Universidad Nacional Autónoma de México

Facultad de Ciencias

Cómputo Evolutivo - 7140

T A R E A 3

Equipo:

Castillo Hernández Antonio - 320017438

☎☎☎

Luna Campos Emiliano - 320292084

☎☎☎

Juárez Cruz Joshua - 320124516

☎☎☎

[0.5cm]

Fecha de entrega:
25 de septiembre del 2025

Profesor:
Oscar Hernández Constantino

Ayudantes:
Rodrigo Fernando Velázquez Cruz
María Adelina Torres Ochoa

Lectura de Ejemplares

a) Describe una versión generalizada para el problema del Sudoku.

i) Enuncia el problema de optimización continua.

Sea n un valor entero con raíz cuadrada exacta, definimos un ejemplar del problema del sudoku como una matriz X de tamaño $n \times n$, donde X_{ij} representa el valor en la celda (i, j). Sí $X_{ij} = 0$ la celda está vacía, de otra forma $X_{ij} \in \{1, \ldots, n\}$. Así, algunas celdas tienen valores fijos (dados en el problema inicial), los cuales no pueden modificarse.

Función objetivo: Sea X' una solución para el problema definido para la matriz X, buscamos minimizar $f(X')$ definida de la siguiente manera:

$$f(X') = C\_filas(X') + C\_columnas(X) + C_bloques(X)$$

donde:

- $C\_filas(X')$ es el número de colisiones en las filas:

$$C\_filas(X') = \sum_{i=1}^{n} n - |\{X_{ij} : j = \{1, \ldots, n\}\}|$$

- $C\_columnas(X')$ es el número de colisiones en las columnas:

$$C\_columnas(X') = \sum_{j=1}^{n} n - |\{X_{ij} : i = \{1, \ldots, n\}\}|$$

- $C\_bloques(X')$ es el número de colisiones en los bloques:

$$C\_bloques(X') = \sum_{B} n - |\{X_{ij} \ : \ (i, j) \in B\}|$$

  B es un bloque del sudoku.

Así, queremos encontrar X tal que $f(X) = 0$, lo que indica una solución válida del Sudoku (sin colisiones).

ii) Describe un formato para ejemplares de una dimensión arbitraria (no solo sudokus de 9x9).

En general, la representación de un Sudoku en una matriz X de dimensión $n \times n$ es como sigue:

- n debe ser un cuadrado perfecto ($n = k^2$, donde $k \in \mathbb{N}$).
- Cada bloque B es de dimensiones $k \times k$.
- Cada celda (variable de decisión) tiene como valores permitidos $\{1, 2, \ldots, n\}$.

b) Implementa un programa que permita la lectura de los archivos con la información de prueba, y cargue los datos en alguna estructura que sea eficiente.

El código completo que implementa la estructura de un ejemplar del problema del sudoku y la lectura de archivos se realizo en python y es el siguiente:

```
class Sudoku:
    def __init__(self, grid):
        self.grid = np.array(grid, dtype=int)
        self.size = self.grid.shape[0]
        self.block_size = int(math.sqrt(self.size))
        self.fixed_cells = (self.grid != 0)

    @classmethod
    def from_file(cls, filename):
        file = open(filename,'r')
        lines = [line.strip() for line in file.readlines() if line.strip()]

        grid = []
        for line in lines:
            row = [int(x) for x in line.split()]
            grid.append(row)

        return cls(grid)
```

Representación de Soluciones

a) Describe e implementa un esquema de representación de soluciones para el problema generalizado del sudoku.

El esquema debería permitir representar soluciones para cualquier dimensión, no solamente de 9x9.

Se deben especificar las características que tiene la representación (tipo de mapeo, tipo de representación, clasificación, etc.), de acuerdo a las propiedades comentadas en clase.

Intentamos separar la representación de una solución de la representación del ejemplar del problema. Decimos que el espacio del problema del sudoku son todas la matrices $n \times n$ que cumplen:

- n es un cuadrado perfecto.
- Cada celda puede contener un valor fijo del conjunto $\{1, 2, ..., n\}$. De otra forma es 0 y puede ser considerada en la representación de la solución.

