Tarea: Algoritmo Genético para Resolver Sudoku

Guía Paso a Paso para Abordar la Tarea

Esta guía te sugiere un orden para entender y completar la asignación:

1. **Leer la Asignación:** Lee este documento (`02_ga_hw.md`) completamente para entender los objetivos y requisitos.
2. **Repasar la Teoría:** Revisa los conceptos de Algoritmos Genéticos vistos en clase y en el material `01_ga.md` y la introducción a optimización en `00_intro.md`.
3. **Entender el Problema:** Asegúrate de comprender bien las reglas del Sudoku y qué constituye una solución válida.
4. **Explorar el Código:**
    - Revisa cada uno de los archivos Python (`environment.py`, `evaluation.py`, `visualization.py`, `solution.py`).
    - Presta especial atención a `solution.py`. Identifica las funciones de aptitud (`fitness_function_*`) y las clases de algoritmos genéticos (`GeneticAlgorithmVariant*`).
    - Comprende cómo se conectan los archivos (qué funciones/clases se importan y usan).
5. **Ejecutar el Ejemplo:** Corre `python solution.py` con la configuración por defecto para ver cómo funciona, qué resultados produce y cómo se visualiza el proceso.
6. **Identificar Tareas:** Localiza en `solution.py` las secciones específicas que debes modificar o implementar:
    - Las funciones `fitness_function_2_student` y `fitness_function_3_student`.
    - Las clases `GeneticAlgorithmVariant*` (puedes modificarlas o crear nuevas clases).
7. **Diseñar Funciones de Aptitud:**
    - Piensa: ¿Cómo se puede medir numéricamente qué tan "bueno" es un tablero de Sudoku que *no* está resuelto? ¿Qué características indican que está "cerca" de la solución?
    - Considera usar las funciones de `evaluation.py` (errores, celdas vacías, etc.) como componentes de tu cálculo de aptitud.
    - Diseña al menos dos enfoques diferentes para calcular la aptitud.
8. **Diseñar Variantes de GA:**
    - Piensa: ¿Cómo podrías cambiar la forma en que se seleccionan los padres (`_selection`)? ¿O cómo se combinan (`_crossover`)? ¿O cómo se introducen pequeños cambios (`_mutation`)?
    - Considera diferentes operadores genéticos (Ruleta, Cruce de N puntos, Mutación por Intercambio, etc.) o estrategias (diferentes parámetros, inicialización heurística).
    - Diseña al menos dos variantes algorítmicas (modificando las existentes o creando nuevas clases).
9. **Implementar Gradualmente:** No intentes implementar todo a la vez.
    - Empieza por implementar *una* nueva función de aptitud o *una* modificación a un método del GA.
10. **Probar Frecuentemente:** Después de cada cambio significativo:

PROF

    - Selecciona tu nueva implementación en `solution.py` (modificando `selected_ga_name` o `selected_fitness_name`).
    - Ejecuta el código y observa si funciona como esperas. Analiza los resultados y la gráfica de convergencia.
11. **Comparar y Analizar:** Una vez que tengas varias funciones de aptitud y variantes de GA funcionales, compáralas ejecutando diferentes combinaciones y analizando los resultados según los puntos de la sección de Evaluación.

---

1. Objetivo

El objetivo de esta tarea es implementar y experimentar con Algoritmos Genéticos (GAs) para resolver puzzles Sudoku. Deberás diseñar y comparar diferentes funciones de aptitud (fitness functions) y variantes de algoritmos genéticos para comprender cómo afectan el rendimiento y la capacidad de convergencia de la solución.

