Universidad del Caribe

2000 CANCUN, QUINTANA ROO, MÉXICO

CONOCIMIENTO Y CULTURA PARA EL DESARROLLO HUMANO

REPORTE:

Ejercicio final Sudoku

ASIGNATURA:

Técnicas algorítmicas

Eugenio Palomares Rodríguez

MATRÍCULA: **200300654**

PROGRAMA EDUCATIVO:

LIC/ING en datos e inteligencia organizacional

PRESENTADO A:

Prof. Emmanuel Saveedra

Cancún, Quintana Roo Nov, 2024

Justificación del trabajo

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.

Instrucciones Específicas:

1. **Elección de Técnica**: Evalúa las tres técnicas algorítmicas y elige la que consideres más efectiva para resolver el problema de Sudoku. Justifica tu elección en un reporte breve (1-2 párrafos), destacando las ventajas de tu técnica en términos de:

    - Complejidad computacional
    - Facilidad de implementación y adaptación al problema de Sudoku

2. **Implementación:**

- Implementa el algoritmo utilizando el lenguaje de programación de tu preferencia.

- Asegúrate de que el código esté optimizado y siga las buenas prácticas de codificación.

- Incluye comentarios en tu código que expliquen el proceso y las decisiones algorítmicas tomadas.

3. Evaluación y Análisis de Complejidad:

- Calcula y reporta la complejidad temporal y espacial de tu algoritmo en notación Big-O.

- Ejecuta el algoritmo e incluye el tiempo de ejecución para la solución del Sudoku.

Contenidos

Divide y vencerás

Merge Sort

Este algoritmo de ordenamiento divide el arreglo en dos mitades hasta que quedan subarreglos de tamaño 1, luego los va fusionando en orden ascendente para producir el arreglo ordenado. Es eficiente y tiene una complejidad de tiempo de O(nlogn)O(n \log n)O(nlogn).

Características de Merge Sort:

- Complejidad temporal: $O(n\log n)$O(nlogn) en el peor, mejor y promedio de los casos.
-
- Complejidad espacial: $O(n)$O(n) debido al uso de espacio adicional para las sublistas durante la fusión.

- Estable: Mantiene el orden relativo de los elementos iguales.

- No in-place: Utiliza memoria extra, lo que puede ser una limitación en grandes conjuntos de datos.

Quick Sort

Elige un elemento llamado pivote y divide el arreglo en dos partes: los elementos menores al pivote y los mayores. Luego aplica recursivamente el mismo proceso a las sublistas, lo que permite ordenar el arreglo de manera eficiente con una complejidad promedio de $O(n\log n)$O(nlogn).

Características de Quick Sort:

Complejidad temporal:

- **Promedio:** $O(n\log n)$O(nlogn)
- **Peor caso:** $O(n2)$O(n 2), si el pivote elegido es siempre el menor o mayor elemento (por ejemplo, si la lista ya está ordenada).
- **Mejor caso:** $O(n\log n)$O(nlogn) cuando el pivote divide el arreglo de manera equilibrada.

Complejidad espacial: $O(\log n)$O(logn) si se implementa de forma recursiva.

No es estable: El orden relativo de los elementos iguales puede cambiar.

In-place: No necesita memoria adicional significativa, ya que reordena los elementos en el mismo arreglo.

Búsqueda Binaria

Este algoritmo busca un valor en un arreglo ordenado dividiéndolo repetidamente a la mitad y descartando la mitad en la que el valor no puede estar. Tiene una complejidad de tiempo de $O(\log n)$O(logn), lo que lo hace muy eficiente para búsquedas en grandes conjuntos de datos.

Caracteristicas de la busqueda binaria

Condición esencial: Para poder aplicar la búsqueda binaria, el conjunto de datos debe estar ordenado previamente. Si la lista no está ordenada, el algoritmo no funcionará correctamente.

Complejidad temporal: $O(\log n)$O(logn), donde n
n es el número de elementos en la lista. Esto significa que el número de comparaciones crece logarítmicamente en función del tamaño de la lista, lo que lo hace muy eficiente para grandes conjuntos de datos.