Así, queremos que el espacio de búsqueda para las soluciones sea solo las asignaciones de valores a celdas vacías (con valor igual a 0). Por lo que usamos la siguiente representación:

- Estructura: Vector de enteros.
- Tamaño: k elementos, donde k es el número de celdas vacías.
- Dominio: Cada elemento x es tal que $x \in \{1, 2, ..., n\}$

Así, dado un vector con las k posiciones de las celdas vacías, digamos P:

$$P = [(r_1, c_1),\ (r_2, c_2),\ \ldots,\ (r_k, c_k)]$$

El vector con los valores de la solución sera S:

$$S = [v_1,\ v_2,\ \ldots,\ v_k]$$

De manera que, $S[i] = v_i$ tiene la posición $P[i] = (r_i, c_i)$. Así, decimos que una solución se compone de los arreglos P y S.

El código con la implementación de la representación de una solución es el siguiente:

```
class SudokuSolution:
    def __init__(self, problem):
        self.problem = problem
        self.empty_positions = []
        for i in range(problem.size):
            for j in range(problem.size):
                if not problem.fixed_cells[i, j]:
                    self.empty_positions.append((i, j))
        self.num_empty = len(self.empty_positions)
        self.position_to_index = {pos: idx for idx, pos in
                                  enumerate(self.empty_positions)}
        self.values = []
```

En el código anterior *problem* es un ejemplar de Sudoku y agregamos una optimización en forma de un diccionario, para conocer el indice de una posición (i, j), lo que será útil después en la función de evaluación. Además, mantenemos los vectores de valores y posiciones debido a su fácil manipulación.

Características de la Representación

- Mapeo Uno a uno: Notemos que, por si solo, el vector S puede representar la solución para múltiples ejemplares, sin embargo, al incluir el uso del vector de celdas vacías P, que llenamos al principio, podemos diferenciar una solución de otra con las posiciones únicas de cada valor.
- Representación lineal: Todos lo que utilizamos para los vectores de valores y posiciones, son arreglos mono-dimensionales, los cuales son estructuras lineales.
- Representación completa: Cada soluciones tiene una representación única y cada representación válida es una solución. Así, cada solución del problema puede ser representada usando nuestro modelo.
- Codificación directa (representación fenotipica): Si bien no empleamos la matriz directamente en la solución, los genes (valores de la solución) contienen los valores de las celdas sin requerir un proceso de decodificación. La evaluación opera directamente sobre estos valores.

b) Describe e implementa una función de evaluación para las soluciones propuestas.

Realiza el análisis de complejidad, y describe qué ventajas o desventajas podría tener la función propuesta.

Para esta parte retomamos la función objetivo, pero la definiremos para la representación que tenemos para una solución. Así, f esta definida como:

$$f(S) = C\_filas(S) + C\_columnas(S) + C\_bloques(S)$$

donde:

- S la representación de una solución.
- $f(S)$ es el número total de colisiones en la solución.
- C_filas número de colisiones en las filas.
- $C\_columnas$ número de colisiones en las columna.
- $C\_bloques$ número de colisiones en los bloques.

Decimos que una solución es correcta, si para cada grupo g (filas, columnas y bloques), cada número i en dicho grupo está en el conjunto $\{1,\ 2,\ \ldots,\ n\}$ y la frecuencia de aparición de i en g es 1. Así, para el calculo de C_filas, $C\_columnas$ y $C\_bloques$ utilizamos las frecuencias de cada número $i \in \{1, 2, \ldots, n\}$. Es decir, podemos definir la siguiente función $group\_conflicts(G)$:

a) $freq[v] \leftarrow$ contar ocurrencias de v en G

b) $colisiones \leftarrow \sum max(0, freq[v] - 1)$, para $v \in \{1,\ 2,\ \ldots,\ n\}$.

c) devolver *colisiones*.

