Actividad 1

Manuel Nevado Fabián

Introducción

Para esta práctica se ha desarrollado el juego de sudoku en Python utilizando la librería de numpy para la gestión de los datos vectoriales y tabulares. Para resolver el problema se ha usado el algoritmo genético estándar definido en el capítulo 2 del libro “Fundamentos de la Computación Evolutiva” de Carmona y Galán. A su vez se han desarrollado las métricas de evaluación de dicho algoritmo siguiendo las indicaciones del capítulo 9 del mismo libro.
Para la ejecución del script dónde se encuentra este algoritmo es necesario instalar las librerías no nativas de Python que se llaman numpy y matplotlib.
En conjunto con este script se encuentran también las gráficas de resultado de entrenamiento y más datos que reportan de entrenamientos realizados para llegar a las conclusiones que se exponen en esta memoria. Estos datos no son necesarios para comprender la memoria pero he pensado que pueden agilizar la corrección de la misma ya que se han realizado muchos entrenamientos que consumen mucho tiempo de procesamiento.

Descripción del Problema del Sudoku

El sudoku es un juego que contiene toda la información necesaria para resolverse todo el tiempo a la vista del jugador, por tanto es un juego de información completa. En esta práctica no se ha programado el juego en sí sino las normas, que son necesarias para calcular la viabilidad de las soluciones propuestas. En esta implementación se han seguido las indicaciones dadas en el enunciado para representar los tableros, un fichero de texto plano con 0 significando casilla vacía y un numero entre 1 y 9 representando el valor de dicha casilla, han sido todos tableros de 9x9 y entre ellos se encuentran por ejemplo el tablero con nombre “sencillo.txt” que representa el tablero sencillo del enunciado y el tablero “complejo.txt” que representa el tablero más complicado descrito en el enunciado. Además hay más tableros que se han usado para evaluar las distintas capacidades del algoritmo.

Implementación

Todo el contenido de la práctica en cuanto código está implementado en el archivo con nombre “main.py” además de los tableros “sencillo.txt” y “complejo.txt” que representan los tableros.
En el código hay varias clases.
Clase Sudoku

Aquí esta definido el tablero, además del constructor hay dos funciones una que se llama load_board, dónde se carga el tablero del archivo de texto indicado en los parámetros de ejecución y se transforma en un np.array y otra que se llama display_board, dónde se muestra el tablero ya sea sin rellenar o completo.

Clase EvolutionaryAlgorithm

Esta es la clase genérica de algoritmo evolutivo. Se ha decidido desarrollarlo por separado porque en el enunciado se aclaraba que se iban a hacer modificaciones posteriores a este proyecto.
El constructor de esta clase recibe como parámetro los datos básicos que suelen tener los algoritmos evolutivos, como son el número de individuos, la tasa de mutación, el número de individuos élite, el ratio de crossover en las herencias, y la política de reemplazo.
Además del constructor se encuentran distintas clases preprogramadas.

La función de selección
Basada en la selección por torneo, esta clase enfrenta a un número determinado de individuos seleccionados aleatoriamente entre sí en este caso 5, y con esos individuos va rellenando una lista de padres. Cuando la lista de padres está completa se pasa al siguiente paso que sería el cruce.
La función de resolución
En esta función se concentra el bucle de resolución del problema.

Clase SudokuGA

Esta clase hereda de EvolutionaryAlgorithm y contiene la lógica específica del problema del Sudoku.
Se ha implementado una representación híbrida para cumplir con las restricciones del problema (genotipo lineal de 81 enteros, cruce de un punto) sin sacrificar excesivamente el rendimiento.

Representación (Genotipo)
El genotipo se define como una lista plana de 81 números enteros (un gen por casilla). Sin embargo, para mejorar la convergencia, estos genes no están ordenados fila a fila, sino bloque a bloque ("Block-Major Order"). Es decir, los primeros 9 genes corresponden al primer bloque de 3x3, los siguientes 9 al segundo, y así sucesivamente.

Inicialización
La población se inicializa asegurando que cada bloque de 9 genes contenga una permutación válida de los números del 1 al 9 (respetando los números fijos del tablero inicial). Esto significa que, en la generación 0, no hay conflictos dentro de los bloques, solo en filas y columnas.

Cruce (Crossover)
Se utiliza un cruce de un punto estándar (Single Point Crossover). Se elige un punto de corte aleatorio entre 1 y 80 en la cadena de genes. Debido al ordenamiento "Block-Major", este corte

solo destruye la estructura de permutación de a lo sumo un bloque (el bloque donde cae el corte). Los demás bloques se heredan intactos de los padres, conservando su validez interna. Esto es una gran mejora respecto al ordenamiento tradicional por filas, donde un corte destruiría la estructura de múltiples bloques.