Complejidad espacial:

- **Versión iterativa:** La versión iterativa de la búsqueda binaria tiene una complejidad espacial de $O(1)$O(1), lo que significa que no necesita memoria adicional significativa aparte del espacio para la lista original.

- **Versión recursiva:** La versión recursiva tiene una complejidad espacial de $O(\log n)$O(logn) debido al uso de la pila de recursión (stack de llamadas).

Backtracking

Es una técnica algorítmica para resolver problemas de toma de decisiones que requieren probar diferentes combinaciones posibles y "retroceder" cuando una opción no cumple con las condiciones del problema. Es especialmente útil en problemas donde se necesita generar soluciones paso a paso y verificar si cada paso conduce a una solución válida.

Recursividad:

El algoritmo suele ser implementado de forma recursiva, ya que va probando opciones y retrocede de manera natural cuando una combinación falla.

Complejidad temporal:

En el peor de los casos, el algoritmo puede tener una complejidad exponencial ($O(b^d)$O(b d)), donde b b es el número de opciones en cada paso y d d es la profundidad o número de decisiones a tomar.
Sin embargo, es más eficiente que una búsqueda exhaustiva (fuerza bruta), ya que evita explorar ramas que no llevarán a una solución válida.

Complejidad espacial:

La complejidad espacial depende de la profundidad del árbol de decisiones. En la versión recursiva, el uso de memoria es proporcional a la cantidad de decisiones tomadas en la rama más larga.

Comparativa (Tomando en cuenta su aplicación al sudoku)

Característica	Merge Sort	Quick Sort	Búsqueda Binaria	Backtracking
Aplicación	Ordenamiento	Ordenamiento	Búsqueda en listas ordenadas	Recursión y búsqueda con retroceso
Estrategia	Fusión	Partición	Búsqueda	Búsqueda de soluciones con restricciones
Complejidad temporal	O(nlogn)	O(nlogn) (promedio), O(n2)O(n^2)O(n2) (peor caso)	O(logn)	Exponencial en el peor caso: $O(bd)$O(b d), donde b b es el número de opciones por paso y d d es la profundidad de decisiones
Complejidad espacial	O(n)	O(logn) (in-place)	O(1) (iterativa), O(logn)O(\log n)O(logn) (recursiva)	Depende de la profundidad del árbol de decisiones: $O(d)$O(d)
Estabilidad	Sí (mantiene el orden relativo)	No necesariamente	No relevante	No relevante (depende del problema)
In-place	No	Sí	N/A (es un algoritmo de búsqueda)	No necesariamente (depende del problema)
Caso peor	O(nlogn)	O(n2) (si pivote malo)	O(logn)	Exponencial, $O(bd)$O(b d)

Requiere lista ordenada	No	No	Sí	No (depende del problema)
Eficiencia en grandes datos	Alta	Alta (excepto en peor caso)	Muy alta en datos ordenados	Baja en problemas grandes sin optimización
Uso común	Grandes volúmenes de datos, estabilidad importante	Datos grandes y desordenados, sin mucho uso de memoria	Búsqueda en datos ordenados	Problemas con restricciones y soluciones secuenciales

Cuál es el más eficiente para aplicarlo en un Sudoku:

- Backtracking es útil para resolver problemas donde hay restricciones que deben cumplirse y donde es necesario probar muchas combinaciones, como en el Sudoku o el problema de las n-reinas.

- Merge Sort y Quick Sort son buenos para ordenar grandes conjuntos de datos.

- Búsqueda Binaria es la mejor opción cuando se trata de buscar elementos en listas que ya están ordenadas.

De los 4 algoritmos vistos, se escogió el backtracking para tenerlo en cuenta y compararlo con los siguientes, se escogerá un representante de cada técnica algorítmica y al final se hará la comparativa entre cada representante para ver cual es el mejor para aplicarlo en el sudoku.

Programación dinámica

Fibonacci

El algoritmo de Fibonacci tiene varias implementaciones, y su estudio es importante tanto desde el punto de vista teórico como práctico. A continuación, te explico el algoritmo de Fibonacci, sus características y las formas de implementarlo.