Dentro del programa en el lenguaje Python se utilizo la siguiente función en la clase *Sudoku-Solution*:

```
def _count_conflicts_in_group(self, values):
    freq = Counter(values)
    return sum(max(0, count - 1) for count in freq.values())
```

Decimos que la complejidad de la función es $O(n)$ pues cada grupo, en cualquiera de sus formas (fila, columna o bloque), contiene n elementos, es decir que el conteo de frecuencias toma n pasos. Luego el conjunto de frecuencias tiene a lo más n elemento, por lo que la suma toma tiempo $O(n)$ también.

Ahora podemos definir las funciones de colisiones como sigue:

- Sea F el conjunto de las filas (vectores de tamaño $1 \times n$) en el ejemplar del problema:

$$C\_filas(S) = \sum_{f \in F} group\_conflicts(f)$$

- Sea C el conjunto de columnas (vectores de tamaño $n \times 1$) en el ejemplar:

$$C\_columnas(S) = \sum_{c \in C} group\_conflicts(c)$$

  ea B el conjunto de bloques (matrices de tamaño $\sqrt{n} \times \sqrt{n}$) en el ejemplar:

$$C\_columnas(S) = \sum_{b \in B} group\_conflicts(b)$$

Claramente, hace falta implementar funciones para encontrar los grupos del ejemplar, por lo que en la siguiente parte mostramos la implementación que nosotros hemos realizado en Python dentro de la clase `SudokuSolution` en forma de métodos:

Obtener un valor de una celda $(i,\ j)$:

```
def get_value(self, row, col):
    if self.problem.fixed_cells[row, col]:
        return self.problem.grid[row, col]
    else:
        idx = self.position_to_index[(row, col)]
        return self.values[idx]
```

Notemos que, gracias a la elección de estructuras, para cada uno de lso casos la búsqueda del valor, dada una posición (row, col) es siempre una operación de tiempo constante, $O(1)$.

Obtener elementos de una fila:

```
def get_row(self, row):
    return [self.get_value(row, col) for col in range(self.problem.size)]
```

Obtener elementos de una columna:

```
def get_column(self, col):
    return [self.get_value(row, col) for row in range(self.problem.size)]
```

Es fácil notar que para ambos casos, tanto para filas como para columnas, la complejidad es de $O(n)$, esto debido a que dado el indice una fila (o de columna, respectivamente) se recorren el las n columnas (filas) y se aplica la función `get_value` con complejidad $O(1)$.

Obtener los elementos en un bloque:

```
def get_block(self, block_row, block_col):
    values = []
    start_row = block_row * self.problem.block_size
    start_col = block_col * self.problem.block_size

    for i in range(start_row, start_row + self.problem.block_size):
        for j in range(start_col, start_col + self.problem.block_size):
            values.append(self.get_value(i, j))
    return values
```

Las asignaciones iniciales toman tiempo $O(1)$, por lo que no contribuyen a la complejidad final. Así, notemos que tanto ciclo `for` exterior como el interior realizan $k = \sqrt{n}$ iteraciones, es decir que realizan $k \times k = k^2 = n$ iteraciones. Luego, las operaciones en el ciclo interno son `append` y `get_value` que toman tiempo $O(1)$, por lo que la complejidad final es $O(n)$.

Función de evaluación que obtiene el número de colisiones:

```
def evaluate(self):
    n = self.problem.size
    k = self.problem.block_size
    total_conflicts = 0

    # 1. Colisiones en filas
    for row in range(n):
        values = self.get_row(row)
        total_conflicts += self._count_conflicts_in_group(values)

    # 2. Colisiones en columnas
    for col in range(n):
        values = self.get_column(col)
        total_conflicts += self._count_conflicts_in_group(values)

    # 3. Colisiones en bloques
    for block_row in range(k):
        for block_col in range(k):
            values = self.get_block(block_row, block_col)
            total_conflicts += self._count_conflicts_in_group(values)

    return total_conflicts
```

La complejidad temporal de la función de evaluación será la suma de las complejidades del calculo de colisiones en fila, en columnas y en bloques. Para el calculo de filas utilizamos las funciones `get_row` y `count_conflicts_in_group` en n iteraciones, las funciones nombradas toman tiempo $O(n)$ ambas, por lo que esta parte toma tiempo $O(n^2)$.

