Sistemas Inteligentes

SUDOKU con solucionador

Tarea 9 extra

Alumno: Andres Soria Cabrera 421114409

Profesor: Rosas Hernández Javier

Grupo: 1754

Fecha: December 9, 2025

Facultad de Estudios Superiores Acatlán
Universidad Nacional Autónoma de México

Repositorio: `https://github.com/Andark11/SUDOKU`

Abstract

Este documento presenta la implementación de un solucionador de Sudoku interactivo que utiliza tres diferentes enfoques algorítmicos: Búsqueda en Profundidad (DFS), Búsqueda en Anchura (BFS) y Algoritmos Genéticos (GA). El sistema permite generar puzzles aleatorios con diferentes niveles de dificultad y compara el rendimiento de cada algoritmo en términos de tiempo de ejecución.

Contents

1 Introducción

El Sudoku es un puzzle lógico que consiste en llenar una cuadrícula de 9×9 dividida en subcuadrículas de 3×3 con dígitos del 1 al 9, cumpliendo las siguientes restricciones:

- Cada fila debe contener todos los dígitos del 1 al 9 sin repetición
- Cada columna debe contener todos los dígitos del 1 al 9 sin repetición
- Cada subcuadrícula de 3×3 debe contener todos los dígitos del 1 al 9 sin repetición

Este proyecto implementa tres enfoques distintos para resolver el problema del Sudoku, cada uno con características y rendimiento diferentes.

2 Características del Sistema

2.1 Funcionalidades Principales

- **Tres algoritmos de resolución**: DFS, BFS y Algoritmo Genético
- **Generación aleatoria**: Puzzles con 4 niveles de dificultad
- **Interfaz interactiva**: Línea de comandos fácil de usar
- **Medición de rendimiento**: Tiempo de ejecución para cada solución
- **Visualización clara**: Tablero con separadores visuales
- **Modo continuo**: Resolver múltiples puzzles sin reiniciar

2.2 Niveles de Dificultad

Nivel	Celdas Vacías
Fácil	30
Medio	40
Difícil	50
Extremo	60

Table 1: Niveles de dificultad disponibles

3 Algoritmos Implementados

3.1 DFS - Búsqueda en Profundidad

3.1.1 Descripción

El algoritmo DFS explora el espacio de búsqueda en profundidad, probando valores posibles y retrocediendo (backtracking) cuando encuentra un callejón sin salida.

3.1.2 Características

- Explora cada rama de posibilidades hasta el final
- Utiliza backtracking para volver atrás cuando encuentra conflictos
- Generalmente el más rápido para la mayoría de los Sudokus
- Complejidad temporal: $O(9^n)$ donde n es el número de celdas vacías

3.1.3 Pseudocódigo

```
function DFS(tablero):
    while not is_complete(tablero):
        tablero, movimientos, actualizado = poblar_movimiento_simple(tablero)

        if actualizado:
            actualizar_movimientos_posibles(tablero)
            continue

        tablero, movimientos, actualizado = poblar_movimiento_inteligente(tablero)

        if actualizado:
            actualizar_movimientos_posibles(tablero)
            continue

        if not is_valid(tablero):
            hacer_backtracking()
            continue

        # Adivinar siguiente movimiento
        seleccionar_celda_con_minimas_opciones()
        probar_primera_opcion()
        guardar_estado_en_historial()

    return tablero, True
```

3.2 BFS - Búsqueda en Anchura

3.2.1 Descripción

El algoritmo BFS explora el espacio de búsqueda nivel por nivel, expandiendo todos los estados posibles en cada nivel antes de continuar al siguiente.

3.2.2 Características

- Explora todas las posibilidades nivel por nivel
- Utiliza una cola (queue) para gestionar los estados
- Garantiza encontrar la solución más "corta" en términos de decisiones
- Complejidad espacial mayor que DFS debido al almacenamiento de estados

3.2.3 Pseudocódigo

```
function BFS(tablero):
    cola = nueva_cola()
    cola.agregar((tablero, movimientos_posibles))

    while cola no esta vacia:
        tablero_actual, movimientos = cola.extraer()

        # Aplicar movimientos obvios
        aplicar_estrategias_basicas(tablero_actual)

        if is_complete(tablero_actual):
            return tablero_actual, True

        if not is_valid(tablero_actual):
            continue

        # Generar nuevos estados
        for cada_opcion in movimientos:
            nuevo_tablero = copiar(tablero_actual)
            aplicar(nuevo_tablero, opcion)

            if is_valid(nuevo_tablero):
                cola.agregar((nuevo_tablero, nuevos_movimientos))

    return tablero_original, False
```

3.3 GA - Algoritmo Genético

3.3.1 Descripción

El algoritmo genético utiliza principios de evolución biológica para optimizar soluciones candidatas a través de generaciones sucesivas.

