



Solucionador de Sudoku

Trabajo Final Inteligencia Artificial

Integrantes:
Bruno Bearzotti
Martina Kozelnik
Martina Lagares
Emanuel Rodriguez
Germán Tarnoski



Contenidos



01

Introducción

02

Demostración

03

Características de
la Población

04

Función de Aptitud

05

Selección

06

Crossover y
Mutación





01

Introducción









Objetivo

Crear un Algoritmo Genético que sea capaz de resolver Sudokus de 9x9.

Se parte de un tablero de sudoku con valores faltantes.



	2		4	5	6	7	8	9
4	5	6	7			1		3
7		9	1	2	3	4	5	
	1	4	3	6	5	8	9	
3	6	5	8	9	7	2	1	4
8		7			4	3	6	5
		1		4	2		7	8
	4	2	9	7		5	3	1
9	7	8	5	3	1	6		2





Tablero

	2		4	5	6	7	8	9
4	5	6	7			1		3
7		9	1	2	3	4	5	
	1	4	3	6	5	8	9	
3	6	5	8	9	7	2	1	4
8		7			4	3	6	5
		1		4	2		7	8
	4	2	9	7		5	3	1
9	7	8	5	3	1	6		2

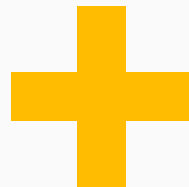


Solución

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

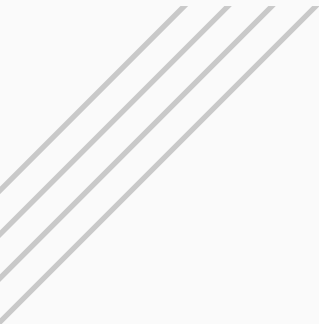


1
1



02

Demostración



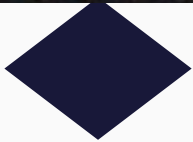


Tablero

0	2	0		4	5	6		7	8	9
4	5	6		7	0	0		1	0	3
7	0	9		1	2	3		4	5	0

0	1	4		3	6	5		8	9	0
3	6	5		8	9	7		2	1	4
8	0	7		0	0	4		3	6	5

0	0	1		0	4	2		0	7	8
0	4	2		9	7	0		5	3	1
9	7	8		5	3	1		6	0	2

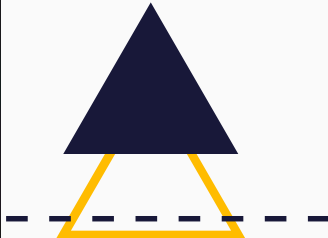


Solucion

1	2	3		4	5	6		7	8	9
4	5	6		7	8	9		1	2	3
7	8	9		1	2	3		4	5	6

2	1	4		3	6	5		8	9	7
3	6	5		8	9	7		2	1	4
8	9	7		2	1	4		3	6	5

5	3	1		6	4	2		9	7	8
6	4	2		9	7	8		5	3	1
9	7	8		5	3	1		6	4	2





03



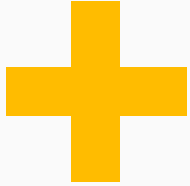
Características de la Población



Población

La población es un array (lista).
Cada individuo de la población representa un intento de solución del sudoku dado, donde se completan las celdas vacías con valores.

poblacion=50
mutation_rate=0,3
generaciones(limite)=100





Genotipo vs fenotipo

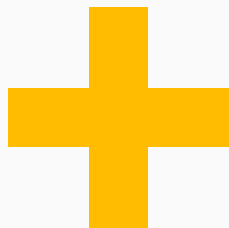


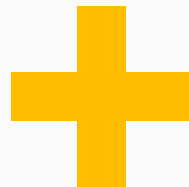
Genotipo

Matriz que representa el
tablero de sudoku.
Completado con valores
del 1 al 9.