Por un argumento semejante al caso anterior sabemos que el calculo de colisiones en las columnas igual toma tiempo $O(n^2)$.

Finalmente, notemos que los ciclos anidados, para el calculo en bloques, realizan un total de $k \times k = \sqrt{n} \times \sqrt{n} = n$ iteraciones, por lo que también tenemos una complejidad de $O(n^2)$.

Así la complejidad final es

$$O(n^2) + O(n^2) + O(n^2) = O(n^2)$$

c) Describir e implementar un generador de soluciones aleatorias.
Gracias a nuestra representación de una solución no hace falta modificar el vector P de posiciones libres en el ejemplar, si no que basta con generar $|P|$ valores del conjunto $\{1,\ 2,\ \ldots,\ n\}$ y guardarlos en S. Sin embargo, debemos de tener cuidado, ya que nos interesa generar la cantidad correcta de valores faltantes en posiciones aleatorias. Esto es, dado un valor i y el número d de veces que aparece como valor fijo en el ejemplar del sudoku, el vector de valores S debe contener exactamente $n - d$ ejemplares de i.

Por lo tanto, procedimiento para generar una solución aleatorio es el siguiente:

a) $repeticiones \leftarrow []$

b) para cada $v \in \{1,\ 2,\ \ldots, n\}$
   - $d \leftarrow$ número de apariciones como valor fijo en el ejemplar del problema.
   - $repeticiones[v] \leftarrow n - d$

c) $S \leftarrow []$

d) Para cada $v \in \{1,\ 2,\ \ldots, n\}$
   - Agregar $repeticiones[v]$ veces el valor v a S.

e) Mezclar aleatoriamente los valores en S

La implementación en el lenguaje Python es la siguiente:

```
def _generate_random_solution(self):
    n = self.problem.size

    fixed_counts = Counter()
    for i in range(n):
        for j in range(n):
            if self.problem.fixed_cells[i, j]:
                value = self.problem.grid[i, j]
                fixed_counts[value] += 1

    needed_counts = {}
    for value in range(1, n + 1):
        current_count = fixed_counts.get(value, 0)
        needed = n - current_count
        needed_counts[value] = needed

    values_needed = []
    for value, count in needed_counts.items():
        values_needed.extend([value] * count)

    random.shuffle(values_needed)

    return values_needed
```

Así, podemos agregar al constructor de `SudokuSolution` la generación de una solución inicial con la línea:

```
self.values = self._generate_random_solution()
```

Recocido Simulado

a) Describe e implementa un método que funcione como operador para generar un vecino aleatorio.

Nuestra implementación copia la solución actual del sudoku, y de manera aleatoria eligen dos celdas, las cuales intercambia de posición para devolver un vecino.

Se considero la opción de intercambiar valores cuyas posiciones se encuentre sobre el mismo grupo (fila, columna o bloque), pero en las pruebas este método completamente aleatorio fue el que mejor resultados presento.

```
def get_neighbor(self):
    if self.num_empty < 2:
        return self.copy()
    # Crear copia de la solución actual
    neighbor = self.copy()
    idx1, idx2 = random.sample(range(self.num_empty), 2)

    neighbor.values[idx1],neighbor.values[idx2]=
    neighbor.values[idx2],neighbor.values[idx1]
    return neighbor
```

b) Describe e implementa un método de enfriamiento por cada integrante del equipo - Justifiquen la elección del esquema de enfriamiento.

Enfriamiento Geométrico: La temperatura se multiplica por una constante entre 0 y 1 y se asigna ese nuevo valor. Sin embargo, para evitar que la temperatura baje muy rápido y para favorecer más la exploración hemos implementado un hiperparametro extra N, el cual representa el número de iteraciones que deberá realizar el recocido simulado antes de disminuir la temperatura. El valor N se actualiza con un factor $p = 1.15$.

```
def geometric_cooling(current_temperature, alpha):
    return alpha * current_temperature
```

Hemos elegido este método de enfriamiento puesto que otorga un enfriamiento lento, lo que en temperaturas altas permite soluciones *malas* con una alta probabilidad (exploración) y refina las soluciones cuando la temperatura es muy baja. Adicionalmente, el hiperparametro N hacer que mientras más baja sea la temperatura, mayor será la exploración, lo que nos permitirá escapar de óptimos locales en temperatura bajas.