1. Explicación del Algoritmo de Fibonacci
La sucesión de Fibonacci se define de la siguiente manera:

- F(0)=0
- F(1)=1
- Para n≥2n≥2, $F(n)$=$F(n-1)$+$F(n-2)$F(n)=F(n−1)+F(n−2)

Cada número de la sucesión es la suma de los dos anteriores. Esto da lugar a una forma natural de implementarlo de manera recursiva. Sin embargo, esta implementación tiene algunos problemas de eficiencia que luego se pueden mejorar con técnicas como la programación dinámica.

Fibonacci recursivo simple

La implementación recursiva sigue directamente la definición matemática

Características:

Simplicidad: Es fácil de entender y sigue directamente la definición matemática de Fibonacci.

Costo computacional elevado: Cada llamada recursiva genera dos más, lo que lleva a una complejidad temporal de O(2^n). Para valores grandes de n , esta implementación es muy ineficiente, porque recalcula muchas veces los mismos valores.

Recursividad profunda: A medida que n aumenta, el número de llamadas recursivas crece exponencialmente, lo que puede causar un desbordamiento de la pila en algunos lenguajes o entornos si n es muy grande.

Fibonacci con memoización (programación dinámica top-down)

Para mejorar la eficiencia, se puede usar memoización, que consiste en almacenar los resultados de las llamadas a subproblemas para no calcularlos nuevamente.

Características:

Complejidad reducida: Este enfoque reduce la complejidad temporal a O(n), ya que cada valor se calcula sólo una vez.

Eficiencia espacial: Necesita un almacenamiento extra para guardar los resultados intermedios, lo que introduce una complejidad espacial de O(n) debido al diccionario (o array) que guarda los valores ya calculados.

Recursividad limitada: Aunque sigue siendo recursivo, no recalcula los subproblemas más de una vez, lo que evita las explosiones de llamadas recursivas.

Fibonacci con tabulación (programación dinámica bottom-up)

La tabulación es otra técnica de programación dinámica en la que construimos una tabla (o arreglo) que almacena los resultados de los subproblemas de forma iterativa.

Características:

Complejidad temporal: Al igual que la memoización, la complejidad temporal es O(n).

Complejidad espacial: Aunque más simple, también tiene una complejidad espacial de O(n), ya que necesita almacenar los valores intermedios en un array.

No recursivo: Esta implementación es iterativa, por lo que evita la recursividad y los posibles problemas con el desbordamiento de la pila.

Knapsack

El problema Knapsack o de la mochila es un problema clásico de optimización combinatoria, donde se busca maximizar el valor total de los elementos seleccionados sin exceder una capacidad máxima determinada.

Para resolverlo con programación dinámica, usamos una matriz dp donde $dp[i][j]$ representa el valor máximo que se puede obtener con los primeros i elementos y una capacidad máxima de j.

Explicación:

- La matriz dp se rellena considerando si incluimos o no un elemento dado.
- Si el peso del elemento actual es mayor que la capacidad actual, no se incluye.
- Si no, decidimos entre incluirlo o no y tomamos el valor máximo.

Longest Common Subsequence (LCS)

El problema del Longest Common Subsequence (LCS) o Subsecuencia Común Más Larga es un clásico problema de programación dinámica. Dado dos secuencias (por ejemplo, cadenas de texto), el objetivo es encontrar la subsecuencia más larga que es común a ambas.

Ejemplo
Dadas dos cadenas:

X = "ABCBDAB"
Y = "BDCABA"

La subsecuencia común más larga es "BDAB" o "BCAB" (ambas de longitud 4).

Complejidad

Tiempo: $O(m \times n)$
Espacio: $O(m \times n)$

Optimización del Espacio
Se puede reducir el espacio a $O(n)$ si solo almacenamos dos filas de la matriz (la fila actual y la fila anterior), ya que solo necesitamos esa información para calcular la siguiente fila.

Aplicaciones del LCS

- **Detección de plagio:** Para comparar similitudes entre dos documentos.
- **Control de versiones:** Herramientas como diff usan LCS para identificar cambios en archivos de texto.
- **Bioinformática:** Comparación de secuencias de ADN.