Fenotipo

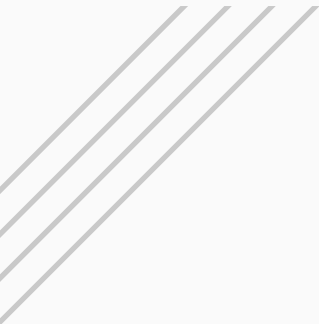
Sudoku Completado





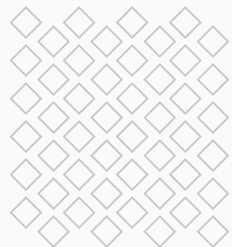
04

Función de Aptitud





Fitness



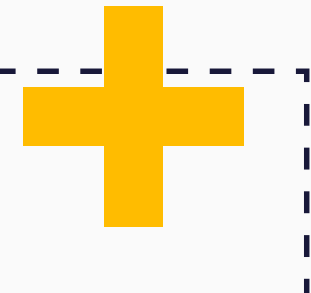
Controla que en cada fila, columna y sector haya números únicos.

Suma la cantidad de números únicos por fila (9 números por cada fila).

Suma la cantidad de números únicos por columna (9 números por cada columna).

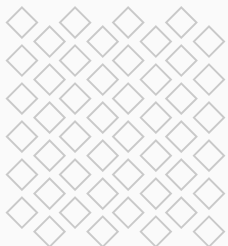


Suma la cantidad de números únicos por sector (9 números por cada sector).





Fitness

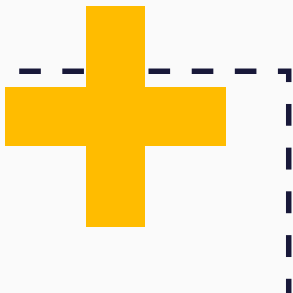


$$9 \times 9 + 9 \times 9 + 9 \times 9$$

$$81 + 81 + 81 = 243$$



El sudoku se resuelve cuando se llega a 243 de fitness.



```
class FitnessEvaluator:
    @staticmethod
    def evaluate(solution):
        fitness = 0

        # Check rows
        # recorre cada fila, y suma los valores unicos.
        for row in solution:
            fitness += len(set(row))

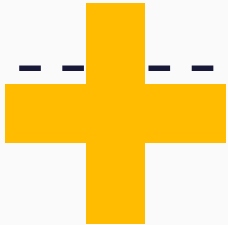
        # Check columns
        # recorre cada columna, y suma los valores unicos.
        for col in range(9):
            column = [solution[row][col] for row in range(9)]
            fitness += len(set(column))

        # Check 3x3 boxes
        # recorre cada cuadrante, y suma los valores unicos.
        for row in range(0, 9, 3):
            for col in range(0, 9, 3):
                box = [solution[r][c] for r in range(row, row+3) for c in range(col, col+3)]
                fitness += len(set(box))

        # el fitness devolveria 243 si no hay errores. 243 valores que no se repiten.
        return fitness
```



```
for i in range(self.max_generations):  
    for solution in population:  
        fitness=fitness_evaluator.evaluate(solution)  
        if fitness == 243:  
            print("Solución encontrada en la generacion", i + 1)  
            print()  
            return solution
```





05

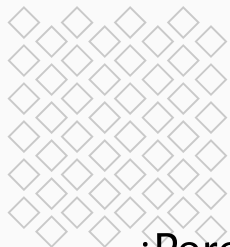
Selección





Método del dardo

Asignamos probabilidades a cada individuo (posible solución) según su fitness.



¿Porque?



Fitness=243

Fitness=240

Fitness=220

Fitness=205

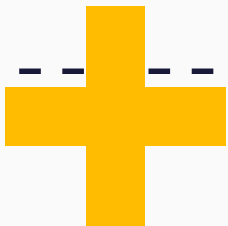
$1/243=0,004115$


$1/240=0,004166$

$1/220=0,00454$

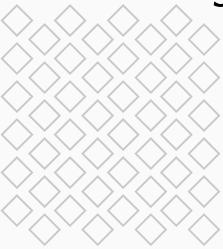
$1/205=0,00487$

Baja probabilidad de ocurrencia, valores pequeños.