Enfriamiento con Decremento Lento: Se calcula con una división, donde el denominador es la temperatura multiplicada por una constante entre 0 y 1; y el numerador es la temperatura actual. Esta funciona para hacer mucha explotación o mucha exploración, por lo que se ha decantado usarla para valores que combinen lo mejor de ambos y den mejores resultados

```
    def slow_cooling(current_temperature, alpha):
return  (current_temperature/(1+alpha*current_temperature))
```

Enfriamiento Linesal: Este otro esquema de enfriamiento es el enfriamiento lineal y en este caso, la temperatura se reduce en cada iteración de manera constante siempre y cuando que $\beta > 0$, y $T_{k+1} = T_k - \beta$.

La elección de β controla la rapidez con la que se enfría el sistema:

- Si β es grande, la temperatura desciende rápido y se privilegia la *explotación*.
- Si β es pequeño, la temperatura desciende lentamente y se favorece la *exploración*.

La principal desventaja es que, si β no se escoge bien, la temperatura puede llegar a 0 demasiado rápido y hacer que el algoritmo se congele antes de explorar lo suficiente.

```
def linear_cooling(current_temperature, beta):
    return current_temperature - beta
```

c) Implementa el algoritmo de recocido simulado para el problema del sudoku utilizando el esquema de representación de soluciones del ejercicio anterior.

```
def simulated_annealing(problem, initial_temp=100.0, alpha=0.0005,
    N0_factor = 2, p=1.15, max_iteration = 250000, cooling='l'):
    current_solution = SudokuSolution(problem)
    current_fitness = current_solution.evaluate()
    best_solution = current_solution.copy()
    best_fitness = current_fitness

    N = int(N0_factor * problem.size)
    temperature = initial_temp
    iteration = 0

    while temperature > 1e-4 and best_fitness > 0 and iteration < max_iteration:
        for _ in range(N):
            neighbor = current_solution.get_neighbor()
            neighbor_fitness = neighbor.evaluate()

            delta_fitness = neighbor_fitness - current_fitness
            if delta_fitness <= 0:
                accept = True
            else:
                probability = math.exp(-delta_fitness / temperature)
                accept = random.random() < probability

            if accept:
                current_solution = neighbor
                current_fitness = neighbor_fitness

                if current_fitness < best_fitness:
                    best_solution = current_solution.copy()
                    best_fitness = current_fitness

            iteration += 1
        if cooling== 'g':
```

```
            alpha = 0.88
            temperature = geometric_cooling(temperature, alpha)
            N = int(N * p)
        if cooling=='s':
            alpha = 0.0005
            temperature = slow_cooling(temperature, alpha)
        if cooling=='l':
            beta = initial_temp/max_iteration
            temperature = linear_cooling(initial_temp, beta * iteration)

    return best_solution, best_fitness
```

Se ha implementado el algoritmo en un método principal con auxiliares. Se inicializan variables de mejor vecino, solución y fitness para ambos casos. Y se aplica el recocido simulado.

Experimentación

Considera los ejemplares de prueba proporcionados. Ejecutar el método al menos 10 veces para cada ejemplar y por cada método de enfriamiento. Deberán registrar y agregar en el reporte tus resultados en una tabla.

Tabla 1: Resultados generales para David_Filmer1.txt

Repetición	Geometric (g)		Slow (s)		Linear (l)	
	Fitness	Iteraciones	Fitness	Iteraciones	Fitness	Iteraciones
1	2	286648	2	250002	15	250002
2	5	286648	3	250002	12	250002
3	2	286648	2	250002	13	250002
4	2	286648	2	250002	14	250002
5	2	286648	4	250002	17	250002
6	2	286648	2	250002	14	250002
7	3	286648	2	250002	16	250002
8	2	286648	2	250002	17	250002
9	2	286648	4	250002	11	250002
10	2	286648	2	250002	13	250002