Explicación aplicada al sudoku

Fibonacci:

¿Por qué no es aplicable?
El problema de Fibonacci se centra en calcular una serie numérica donde cada término es la suma de los dos anteriores. No hay ninguna estructura secuencial o cálculo numérico directo en la resolución de un Sudoku, ya que el Sudoku es un problema de restricciones lógicas y no sigue una serie secuencial.
El Sudoku requiere verificar restricciones en filas, columnas y subcuadrículas de 3x3, lo cual no se ajusta a la lógica de Fibonacci.

LCS (Longest Common Subsequence):

¿Por qué no es aplicable?
El LCS se enfoca en encontrar la subsecuencia común más larga entre dos cadenas de texto. Es un problema de análisis de secuencias, y el Sudoku no se basa en la comparación de cadenas.
El Sudoku implica rellenar un tablero con números del 1 al 9 cumpliendo ciertas restricciones, no comparar secuencias o buscar subsecuencias comunes.

Además, LCS se resuelve mediante una matriz de programación dinámica que analiza similitudes entre dos secuencias, pero el Sudoku trabaja con restricciones de validez para cada celda individual.

Knapsack (Mochila):
¿Por qué podría ser parcialmente aplicable?
El Knapsack se centra en maximizar el valor de una selección de elementos dentro de una capacidad limitada. Aunque no es exactamente lo que necesita un Sudoku, hay una analogía interesante: asignar números a las celdas del tablero bajo ciertas restricciones.
Sin embargo, la clave del Knapsack es maximizar el valor, mientras que en Sudoku, el objetivo es encontrar una asignación válida de números. No se trata de optimizar un valor, sino de cumplir restricciones en filas, columnas y subcuadrículas.
Para que Knapsack fuera aplicable, se necesitaría modificarlo para manejar un problema de asignación y validación de restricciones, lo que lo aleja de su propósito original.

Algoritmos voraces

Dijkstra

El algoritmo de Dijkstra es un método clásico en teoría de grafos para encontrar el camino más corto desde un vértice origen a todos los demás vértices en un grafo ponderado con pesos no negativos.

Características del algoritmo

- **Grafo aplicable:** Funciona con grafos dirigidos y no dirigidos, pero los pesos deben ser no negativos.
- **Determinista:** Siempre encuentra la solución óptima.
- **Estructura utilizada:** Comúnmente se implementa con una cola de prioridad (heap binario o Fibonacci heap) para mejorar su eficiencia.
- **Limitaciones:** No maneja grafos con pesos negativos (para esto se utiliza Bellman-Ford).

Complejidad temporal

Depende de la implementación de la estructura de datos:

- Usando una lista de adyacencia y sin cola de prioridad: O(V 2), donde V es el número de vértices.
- Usando una cola de prioridad con un heap binario: O((V+E)logV), donde E es el número de aristas.
- Usando una cola de prioridad con un Fibonacci heap: O(E+VlogV), lo cual es más eficiente en grafos densos.

Complejidad temporal

Depende de la implementación de la estructura de datos:

- Usando una lista de adyacencia y sin cola de prioridad: O(V 2), donde V es el número de vértices.
- Usando una cola de prioridad con un heap binario: O((V+E)logV), donde E es el número de aristas.
- Usando una cola de prioridad con un Fibonacci heap: O(E+VlogV), lo cual es más eficiente en grafos densos.

Kruskal

El algoritmo de Kruskal es un método eficiente para encontrar el árbol de expansión mínima (MST, Minimum Spanning Tree) en un grafo no dirigido y ponderado. Un árbol de expansión mínima conecta todos los vértices del grafo con el menor costo total sin formar ciclos.

Propiedades

Complejidad:

- **Ordenar las aristas:** O(ElogE), donde E es el número de aristas.
- **Operaciones de conjuntos disjuntos:** O(Eα(V)), donde α es la inversa de la función de Ackermann, muy lenta para valores prácticos.
- **Complejidad total:** O(ElogE), dado que logE≥α(V) para grafos típicos.

Ventajas:

- Fácil de implementar.
- Ideal para grafos dispersos (con pocas aristas).

Desventajas:

- No es tan eficiente como el algoritmo de Prim para grafos densos.