2. Descripción del Código Proporcionado

Se te proporciona una base de código en Python estructurada en varios archivos dentro del directorio `curso/temas/16_optimizacion/`:

- **`environment.py`**: Define la clase `SudokuEnvironment`.
    - **`__init__(self, size=9)`**: Inicializa el entorno con un tamaño `size` (por defecto 9). Genera internamente una solución completa.
    - **`generate_puzzle(self, difficulty=0.5)`**: Genera y devuelve un tablero (`np.ndarray`) con el puzzle inicial, removiendo celdas según la `difficulty` (0 a 1).
    - **`get_initial_board(self)`**: Devuelve una copia del tablero (`np.ndarray`) del puzzle inicial generado.
    - **`get_solution(self)`**: Devuelve una copia del tablero (`np.ndarray`) con la solución completa (para referencia).
    - **`get_size(self)`**: Devuelve el tamaño N del Sudoku (ej. 9).
    - **`is_solved(self, board)`**: Verifica si el `board` (`np.ndarray`) proporcionado es una solución completa y válida. Devuelve `True` o `False`.
- **`evaluation.py`**: Contiene funciones auxiliares para evaluar tableros. Todas toman un `board` (`np.ndarray`) como entrada principal:
    - `calculate_row_errors(board)`: Devuelve el número total de errores (duplicados) en todas las filas.
    - `calculate_col_errors(board)`: Devuelve el número total de errores en todas las columnas.
    - `calculate_subgrid_errors(board)`: Devuelve el número total de errores en todas las subcuadrículas.
    - `count_empty_cells(board)`: Devuelve el número de celdas con valor 0.
    - `check_initial_clues(candidate_board, initial_board)`: Toma el tablero candidato y el tablero inicial. Devuelve el número de pistas iniciales que fueron modificadas incorrectamente en el candidato.
- **`visualization.py`**:

PROF

    - `plot_sudoku_board(board, initial_board=None, title="Sudoku Board", pause_time=0.1)`: Dibuja el `board` (`np.ndarray`) actual. Usa `initial_board` para resaltar pistas. Muestra `title` y pausa por `pause_time`.
    - `plot_convergence(fitness_history, title="GA Convergence")`: Grafica la lista `fitness_history` (mejor aptitud por generación).
    - `close_plot()`: Cierra las ventanas de visualización.
- `solution.py`: **Este es el archivo principal que deberás modificar.** Contiene:
    - Configuración global: `SUDOKU_SIZE`, `PUZZLE_DIFFICULTY`.
    - **Funciones de Aptitud (Fitness Functions)**:
        - Formato esperado: `def mi_funcion_fitness(candidate_board: np.ndarray, initial_board: np.ndarray, env: SudokuEnvironment) -> float:`
        - **Inputs**: El tablero candidato (`candidate_board`), el tablero inicial (`initial_board`), y el objeto del entorno (`env`).
        - **Output**: Un número flotante (`float`) que representa la aptitud (mayor es mejor).
        - Ejemplo provisto: `fitness_function_1_binary`.
        - Placeholders: `fitness_function_2_student`, `fitness_function_3_student` (devuelven 0.0 por defecto).
    - **Clases de Algoritmos Genéticos (GA Classes)**:
        - Ejemplos provistos: `GeneticAlgorithmVariant1_NaiveStdParams`, `GeneticAlgorithmVariant2_HighMutation`, `GeneticAlgorithmVariant3_LowCXNoElite`.
        - **`__init__(self, env: SudokuEnvironment, fitness_func)`**:
            - **Inputs**: El objeto del entorno (`env`) y la función de aptitud seleccionada (`fitness_func`).
            - **Acción**: Inicializa el algoritmo, incluyendo la definición de parámetros *internos* (tamaño de población, tasas, etc.) y la creación de la población inicial llamando a `_initialize_population`.
        - **`_initialize_population(self)`**:
            - **Inputs**: `self` (acceso a `self.pop_size`, `self.initial_board`, etc.).
            - **Output**: Una lista (`list`) de individuos (tableros `np.ndarray`), representando la población inicial.
        - **`_selection(self, fitness_scores: list)`**:
            - **Inputs**: `self` y una lista (`fitness_scores`) con la aptitud de cada individuo de la población actual.
            - **Output**: Una lista (`list`) de individuos seleccionados (padres) para la reproducción.
        - **`_crossover(self, parent1: np.ndarray, parent2: np.ndarray)`**:
            - **Inputs**: `self` y dos individuos padres (`parent1`, `parent2`).
            - **Output**: Una tupla con dos nuevos individuos hijos (`child1: np.ndarray`, `child2: np.ndarray`) creados a partir de los padres.
        - **`_mutation(self, individual: np.ndarray)`**:
            - **Inputs**: `self` y un individuo (`individual`).
            - **Output**: El individuo potencialmente modificado (`np.ndarray`) tras aplicar la mutación.