Tabla 2: Resultados generales para Easy1.txt

Repetición	Geometric (g)		Slow (s)		Linear (l)	
	Fitness	Iteraciones	Fitness	Iteraciones	Fitness	Iteraciones
1	0	40531	0	75474	4	250002
2	0	26650	2	250002	2	250002
3	0	23173	0	93438	9	250002
4	0	26650	2	250002	8	250002
5	5	286648	0	85896	2	250002
6	0	40531	0	70146	6	250002
7	0	17519	0	118980	11	250002
8	2	286648	0	87732	4	250002
9	0	13243	0	80010	7	250002
10	0	46609	0	106524	6	250002

Tabla 3: Resultados generales para Hard1.txt

Repetición	Geometric (g)		Slow (s)		Linear (l)	
	Fitness	Iteraciones	Fitness	Iteraciones	Fitness	Iteraciones
1	2	286648	2	250002	12	250002
2	2	286648	4	250002	12	250002
3	2	286648	0	147510	14	250002
4	4	286648	2	250002	12	250002
5	4	286648	4	250002	8	250002
6	2	286648	0	101556	7	250002
7	2	286648	2	250002	12	250002
8	4	286648	0	104688	11	250002
9	2	286648	4	250002	11	250002
10	4	286648	2	250002	11	250002

Tabla 4: Resultados generales para Medium1.txt

Repetición	Geometric (g)		Slow (s)		Linear (l)	
	Fitness	Iteraciones	Fitness	Iteraciones	Fitness	Iteraciones
1	6	286648	4	250002	6	250002
2	5	286648	0	96696	9	250002
3	2	286648	2	250002	11	250002
4	4	286648	0	132786	15	250002
5	5	286648	2	250002	6	250002
6	0	70877	2	250002	11	250002
7	7	286648	4	250002	12	250002
8	2	286648	2	250002	13	250002
9	0	70877	4	250002	15	250002
10	4	286648	0	120888	13	250002

Tabla 5: Resultados generales para SD2.txt

Repetición	Geometric (g)		Slow (s)		Linear (l)	
	Fitness	Iteraciones	Fitness	Iteraciones	Fitness	Iteraciones
1	2	286648	2	250002	12	250002
2	2	286648	4	250002	14	250002
3	2	286648	2	250002	12	250002
4	3	286648	2	250002	15	250002
5	2	286648	3	250002	12	250002
6	2	286648	2	250002	11	250002
7	4	286648	2	250002	16	250002
8	7	286648	2	250002	8	250002
9	2	286648	2	250002	17	250002
10	3	286648	2	250002	17	250002

Tabla 6: Resultados combinados de todos los ejemplares

Ejemplar	Método Enfriamiento	Dimensión	Mejor valor f(x)	Valor promedio f(x)	Peor valor f(x)
David_Filmer1	Geometric	9x9	2.0	2.4	5.0
David_Filmer1	Slow	9x9	2.0	2.5	4.0
David_Filmer1	Linear	9x9	11.0	14.2	17.0
Easy1	Geometric	9x9	0.0	0.7	5.0
Easy1	Slow	9x9	0.0	0.4	2.0
Easy1	Linear	9x9	2.0	5.9	11.0
Hard1	Geometric	9x9	2.0	2.8	4.0
Hard1	Slow	9x9	0.0	2.0	4.0
Hard1	Linear	9x9	7.0	11.0	14.0
Medium1	Geometric	9x9	0.0	3.5	7.0
Medium1	Slow	9x9	0.0	2.0	4.0
Medium1	Linear	9x9	6.0	11.1	15.0
SD2	Geometric	9x9	2.0	2.7	7.0
SD2	Slow	9x9	2.0	2.3	4.0
SD2	Linear	9x9	8.0	13.4	17.0

Conclusión:

La práctica mostró que el recocido simulado es una técnica efectiva para resolver Sudokus, pero su desempeño depende del esquema de enfriamiento. El geométrico y el *slow* lograron buenos resultados, incluso resolviendo algunos casos sin ninguna clase de error, mientras que el lineal tuvo un rendimiento mucho menor al mantener más conflictos y colisiones.