Prim

El algoritmo de Prim es otro método para encontrar el árbol de expansión mínima (MST, Minimum Spanning Tree) de un grafo no dirigido y ponderado. A diferencia del algoritmo de Kruskal, Prim crece un único árbol, añadiendo vértices al MST en cada paso.

Propiedades

Estructura de datos utilizada:
Se usa típicamente un montículo de prioridad (cola de prioridad) para seleccionar eficientemente la arista de menor peso.
Esto se implementa con una estructura de tipo heap en Python (heapq).

Complejidad:
O(ElogV), donde E es el número de aristas y V el número de vértices.

Comparativa

Caracteristica	Prim	Dijkstra	Kruskal
Propósito principal	Encontrar el árbol de expansión mínima (MST).	Encontrar la distancia más corta desde un nodo origen a todos los demás nodos.	Encontrar el árbol de expansión mínima (MST).
Tipo de problema	Árbol de expansión mínima (MST).	Caminos mínimos (Shortest Path).	Árbol de expansión mínima (MST).
Tipo de grafo	Grafo no dirigido y ponderado.	Grafo dirigido o no dirigido y ponderado (sin pesos negativos).	Grafo no dirigido y ponderado.
Estrategia	Crece un MST a partir de un nodo inicial.	Crece un conjunto de distancias mínimas a partir del nodo origen.	Selecciona aristas en orden de menor peso.
Estructura usada	Montículo de prioridad (heapq).	Montículo de prioridad (heapq).	Conjuntos disjuntos (Union-Find).
Complejidad temporal	O(ElogV)	O(ElogV) (usando heapq).	O(ElogE), donde ElogE≥ElogV.

Los algoritmos de Prim, Kruskal y Dijkstra no son adecuados para resolver un Sudoku directamente, ya que el problema del Sudoku no se modela como un grafo con árboles mínimos o caminos más cortos.

¿Por qué no Prim, Dijkstra o Kruskal?

Prim y Kruskal trabajan con grafos ponderados y buscan árboles de expansión mínima, lo cual no tiene relación directa con el Sudoku, ya que no hay un concepto de pesos o conexiones entre celdas.
Dijkstra se centra en encontrar caminos más cortos en un grafo, lo cual tampoco se aplica, ya que el Sudoku no tiene una representación natural como problema de caminos.

Elección técnica

Después de una larga comparativa entre cada tipo de programación y cada algoritmo, podemos concluir que el mejor algoritmo para poder resolver un sudoku es el algoritmo de Backtracking. A continuación su justificación:

Justificación del uso de Backtracking para resolver un Sudoku

El problema del Sudoku se clasifica como un problema de satisfacción de restricciones (CSP, por sus siglas en inglés), donde se deben llenar las celdas vacías con números del 1 al 9, cumpliendo con tres reglas fundamentales:

- **Regla de filas:** Cada número debe aparecer exactamente una vez por fila.
- **Regla de columnas:** Cada número debe aparecer exactamente una vez por columna.
- **Regla de subcuadrículas (3x3)**: Cada número debe aparecer exactamente una vez en cada subcuadrícula de tamaño 3x3.

Resolver un Sudoku implica encontrar una combinación válida de números que satisfaga estas restricciones. El algoritmo de Backtracking es particularmente adecuado para este tipo de problema debido a sus características y capacidades.

Complejidad computacional

- **Peor caso:** O(9 m), pero este caso es poco común en Sudokus razonables.
- **Caso práctico promedio:** O(k m), donde k es menor que 9, lo que hace que el algoritmo sea eficiente para tableros estándar de 9x9.
- **Tamaño fijo del tablero:** Dado que el Sudoku estándar tiene un tamaño fijo, el algoritmo funciona eficientemente en la mayoría de los casos prácticos.

A pesar de su complejidad teórica alta, el Backtracking es efectivo en resolver Sudokus debido al impacto significativo de las restricciones y posibles optimizaciones como heurísticas.

¿Por qué Backtracking es una buena elección?

