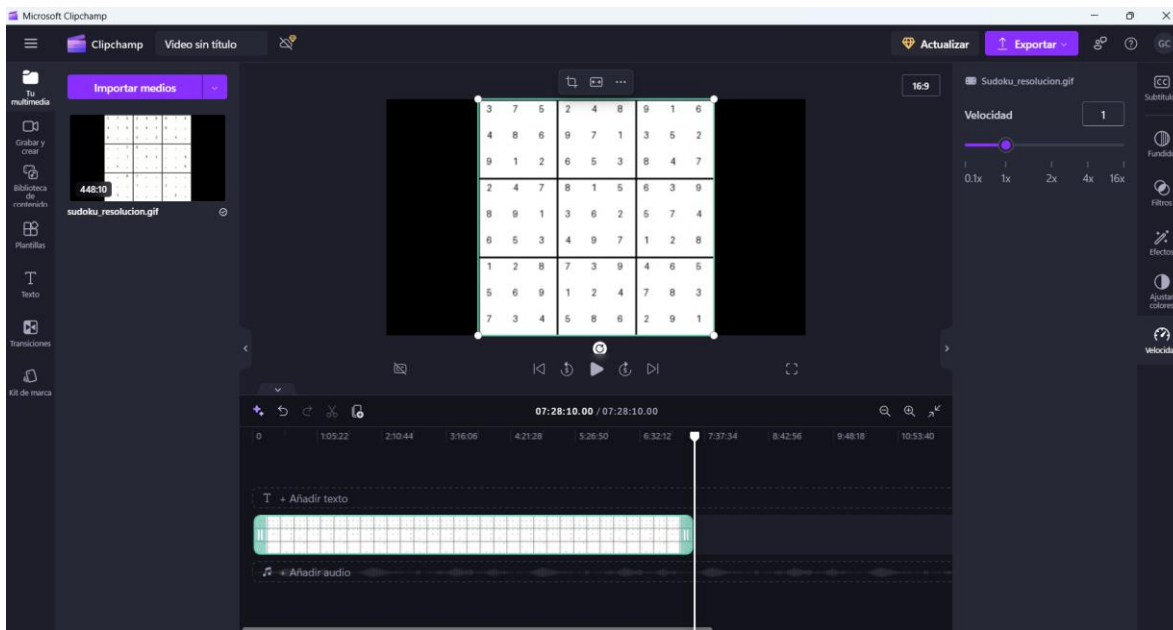
PS C:\Users\DELL> C:\Users\DELL\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/DELL/Desktop/Carpetas_codigo/Sudoku2.py"

Tablero inicial:

7	2	4	1	6	
1	3	5	8	9	7
9	6	3	4	2	1
7	8	1	2	3	5
1	6	2	5	4	7
5	4	7	3	6	2
8	7	9	1	2	4
2	1	3	6	5	8
3	5	2	7	8	9

Resolviendo Sudoku...

Tiempo de resolución: 114.0546 segundos



4. Comparación con Otras Técnicas

- Programación Dinámica: Aunque es eficiente en problemas que se pueden dividir en subproblemas independientes, el Sudoku no es adecuado para esta técnica debido a la interdependencia entre filas, columnas y subcuadros. Además, el

almacenamiento de subproblemas requiere una cantidad de memoria inabordable ($O(9^{81})$).

- **Divide y Vencerás:** Esta técnica no se adapta bien al Sudoku, ya que no es posible dividir el tablero en subproblemas independientes (como bloques de 3x3) sin violar las restricciones globales. Aunque podría usarse para resolver subconjuntos del problema, la validación cruzada entre subproblemas añadiría complejidad adicional.

Conclusión

El **backtracking** se destacó como la técnica más efectiva para resolver Sudoku debido a su balance entre facilidad de implementación y capacidad para manejar problemas con restricciones. Aunque su complejidad teórica es alta ($O(9^k)$), en la práctica, las validaciones implementadas lo hacen eficiente para tableros estándar. Técnicas como programación dinámica y divide y vencerás, aunque útiles en otros contextos, no se ajustan bien a las características del Sudoku.