3.3.2 Componentes Principales

Representación

- **Cromosoma**: Un tablero completo de Sudoku 9×9
- **Gen**: Cada número en el tablero
- **Población**: Conjunto de tableros candidatos

Función de Fitness La función de fitness $f(x)$ se define como:

$$f(x) = 1000 \cdot C_{\text{fijos}} + C_{\text{filas}} + C_{\text{columnas}} + C_{\text{cajas}} \tag{1}$$

donde:

- C_{fijos} = Número de celdas fijas modificadas (penalización)

- $C_{\text{filas}} = \sum_{i=1}^{9}(9 - |\text{únicos en fila } i|)$
- $C_{\text{columnas}} = \sum_{j=1}^{9}(9 - |\text{únicos en columna } j|)$
- $C_{\text{cajas}} = \sum_{k=1}^{9}(9 - |\text{únicos en caja } k|)$

Operadores Genéticos Selección por Torneo: Se seleccionan aleatoriamente 5 individuos y se elige el mejor.

Cruce: Intercambio de filas completas entre dos padres:

```
function crossover(padre1, padre2, tablero_original):
    hijo = copia(padre1)

    for fila in 0..8:
        if random() < 0.5:
            hijo[fila] = copia(padre2[fila])

    return hijo
```

Mutación: Intercambio de dos valores en celdas no fijas de una fila:

```
function mutate(individuo, tablero_original, tasa_mutacion):
    for fila in 0..8:
        if random() < tasa_mutacion:
            columnas_mutables = [col donde tablero_original[fila][col] == 0]

            if len(columnas_mutables) >= 2:
                col1, col2 = seleccionar_aleatorio(columnas_mutables, 2)
                intercambiar(individuo[fila][col1], individuo[fila][col2])

    return individuo
```

3.3.3 Parámetros del Algoritmo

Parámetro	**Valor**
Tamaño de población	200
Generaciones máximas	5000
Tasa de mutación	0.4
Elitismo	5 individuos
Tamaño de torneo	5 individuos
Generaciones sin mejora (stop)	500

Table 2: Parámetros del algoritmo genético

3.3.4 Pseudocódigo

```
function GA(tablero):
    poblacion = inicializar_poblacion(200, tablero)
    mejor_fitness = infinito
```

```
sin_mejora = 0

for generacion in 1..5000:
    ordenar(poblacion, por=fitness)

    if poblacion[0].fitness == 0:
        return poblacion[0], True

    if poblacion[0].fitness < mejor_fitness:
        mejor_fitness = poblacion[0].fitness
        sin_mejora = 0
    else:
        sin_mejora += 1

    if sin_mejora >= 500:
        break

    nueva_poblacion = []

    # Elitismo
    agregar_mejores(nueva_poblacion, 5)

    # Generar descendencia
    while len(nueva_poblacion) < 200:
        padre1 = seleccion_torneo(poblacion)
        padre2 = seleccion_torneo(poblacion)
        hijo = crossover(padre1, padre2)
        hijo = mutate(hijo, 0.4)
        nueva_poblacion.agregar(hijo)

    poblacion = nueva_poblacion

return mejor_individuo(poblacion), fitness == 0
```

4 Estructura del Proyecto

4.1 Organización de Archivos

- `main.py` - Interfaz principal y generación de Sudokus
- `dfs.py` - Implementación del algoritmo DFS
- `bfs.py` - Implementación del algoritmo BFS
- `genetic_algorithm.py` - Implementación del Algoritmo Genético
- `posibles_movimientos.py` - Funciones para calcular movimientos válidos
- `populated_move.py` - Estrategia básica de llenado