1. **Exploración sistemática del espacio de soluciones:**
    - El algoritmo explora todas las posibles combinaciones de números para las celdas vacías, avanzando cuando una opción es válida y retrocediendo cuando se encuentra un conflicto.
    - Este enfoque garantiza que si existe una solución, el algoritmo la encontrará.

2. **Facilidad de implementación:**
   - El Backtracking es fácil de entender e implementar, lo que lo convierte en una opción ideal para problemas bien definidos como el Sudoku.

3. **Eficiencia razonable para tableros estándar (9x9):**

- Aunque en el peor de los casos el algoritmo tiene una complejidad exponencial O(9 m) (donde m es el número de celdas vacías), en tableros estándar, la combinación de reglas y restricciones reduce significativamente el espacio de búsqueda.

4. **Aplicación directa de restricciones:**
   - Las reglas del Sudoku (filas, columnas y subcuadrículas) se pueden implementar como funciones de validación. Esto permite al algoritmo descartar soluciones inválidas rápidamente, evitando recorrer caminos innecesarios.
5. **No requiere estructuras de datos avanzadas:**
   - A diferencia de otros métodos como Programación con Restricciones o Exact Cover, el Backtracking no necesita estructuras complejas ni conceptos matemáticos avanzados. Solo requiere un enfoque recursivo y una función para verificar la validez de cada intento.

Funcionamiento del Backtracking en el Sudoku

El algoritmo funciona de la siguiente manera:

1. **Buscar una celda vacía:**

- Se identifica la primera celda vacía en el tablero.

2. **Intentar colocar un número**

- Se prueba con números del 1 al 9.

3. **Verificar restricciones**

- Se valida si el número colocado cumple las reglas del Sudoku:
  - No se repite en la fila.
  - No se repite en la columna.
  - No se repite en la subcuadrícula 3x3.

4. **Avanzar o retroceder:**

- Si el número es válido, se procede a resolver la siguiente celda vacía recursivamente.
- Si no hay números válidos para una celda, el algoritmo retrocede y prueba otro número en la celda anterior (backtracking).

5. **Repetir hasta completar:**

- Este proceso continúa hasta que todas las celdas estén llenas con números válidos o se determine que no existe solución.

Ventajas de Backtracking en el Sudoku

Completo:
Garantiza encontrar una solución si existe, o confirmar que no hay solución válida.

Adaptable:
Puede modificarse fácilmente para tableros de diferentes tamaños o variantes del Sudoku (por ejemplo, Sudoku irregular).

Controlado por restricciones:
El algoritmo evita explorar caminos inválidos desde etapas tempranas gracias a las reglas del Sudoku.

Simplicidad:
No requiere herramientas avanzadas, siendo accesible para estudiantes y programadores principiantes.

Limitaciones y cómo manejarlas

1. **Complejidad en tableros grandes:**

En Sudokus de tamaños grandes o con muchas celdas vacías, el tiempo de ejecución puede aumentar significativamente. Esto se mitiga aplicando heurísticas, como elegir la celda con menos opciones válidas primero.

2. **No es paralelizable:**

El enfoque recursivo limita la paralelización, pero esto no es un problema crítico para tableros estándar.

Conclusión

El algoritmo de Backtracking es una solución natural para resolver Sudokus debido a su capacidad para explorar sistemáticamente el espacio de soluciones, validar restricciones en cada paso y garantizar la corrección de la solución encontrada. Aunque no es el más eficiente para tableros extremadamente grandes, su simplicidad y efectividad en tableros estándar lo convierten en una elección óptima para este tipo de problema.

Implementación

Para la ejecución del algoritmo yo escogí Python como lenguaje de programación para ejecutarlo, ya que es una excelente opción para implementar el algoritmo de backtracking en la resolución de un Sudoku debido a su sintaxis simple y legible, que permite escribir código claro y enfocado en la lógica del problema. Esto facilita tanto el desarrollo como la comprensión del algoritmo.

Además, Python cuenta con herramientas y bibliotecas que simplifican el manejo de datos y la optimización del programa. Su comunidad amplia y recursos abundantes también ofrecen soporte para resolver problemas y mejorar la implementación.