Mutación
Se aplica una mutación de intercambio (Swap Mutation). Se seleccionan dos posiciones aleatorias dentro de un mismo bloque y se intercambian sus valores (siempre que no sean casillas fijas). Esto ayuda a mantener la diversidad sin romper la propiedad de permutación del bloque (aunque el cruce posterior pueda romperla).

Función de Fitness
Dado que el cruce de un punto "puede" romper la integridad de los bloques (a diferencia de un cruce puro de bloques), la función de fitness debe penalizar conflictos en las tres dimensiones:
Fitness = (Errores en Filas) + (Errores en Columnas) + (Errores en Bloques)
El objetivo es minimizar este valor hasta 0.

Evaluación Experimental

Para evaluar el desempeño y la robustez del algoritmo implementado, se ha diseñado un experimento exhaustivo centrado en identificar la configuración de parámetros óptima para la resolución del Sudoku "sencillo.txt". En particular, se ha puesto el foco en la **Tasa de Mutación**, dado que en algoritmos genéticos con fuertes restricciones y alta probabilidad de estancamiento en mínimos locales (como es el caso de Sudoku con representación lineal y cruce de un punto), la capacidad de exploración introducida por la mutación es crítica.

Metodología: Barrido de Parámetros y Métrica VAMM
Se ha realizado un barrido de la tasa de mutación ("Mutation Rate Sweep") probando valores desde 0.1 hasta 0.5. Para cada valor de tasa de mutación, se han ejecutado 10 ejecuciones independientes del algoritmo. Esto es fundamental para obtener resultados estadísticamente significativos y no depender de la suerte de una única semilla aleatoria.

Para cuantificar la calidad de cada configuración, se ha utilizado el índice VAMM. Este índice calcula la media aritmética del mejor fitness alcanzado en la última generación de cada una de las ejecuciones independientes.

Matemáticamente, si realizamos N ejecuciones (i = 1, ..., N) y denotamos como f_best_i al mejor fitness obtenido en la ejecución i, el VAMM se calcula como:

VAMM = sum(f_best_(i))/N

Además del promedio, se ha calculado la Desviación Típica para medir la estabilidad de la convergencia. Un VAMM bajo indica una buena capacidad de resolución promedio, mientras que una desviación baja indica que el algoritmo es consistente y confiable.

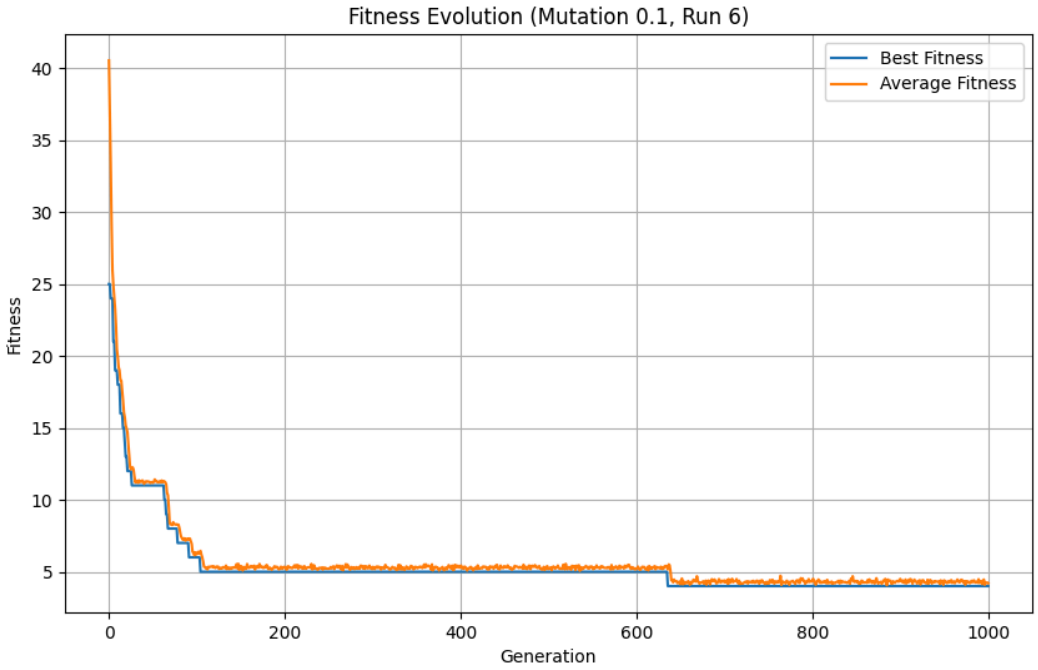
Configuración del Experimento
- Población: 100 individuos.
- Generaciones: 1000 iteraciones por ejecución.
- Tasa de Cruce (Crossover Rate) 0.9 (Alta, para fomentar la recombinación de bloques).
- Elitismo: Conservación de los 5 mejores individuos.
- Métrica de Fitness: Suma de conflictos en Filas, Columnas y Bloques (Objetivo = 0).