- `populated_move_smart.py` - Estrategia inteligente de llenado
- `is_complete_board.py` - Verificación de tablero completo
- `dead_end.py` - Validación de estado del tablero

4.2 Estrategias de Resolución

El sistema utiliza múltiples estrategias complementarias:

1. **Movimiento único**: Coloca números cuando solo hay una opción válida en una celda
2. **Estrategia inteligente**: Encuentra números que solo pueden ir en una posición específica dentro de:
   - Filas
   - Columnas
   - Cajas 3×3
3. **Búsqueda con backtracking/evolución**: Prueba posibilidades cuando las estrategias básicas no son suficientes

5 Uso del Sistema

5.1 Instalación

No se requieren librerías externas. Solo Python 3.x:

```
python3 main.py
```

5.2 Flujo de Ejecución

1. Seleccionar algoritmo (DFS, BFS o GA)
2. Seleccionar nivel de dificultad (1-4)
3. El sistema genera un Sudoku aleatorio
4. El algoritmo resuelve el puzzle
5. Se muestra el tiempo de ejecución
6. Opción de resolver otro puzzle o salir

Algoritmo	Tiempo (Fácil)	Tiempo (Difícil)	Garantía
DFS	∼0.01s	∼0.05s	Alta
BFS	∼0.01s	∼0.10s	Alta
GA	∼0.87s	∼2.40s	Media

Table 3: Comparación de rendimiento de los algoritmos

6 Resultados y Análisis de Rendimiento

6.1 Comparación de Algoritmos

6.2 Ventajas y Desventajas

6.2.1 DFS

Ventajas:

- Muy rápido para la mayoría de casos
- Bajo consumo de memoria
- Implementación relativamente simple

Desventajas:

- Puede quedarse en ramas profundas infructuosas
- No garantiza la solución más corta

6.2.2 BFS

Ventajas:

- Garantiza solución óptima en términos de pasos
- Explora sistemáticamente

Desventajas:

- Mayor consumo de memoria
- Más lento que DFS en casos complejos

6.2.3 Algoritmo Genético

Ventajas:

- Enfoque completamente diferente (optimización vs búsqueda)
- Puede escapar de óptimos locales
- Paralelizable naturalmente
- Educativo para entender evolución artificial

Desventajas:

- Significativamente más lento
- No garantiza encontrar la solución
- Requiere ajuste de parámetros
- Comportamiento estocástico

7 Conclusiones

Este proyecto demuestra tres enfoques fundamentalmente diferentes para resolver el problema del Sudoku:

- **DFS y BFS** son algoritmos de búsqueda deterministas que exploran el espacio de soluciones de manera sistemática. Son eficientes y garantizan encontrar la solución si existe.
- **El Algoritmo Genético** representa un paradigma de optimización inspirado en la naturaleza. Aunque más lento, ilustra cómo problemas complejos pueden abordarse mediante evolución artificial.

El sistema permite experimentar y comparar estos enfoques en tiempo real, proporcionando una herramienta educativa valiosa para entender diferentes paradigmas de resolución de problemas.

7.1 Trabajo Futuro

- Implementar interfaz gráfica (GUI)
- Agregar más algoritmos (A*, Simulated Annealing)
- Optimizar el algoritmo genético con mejores operadores
- Implementar paralelización para GA
- Agregar estadísticas detalladas de rendimiento

8 Referencias

- Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson.
- Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems*. MIT Press.
- Cormen, T. H., et al. (2009). *Introduction to Algorithms*. MIT Press.

A Ejemplo de Ejecución

```
==================================================
=== SOLUCIONADOR DE SUDOKU ===
==================================================

Seleccione el algoritmo:
1. DFS (Depth-First Search - Busqueda en Profundidad)
2. BFS (Breadth-First Search - Busqueda en Anchura)
3. GA (Genetic Algorithm - Algoritmo Genetico)

Ingrese su opcion (1-3): 1

Seleccione el nivel de dificultad:
1. Facil (30 celdas vacias)
2. Medio (40 celdas vacias)
3. Dificil (50 celdas vacias)
4. Extremo (60 celdas vacias)

Ingrese su opcion (1-4): 2

Generando Sudoku nivel medio...

Tablero 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

Resolviendo con DFS...

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 ejecucion: 0.0149 segundos
```