        - `run(self, max_generations: int, visualize=True)`:
            - **Inputs**: `self`, el número máximo de generaciones a ejecutar (`max_generations`), y un booleano `visualize` para activar/desactivar la visualización en tiempo real.
            - **Output**: Una tupla conteniendo el mejor tablero encontrado (`best_solution: np.ndarray` o `None`) y una lista con el historial de la mejor aptitud por generación (`best_fitness_history: list`).
    - Diccionarios (`FITNESS_FUNCTIONS`, `GA_VARIANTS`) para seleccionar implementaciones por nombre.
    - Bloque principal (`if __name__ == "__main__":`) que configura el entorno, selecciona (usando `selected_ga_name`, `selected_fitness_name`), instancia y ejecuta la combinación elegida.
- `requirements.txt`: Lista las dependencias (`numpy`, `matplotlib`).

3. Tarea a Realizar

Tu tarea consiste en explorar diferentes estrategias para resolver el Sudoku usando GAs. Debes implementar lo siguiente dentro de `solution.py`:

1. **Nuevas Funciones de Aptitud:**
    - Implementa la lógica para `fitness_function_2_student` y `fitness_function_3_student` siguiendo el formato `def func(candidate_board, initial_board, env) -> float`.
    - Diseña métricas que evalúen qué tan bueno es un `candidate_board` comparado con el `initial_board` y las reglas del Sudoku (puedes usar `env.is_solved` o las funciones de `evaluation.py`). Devuelve un `float` donde mayor sea mejor.
2. **Nuevas Variantes de Algoritmos Genéticos:**
    - Modifica las clases GA existentes o crea nuevas clases. Si creas nuevas, asegúrate de que tengan el método `__init__(self, env, fitness_func)` y `run(self, max_generations, visualize=True)`.
    - Implementa o modifica los métodos internos (`_initialize_population`, `_selection`, `_crossover`, `_mutation`) con diferentes estrategias. Define los parámetros necesarios dentro de `__init__`.
    - El objetivo es tener al menos **dos variantes funcionales adicionales** que puedas comparar.

PROF

4. Cómo Ejecutar y Experimentar

1. **Instalar Dependencias:** Asegúrate de tener Python instalado. Abre una terminal en el directorio `curso/temas/16_optimizacion/` y ejecuta:

```
pip install -r requirements.txt
```

2. **Seleccionar Configuración:** Abre `solution.py`. En la sección `Experimentation Setup` (cerca del final del archivo), modifica las variables:
    - `selected_ga_name`: Elige el nombre de la clase del GA que quieres ejecutar (debe coincidir con una clave del diccionario `GA_VARIANTS`).
    - `selected_fitness_name`: Elige el nombre de la función de aptitud que quieres usar (debe coincidir con una clave del diccionario `FITNESS_FUNCTIONS`).
3. **Ajustar Generaciones:** Modifica la variable `MAX_GENERATIONS` para controlar cuánto tiempo (en generaciones) se ejecuta el algoritmo.
4. **Ejecutar:** Desde la terminal, en el mismo directorio, ejecuta el script:

    ```
    python solution.py
    ```

5. **Observar:** El script mostrará el puzzle inicial. Luego, si la visualización está activada (`visualize=True` en la llamada a `run`), verás cómo evoluciona la mejor solución encontrada en cada generación (con errores en rojo). Finalmente, mostrará la mejor solución final, indicará si es válida y completa, y mostrará la gráfica de convergencia de la aptitud. Cierra las ventanas de gráficos para que el script termine completamente.
6. **Experimentar:** Prueba diferentes combinaciones de tus funciones de aptitud y variantes de GA. Modifica los parámetros internos de tus clases de GA. Observa cómo cambian los resultados: ¿Se resuelve el Sudoku? ¿En cuántas generaciones? ¿Cómo es la curva de convergencia?

5. Evaluación

El objetivo principal es la experimentación y la comprensión. Compara tus implementaciones:

- ¿Qué función de aptitud parece guiar mejor al algoritmo hacia la solución?
- ¿Qué variante del GA (con qué operadores/parámetros) converge más rápido o encuentra mejores soluciones?
- ¿Cómo afecta el número de generaciones (`MAX_GENERATIONS`)?
- ¿Funciona el enfoque para diferentes dificultades de puzzle (`PUZZLE_DIFFICULTY`)?

PROF

No necesariamente tienes que encontrar una configuración que resuelva todos los Sudokus rápidamente, sino entender los tradeoffs y el impacto de tus decisiones de diseño en el comportamiento del algoritmo genético.

---

6. Pregunta Adicional para Reflexionar: Hacia un Solucionador General de Sudoku

La tarea actual se enfoca en resolver un puzzle específico generado al inicio de cada ejecución de `solution.py`. Ahora, reflexiona sobre el siguiente desafío:

Pregunta Central: ¿Cómo modificarías o rediseñarías el enfoque basado en Algoritmos Genéticos para crear un programa que pueda recibir *cualquier* puzzle Sudoku válido como entrada y intentar resolverlo?

Guía para la Reflexión (no necesitas implementar esto, solo pensarlo):

- **Retos Principales:** ¿Cuáles serían las mayores dificultades al pasar de resolver un puzzle *generado internamente* a resolver un puzzle *arbitrario* proporcionado como entrada?
- **Entrada del Sistema:** ¿Cómo cambiaría la forma en que el programa recibe el puzzle a resolver? Ya no usaría `environment.generate_puzzle()`. ¿Cómo leería un puzzle externo?
- **Manejo de Pistas Iniciales:** El código actual respeta las pistas iniciales del puzzle generado. ¿Cómo te asegurarías de que tu GA general respete *siempre* las pistas fijas de *cualquier* puzzle de entrada?
- **Funciones de Aptitud:** ¿Las funciones de aptitud que diseñaste (o la binaria) seguirían siendo válidas? ¿Necesitarían adaptarse para funcionar bien con puzzles de diferentes niveles de dificultad o con pocas/muchas pistas iniciales?
- **Representación:** ¿La representación actual del individuo (un tablero `np.ndarray` que evoluciona) es adecuada para un solver general? ¿Hay alternativas?
- **Parámetros del GA:** ¿Deberían los parámetros del GA (tamaño de población, tasas, etc.) ser fijos, o podrían/deberían ajustarse dinámicamente según las características del puzzle de entrada (ej. número de pistas, dificultad estimada)?
- **Robustez y Generalización:** ¿Qué significa que un GA sea "robusto" en este contexto? ¿Cómo podrías medir si tu enfoque generaliza bien a diferentes tipos de puzzles Sudoku (fáciles, difíciles, con patrones específicos)?
- **Pruebas:** Si construyeras este solver general, ¿cómo lo probarías exhaustivamente? ¿Qué colección de puzzles Sudoku usarías para validar su efectividad?

Considerar estas preguntas te ayudará a profundizar en las capacidades y limitaciones de los Algoritmos Genéticos y en los desafíos de aplicar técnicas de optimización a problemas más generales.