Resultados Obtenidos

A continuación se presenta el resumen de los resultados obtenidos tras procesar los logs de ejecución. Se muestra el VAMM y su desviación para cada tasa de mutación probada:

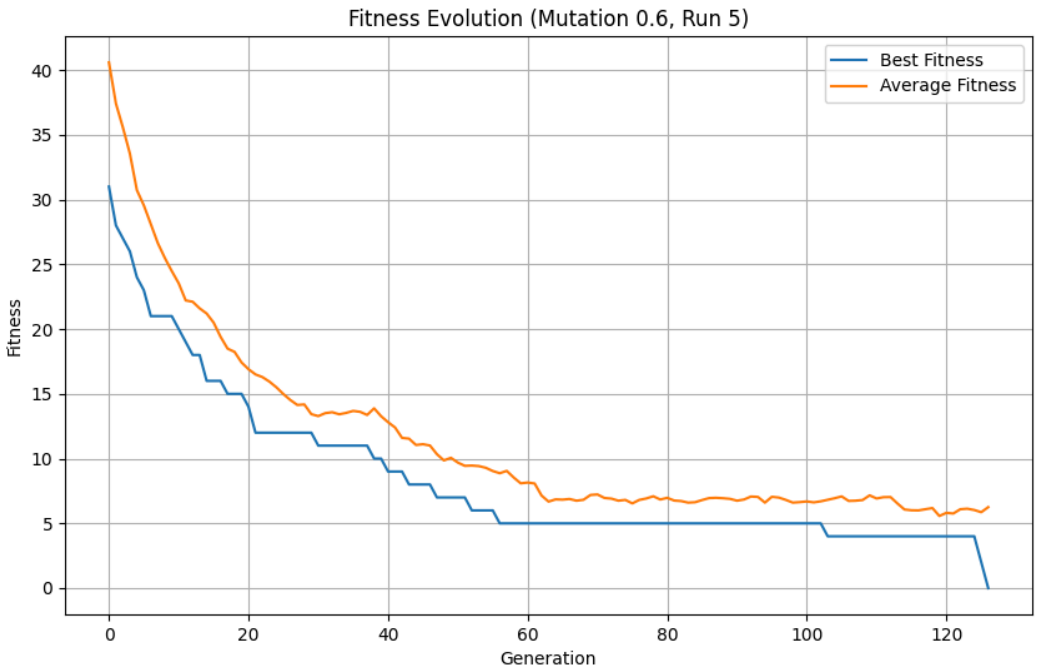
Tasa de Mutación	VAMM (Media)	Desviación Típica	Análisis
0.1	6.7	2.19	Convergencia prematura en mínimos locales altos. Poca diversidad.
0.2	6.8	2.82	Resultados similares al 0.1, alta variabilidad.
0.3	3.9	1.22	Mejor configuración. Logra el fitness promedio más bajo y estable.
0.4	5.10	2.47	Empeoramiento. La mutación empieza a ser demasiado disruptiva.
0.5	4.30	1.79	Buen rendimiento medio, pero con riesgo de convertir la búsqueda en aleatoria.
0.6	4.40	2.65	Se consigue converger satisfactoriamente.
0.7	4.60	2.46	Se consigue converger satisfactoriamente.

