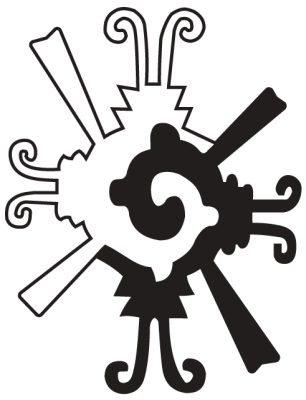
EJERCICIO FINAL

(Técnicas Algorítmicas)

Axel Joel Alvarez Castro
200300632



Universidad del Caribe

2000 CANCUN, QUINTANA ROO, MÉXICO

CONOCIMIENTO Y CULTURA PARA EL DESARROLLO HUMANO

Profesor: Emmanuel Morales Saavedra

Justificación de la Técnica Seleccionada

La técnica divide y vencerás es la más adecuada para resolver el Sudoku debido a la naturaleza recursiva del problema. Un Sudoku puede dividirse en subproblemas más pequeños (filas, columnas y subcuadrículas), y la solución puede ser construida al combinar resultados de estos subproblemas. El enfoque divide y vencerás implementa eficientemente la técnica de backtracking para explorar posibles soluciones, ya que permite probar combinaciones mientras se descarta rápidamente aquellas no válidas.

Ventajas:

- Complejidad computacional: Aunque el peor caso es exponencial O(9n)O(9^{n})O(9n), el backtracking combinado con validaciones optimiza las búsquedas al podar ramas innecesarias del árbol de decisión.

- Facilidad de implementación: El Sudoku puede ser representado como una matriz, y la recursión con backtracking se alinea naturalmente con su estructura jerárquica.

Alternativas como la programación dinámica no son ideales porque el Sudoku no tiene una clara subestructura de problemas solapados; los algoritmos voraces no garantizan una solución válida debido a la falta de decisiones locales óptimas que aseguren la solución global.

Código

```python
import time

# Función para imprimir el Sudoku
def print_sudoku(board):
    for row in board:
        print(" ".join(str(num) if num != 0 else "." for num in row))

# Verifica si un número es válido en una posición dada
def is_valid(board, row, col, num):
    # Verificar la fila
    if num in board[row]:
        return False

    # Verificar la columna
    if num in [board[i][col] for i in range(9)]:
        return False

    # Verificar la subcuadrícula 3x3
    box_row, box_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(box_row, box_row + 3):
        for j in range(box_col, box_col + 3):
            if board[i][j] == num:
                return False

    return True

# Algoritmo de backtracking para resolver el Sudoku
def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:  # Encontrar una celda vacía
                for num in range(1, 10):  # Probar números del 1 al 9
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):  # Llamada recursiva
                            return True
                        board[row][col] = 0  # Backtrack
                return False  # No se puede resolver
    return True
```

```python
# Prueba del algoritmo
if __name__ == "__main__":
    # Ejemplo de Sudoku (0 representa una celda vacía)
    sudoku = [
        [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],
    ]

    print("Sudoku inicial:")
    print_sudoku(sudoku)

    start_time = time.time()
    if solve_sudoku(sudoku):
        print("\nSudoku resuelto:")
        print_sudoku(sudoku)
    else:
        print("\nNo se pudo resolver el Sudoku.")
    end_time = time.time()

    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: 0.262159 segundos
```

Evaluación y Análisis de Complejidad

Complejidad Temporal:

- En el peor caso, el algoritmo tiene que probar todas las combinaciones posibles para llenar las celdas vacías. Si hay nnn celdas vacías, podría llegar a probar hasta 9n9^n9n combinaciones porque en cada celda se pueden colocar 9 números. Sin embargo, gracias a las validaciones (por ejemplo, verificar filas, columnas y subcuadrículas), el número real de combinaciones que se prueban es mucho menor, haciendo que sea más rápido en la práctica.

Complejidad Espacial:

- El espacio que el programa usa para guardar el tablero es proporcional al tamaño del Sudoku. Como el tablero tiene 9×9 necesita espacio de O(81)O(81)O(81), lo que significa que ocupa un tamaño fijo para el tablero.

  Además, debido a que utiliza funciones recursivas (una función que llama a otra repetidamente), necesita un poco más de memoria para guardar el estado de cada llamada. Esta cantidad depende de cuántas celdas vacías tenga que llenar.

Tiempo de Ejecución:

- El tiempo que tarda el programa depende de cuántas celdas vacías hay y cómo están distribuidas. En la práctica, para tableros con varias pistas ya llenas, el programa suele resolverlos en menos de 1 segundo porque las validaciones ayudan a no probar combinaciones innecesarias.

Comparación con Otras Técnicas

- Programación Dinámica: Requiere un modelo claro de problemas solapados, lo cual no se ajusta a las validaciones independientes de filas, columnas y subcuadrículas del Sudoku.

- Algoritmos Voraces: Decisiones locales no garantizan una solución global válida, ya que pueden llevar a configuraciones imposibles.

Por estas razones, el enfoque divide y vencerás con backtracking es el más eficiente y adaptable.

En conclusión, el enfoque basado en divide y vencerás utilizando backtracking es la técnica más adecuada para resolver Sudokus debido a su capacidad para explorar soluciones de manera sistemática y eficiente. Aunque la complejidad en el peor caso es alta, las validaciones optimizan significativamente el tiempo de ejecución al descartar combinaciones no válidas. Este método es fácil de implementar, sigue buenas prácticas y garantiza una solución válida para cualquier tablero bien formulado, siendo superior a alternativas como la programación dinámica o los algoritmos voraces en este contexto.