Código optimizado

Compartiré capturas acerca del codigo realizado, en el ya viene la implementación del algoritmo Backtracking al sudoku y su tiempo de ejecución

```python
#Eugenio Palomares Rodriguez 200300654
import time  # Importamos el módulo para medir el tiempo

# Estados para rastrear restricciones en filas, columnas y subcuadros 3x3
rows = [set() for _ in range(9)]  # Conjunto de números en cada fila
cols = [set() for _ in range(9)]  # Conjunto de números en cada columna
boxes = [set() for _ in range(9)]  # Conjunto de números en cada subcuadro 3x3
```

```python
def preprocess(grid):
    """
    Inicializa las restricciones basadas en la cuadrícula inicial.
    Llena las estructuras `rows`, `cols` y `boxes` con los números
    ya presentes en la cuadrícula.
    """
    for r in range(9):  # Recorre cada fila
        for c in range(9):  # Recorre cada columna
            num = grid[r][c]
            if num != 0:  # Si la celda no está vacía
                rows[r].add(num)  # Añade el número a las restricciones de la fila
                cols[c].add(num)  # Añade el número a las restricciones de la columna
                boxes[(r // 3) * 3 + (c // 3)].add(num)  # Añade al subcuadro correspondiente
```

```python
def is_valid_move_fast(row, col, num):
    """
    Verifica si es válido colocar 'num' en la posición (row, col) usando las restricciones precomputadas.
    Devuelve True si no hay conflictos en la fila, columna o subcuadro.
    """
    box_index = (row // 3) * 3 + (col // 3)  # Calcula el índice del subcuadro
    return num not in rows[row] and num not in cols[col] and num not in boxes[box_index]
```

```python
def find_empty_location_with_constraints(grid):
    """
    Encuentra la celda vacía con la menor cantidad de opciones válidas (más restringida).
    Esto optimiza el backtracking al reducir el número de caminos explorados.
    """
    min_options = 10  # Número máximo de opciones + 1
    best_cell = None  # Inicializa la mejor celda como None

    for r in range(9):  # Recorre todas las filas
        for c in range(9):  # Recorre todas las columnas
            if grid[r][c] == 0:  # Si la celda está vacía
                # Cuenta cuántos números del 1 al 9 son válidos para esta celda
                options = sum(1 for num in range(1, 10) if is_valid_move_fast(r, c, num))
                if options < min_options:  # Si tiene menos opciones, es más restringida
                    min_options = options
                    best_cell = (r, c)  # Actualiza la mejor celda

    return best_cell  # Devuelve la celda más restringida o None si no hay vacías
```

```python
def solve_sudoku_optimized(grid):
    """
    Resuelve el Sudoku usando restricciones precomputadas y priorización de celdas restringidas.
    Retorna True si encuentra una solución; False de lo contrario.
    """
    # Encuentra la celda vacía más restringida
    find = find_empty_location_with_constraints(grid)
    if not find:  # Si no hay celdas vacías, el Sudoku está resuelto
        return True
    row, col = find  # Desempaqueta las coordenadas de la celda más restringida

    for num in range(1, 10):  # Intenta colocar números del 1 al 9
        if is_valid_move_fast(row, col, num):  # Si el movimiento es válido
            grid[row][col] = num  # Coloca el número en la celda
            rows[row].add(num)  # Actualiza las restricciones para la fila
            cols[col].add(num)  # Actualiza las restricciones para la columna
            boxes[(row // 3) * 3 + (col // 3)].add(num)  # Actualiza el subcuadro

            if solve_sudoku_optimized(grid):  # Llama recursivamente al siguiente paso
                return True  # Si encuentra solución, termina

            # Si no encuentra solución, retrocede (backtracking)
            grid[row][col] = 0  # Vacía la celda
            rows[row].remove(num)  # Elimina el número de las restricciones de la fila
            cols[col].remove(num)  # Elimina el número de las restricciones de la columna
            boxes[(row // 3) * 3 + (col // 3)].remove(num)  # Elimina del subcuadro

    return False  # Si no se pudo colocar ningún número, regresa False
```

```python
def print_sudoku_pretty(grid):
    """
    Imprime el Sudoku de manera legible, con separadores entre subcuadros 3x3.
    """
    for i, row in enumerate(grid):
        if i % 3 == 0 and i != 0:  # Imprime una línea separadora entre bloques
            print("-" * 21)
        # Agrupa la fila en bloques de 3 y los imprime con separadores
        print(" | ".join(" ".join(str(cell) if cell != 0 else "." for cell in row[j:j+3]) for j in range(0, 9, 3)))
```

```python
# Ejemplo de Sudoku
grid = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

# Preprocesa las restricciones iniciales
preprocess(grid)

# Imprime el Sudoku inicial
print("Sudoku inicial:")
print_sudoku_pretty(grid)
```

```python
# Mide el tiempo de ejecución
start_time = time.time()  # Comienza a contar el tiempo

# Resuelve el Sudoku
if solve_sudoku_optimized(grid):
    print("\nSudoku resuelto:")
    print_sudoku_pretty(grid)
else:
    print("\nNo existe solución para este Sudoku.")

end_time = time.time()  # Termina de contar el tiempo

# Imprime el tiempo de ejecución
print(f"\nTiempo de ejecución: {end_time - start_time:.6f} segundos")
```

```
Sudoku inicial:
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

Sudoku resuelto:
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
---------------------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
---------------------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9

Tiempo de ejecución: 0.014183 segundos
```