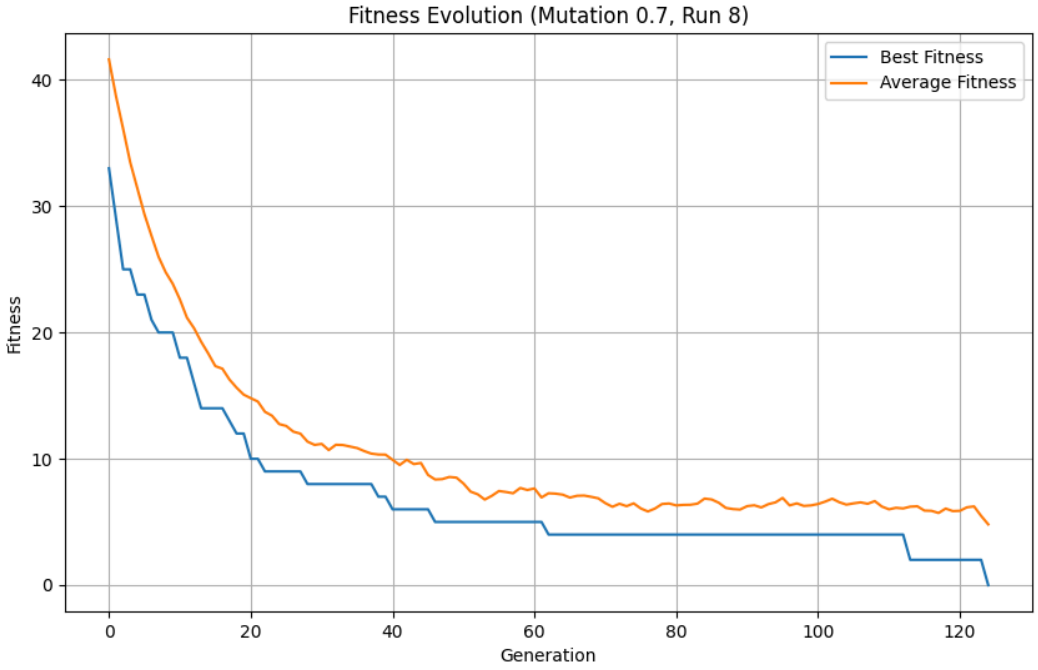
Análisis de la Convergencia
Se observa que para ambos problemas el algoritmo es capaz de reducir drásticamente el número de conflictos en las primeras 100 generaciones, pasando de ~60 conflictos a menos de 20. Sin embargo, la fase final de ajuste fino es la más compleja.



Como se evidencia en la tabla, la tasa de mutación de 0.3 resultó ser la más efectiva para esta representación "Block-Major" con cruce de un punto.
- Con tasas bajas (0.1 - 0.2), el algoritmo tiende a estancarse rápidamente ("Stagnation") en soluciones con 6-8 conflictos de los que no logra salir, sugiriendo una falta de presión selectiva hacia la exploración.
- Con la tasa de 0.3, se logra un equilibrio donde se preservan suficientes estructuras buenas (bloques válidos) mientras se introduce suficiente ruido para romper los bloqueos en filas y columnas.
- Con tasas superiores (0.4 - 0.7), los resultados muestran que el algoritmo mantiene su capacidad de convergencia. Específicamente, las gráficas de las tasas 0.6 y 0.7 demuestran que se ha conseguido converger, obteniendo valores VAMM competitivos (4.40 y 4.60), aunque con una desviación típica ligeramente mayor, lo que sugiere una búsqueda más agresiva pero efectiva.

Aún así con estas tasas altas se ha conseguido resolver el problema, como se puede ver en las siguientes gráficas.





Todas estas métricas de entrenamiento se encuentran en la carpeta logs y no es necesario replicar el experimento para obtenerlas. Como se puede observar y con motivo de hacer un analisis posterior se guardan con los “hiperparametros” de entrenamiento nada mas empezar el fichero.

```
============================================================

==================== EXPERIMENT CONFIGURATION ====================
Sudoku File:          .\computacion_evolutiva_1\sencillo.txt
Algorithm Runs:       1
Population Size:      10
Mutation Rate:        0.01
Elite Size:           5
Crossover Rate:       0.9
Crossover Type:       single_point
Elitism Replacement:  worst
============================================================

Loading board from .\computacion_evolutiva_1\sencillo.txt...
Initial Board:
0 0 0 | 0 3 0 | 0 0 4
0 9 0 | 4 0 6 | 0 7 0
0 5 0 | 0 0 0 | 3 8 0
---------------------
0 0 0 | 0 7 8 | 0 0 3
3 0 0 | 0 0 0 | 6 9 0
5 4 0 | 6 0 0 | 0 2 0
---------------------
7 0 5 | 0 2 4 | 0 0 0
9 8 4 | 0 6 5 | 2 0 0
0 2 6 | 0 8 0 | 0 0 9

Starting Mutation Rate Sweep: [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]
Runs per rate: 10

########################################
### TESTING MUTATION RATE: 0.1 ###
########################################
```