A partir de esas probabilidades, se toman al azar dos individuos que serán los padres.




```
probabilities = self.calculateProbabilities(population, fitness_evaluator)
```

```
parents = random.choices(population, probabilities, k=2)  
parent1, parent2 = parents[0], parents[1]
```



```
def calculateProbabilities(self, population, fitness_function):  
    fitnesses = [fitness_function.evaluate(solution) for solution in population]  
    total_fitness = sum(fitnesses)  
    return [fitness / total_fitness for fitness in fitnesses]
```





06

Crossover y Mutación



Crossover

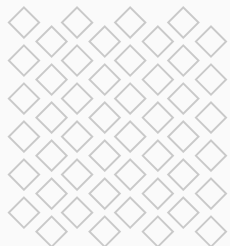
Se usan dos padres para crear dos hijos. Cada hijo tendrá una parte de cada padre. Cuánto toman de cada padre es determinado aleatoriamente.

6	9	2	7	3	8	4	1	5
3	1	7	4	2	5	6	9	8
5	4	8	6	1	9	3	2	7
2	7	4	5	6	1	8	3	9
1	8	5	9	7	3	2	4	6
9	3	6	2	8	4	5	7	1
7	2	1	3	5	6	9	8	4
8	5	9	1	4	2	7	6	3
4	6	3	8	9	7	1	5	2

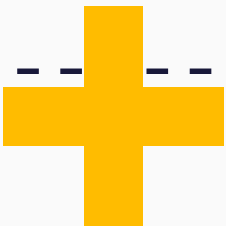
4	7	8	3	9	2	6	5	1
6	9	3	7	1	5	4	2	8
5	2	1	4	6	8	3	9	7
1	6	5	8	2	4	7	3	9
3	4	9	1	5	7	2	8	6
7	8	2	6	3	9	5	1	4
2	5	6	9	4	1	8	7	3
8	1	4	2	7	3	9	6	5
9	3	7	5	8	6	1	4	2



```
child1, child2 = self.crossover_operator.crossover(parent1, parent2)
```



```
def crossover(solution1, solution2):  
    row = random.randint(0, 8)  
  
    # Usar deepcopy para realizar una copia profunda  
    child1 = copy.deepcopy(solution1[:row] + solution2[row:])  
    child2 = copy.deepcopy(solution2[:row] + solution1[row:])  
  
    return child1, child2
```





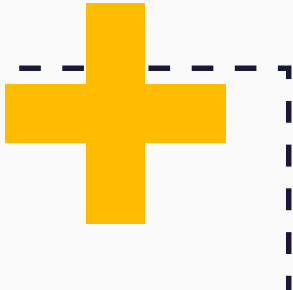
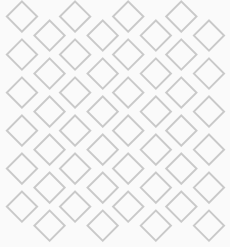
Mutacion

`mutation_rate=0,3`

Usando un número elegido al azar y comparándolo con el mutation rate, se determinará si se aplica una mutación a ese individuo o no.

Se comprobará que el casillero a mutar no sea parte del tablero inicial.

Se inserta un valor que no esté repetido en la fila, columna o cuadrante.



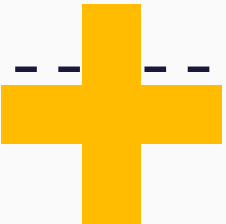
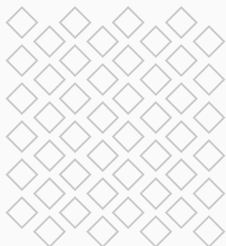