Reporte final

Justificación de la Técnica Seleccionada
La técnica utilizada combina Backtracking con optimizaciones específicas:

1. **Precomputación de Restricciones:**
- Se usan estructuras auxiliares (rows, cols, y boxes) para almacenar los números ya presentes en filas, columnas y subcuadros. Esto evita búsquedas innecesarias y acelera la validación de movimientos.
2. **Priorización de Celdas Más Restringidas:**
- En lugar de procesar las celdas vacías en orden arbitrario, se selecciona la celda con el menor número de opciones válidas. Esto reduce drásticamente la exploración del espacio de soluciones.
3. **Retroceso (Backtracking):**
- Se aplica un enfoque recursivo para probar diferentes valores, retrocediendo si una decisión resulta en un callejón sin salida. Esto garantiza que todas las soluciones posibles sean exploradas eficientemente.

Ventajas de esta combinación:

- Minimiza la cantidad de celdas exploradas gracias a la priorización.
- Reduce la complejidad de validar movimientos mediante restricciones precomputadas.
- Es adecuada para resolver Sudokus con diferentes niveles de dificultad.

Análisis de la Complejidad Computacional

Complejidad en el Peor Caso:

- Para un Sudoku vacío (81 celdas) con un espacio de búsqueda de 9^{81} posibilidades, el algoritmo explora solo las configuraciones válidas debido a las optimizaciones.
- Validar un movimiento en O(1) gracias a las estructuras auxiliares.
- Seleccionar la celda más restringida tiene una complejidad aproximada de O(81×9), ya que se evalúan hasta 9 opciones por celda vacía.
- El backtracking en el peor caso podría tener una complejidad O(k n), donde k es el promedio de opciones válidas por celda y n es el número de celdas vacías.

Complejidad Promedio:

- Gracias a las restricciones precomputadas y la priorización, el número promedio de decisiones exploradas es significativamente menor. Para Sudokus típicos, esto reduce el espacio de búsqueda en órdenes de magnitud.
-

Optimización Principal:

- La técnica selecciona celdas con menos opciones posibles, lo que mejora el tiempo promedio al reducir ramas exploradas en el árbol de decisión.

Conclusión

La implementación optimizada del algoritmo de Backtracking demuestra ser eficiente para resolver Sudokus. La selección de celdas más restringidas y la validación con restricciones precomputadas mejoraron significativamente el rendimiento. El tiempo de ejecución muestra que esta solución es adecuada para resolver Sudokus de manera rápida y confiable, incluso en niveles de dificultad avanzados.

Bibliografía

- Méndez, M. (20 de abril de 2020). Algoritmos y Programación II Análisis de Algoritmos. Facultad de Ingeniería.
- "IT0107 - Técnicas Algorítmicas", secciones de: Programación Dinámica (Unidad IV), Divide y Vencerás (Unidad III), Algoritmos Voraces (Unidad II).