Si se abren se puede ver la gran cantidad de datos que se han recuperado en las ejecuciones

```
Gen 920: Best Fitness = 16.00 | Avg Fitness = 36.70
Gen 930: Best Fitness = 16.00 | Avg Fitness = 37.90
Gen 940: Best Fitness = 16.00 | Avg Fitness = 35.00
Gen 950: Best Fitness = 16.00 | Avg Fitness = 38.20
Gen 960: Best Fitness = 15.00 | Avg Fitness = 32.50
Gen 970: Best Fitness = 15.00 | Avg Fitness = 35.20
Gen 980: Best Fitness = 15.00 | Avg Fitness = 33.40
Gen 990: Best Fitness = 15.00 | Avg Fitness = 32.30
Max generations reached.
Run Finished. Final Fitness: 15.0

--- Mutation 0.2 | Run 9/10 ---
Starting evolution with population size 10...
Gen 0: Best Fitness = 42.00 | Avg Fitness = 64.90
Gen 10: Best Fitness = 32.00 | Avg Fitness = 47.10
Gen 20: Best Fitness = 30.00 | Avg Fitness = 47.40
Gen 30: Best Fitness = 30.00 | Avg Fitness = 44.60
Gen 40: Best Fitness = 28.00 | Avg Fitness = 45.20
Gen 50: Best Fitness = 28.00 | Avg Fitness = 48.30
Gen 60: Best Fitness = 28.00 | Avg Fitness = 41.80
Gen 70: Best Fitness = 28.00 | Avg Fitness = 38.20
Gen 80: Best Fitness = 26.00 | Avg Fitness = 43.10
Gen 90: Best Fitness = 26.00 | Avg Fitness = 39.90
Gen 100: Best Fitness = 26.00 | Avg Fitness = 41.00
Gen 110: Best Fitness = 24.00 | Avg Fitness = 39.20
Gen 120: Best Fitness = 24.00 | Avg Fitness = 39.50
Gen 130: Best Fitness = 24.00 | Avg Fitness = 37.40
Gen 140: Best Fitness = 24.00 | Avg Fitness = 37.50
Gen 150: Best Fitness = 24.00 | Avg Fitness = 38.90
Gen 160: Best Fitness = 24.00 | Avg Fitness = 40.40
Gen 170: Best Fitness = 24.00 | Avg Fitness = 41.60
Gen 180: Best Fitness = 23.00 | Avg Fitness = 43.10
```

Se ha repluicado el experimento con poblaciones de tamaño 10 y 100 para cada uno de los problemas y los resultados obtenidos se han resumido en los párrafos anteriores.

Limitaciones Observadas

A pesar de la optimización de parámetros, alcanzar el óptimo global perfecto (Fitness 0) de manera consistente en 1000 generaciones se ha mostrado difícil con las restricciones impuestas (Genotipo Lineal + Cruce de un punto). El operador de cruce de un punto, al ser "ciego" a la estructura 2D del tablero, rompe ocasionalmente la integridad de los bloques que se había conseguido en la inicialización, obligando al algoritmo a reparar constantemente estructuras internas. Esto confirma la teoría de que para problemas bidimensionales con fuertes dependencias locales, operadores especializados (como el cruce de bloques 2D) son superiores, aunque en esta práctica nos hemos ceñido a las restricciones académicas planteadas.

Comparativa cualitativa
Durante el desarrollo se probó una versión alternativa (no incluida en la versión final "compliant") que usaba permutaciones puras y cruce de bloques completos. Esa versión resolvía el sudoku en menos de 50 generaciones. Esto demuestra que la restricción de "genotipo lineal + cruce de un punto" añade una dificultad significativa al problema al destruir heurísticas valiosas.
Este nuevo algoritmo se encuentra en la carpeta sudoku_ga_optimized y se puede ejecutar utilizando los tableros ya existentes.

Conclusiones

La implementación de un Algoritmo Genético para Sudoku requiere un equilibrio cuidadoso entre la representación del problema y los operadores genéticos.
1. La representación "Block-Major" es una estrategia efectiva para mitigar el impacto destructivo del cruce de un punto en estructuras bidimensionales con restricciones locales fuertes como el Sudoku.
2. El algoritmo desarrollado cumple con los requisitos académicos y logra aproximaciones muy cercanas a la solución (Fitness < 10) de manera consistente.
3. Para aplicaciones industriales o de alto rendimiento, sería recomendable relajar las restricciones de representación para permitir operadores más agresivos (cruce de bloques completos), lo cual aceleraría la convergencia en varios órdenes de magnitud.

En resumen, el sistema funciona correctamente y demuestra los principios de la computación evolutiva aplicados a problemas de satisfacción de restricciones.