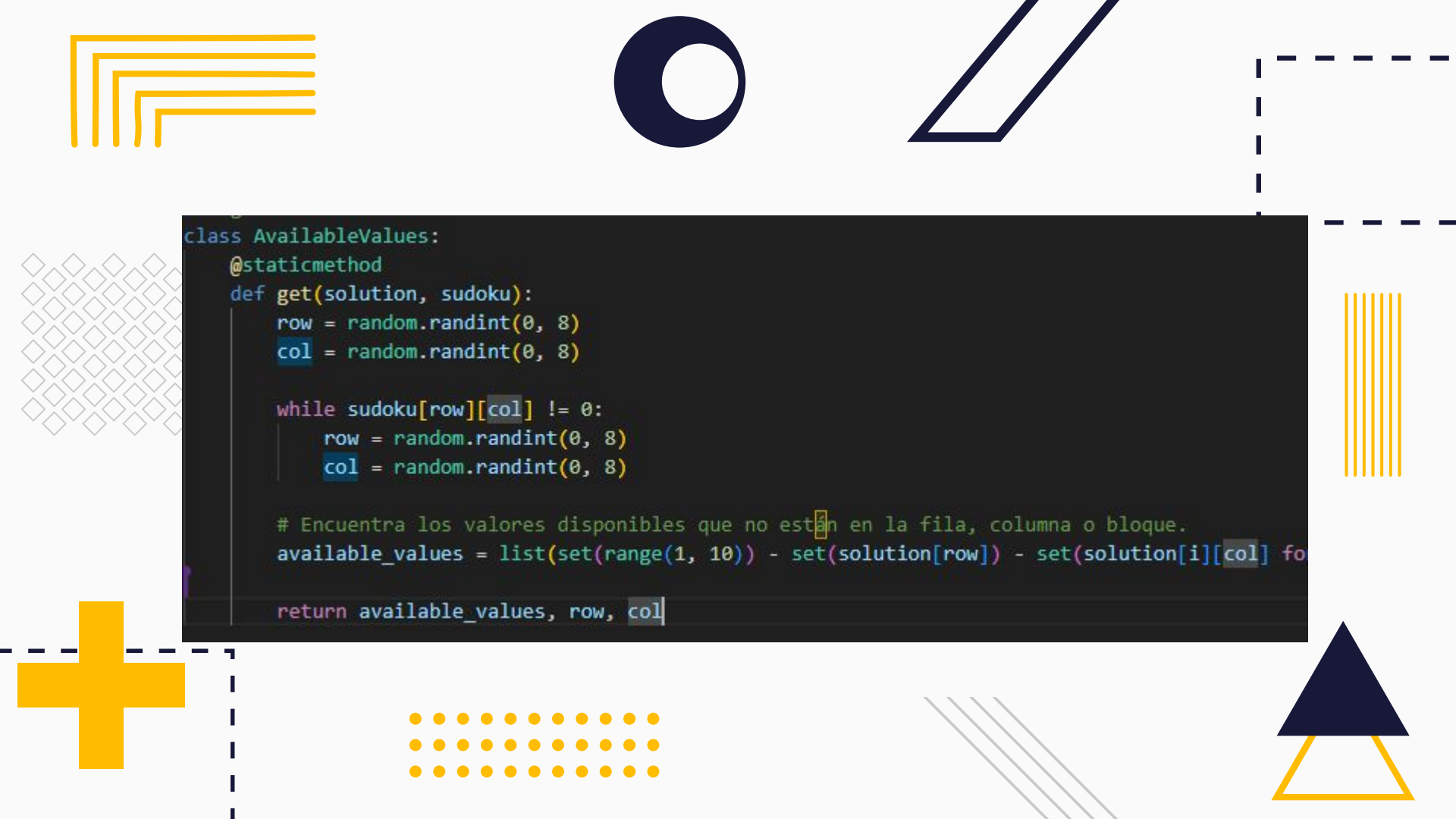
```
# Muta a los hijos con una cierta probabilidad
if random.random() < mutation_rate:
    child1 = self.mutation_operator.mutate(child1, sudoku)
if random.random() < mutation_rate:
    child2 = self.mutation_operator.mutate(child2, sudoku)
```

```
def mutate(self, solution, sudoku):
    for _ in range(20): # Intenta realizar 20 mutaciones
        available_values, row, col = AvailableValues.get(solution, sudoku)

        if available_values:
            value = random.choice(available_values)
            solution[row][col] = value
            break

    return solution
```





```
class AvailableValues:
    @staticmethod
    def get(solution, sudoku):
        row = random.randint(0, 8)
        col = random.randint(0, 8)

        while sudoku[row][col] != 0:
            row = random.randint(0, 8)
            col = random.randint(0, 8)

        # Encuentra los valores disponibles que no están en la fila, columna o bloque.
        available_values = list(set(range(1, 10)) - set(solution[row]) - set(solution[i][col] for i in range(0, 8) if (i - row) % 3 != 0 and (i + row) % 3 != 0))

        return available_values, row, col
```




06

Conclusiones





Gracias

