SEMINARIO DE SOLUCION DE PROBLEMAS DE INTELIGENCIA ARTIFICIAL I

Vázquez Pérez Ignacio David

218292866

Ingeniería en computación

Algoritmo Genético Binario

Hay una fila de n monedas cuyos valores son enteros positivos C1, C2, .. , Cn no necesariamente distintos. La meta es tomar la máxima cantidad de dinero sujeto a la restricción de que no pueden ser tomadas dos monedas adyacentes en la fila.

Implementar una solución al problema descrito utilizando un Algoritmo Genético.

Resolver el problema de la fila de monedas para la siguiente lista de monedas: [1, 20, 5, 1, 2, 5, 5, 1, 5, 2, 2, 1, 10, 5, 10, 5, 20, 20, 20, 5, 1, 1, 20, 20, 1, 10, 2, 10, 5, 2, 10, 1, 20, 1, 20, 10, 5, 5, 20, 2, 10, 1, 2, 5, 10, 20, 10, 2, 5, 5, 20, 1, 1, 5, 10, 10, 10, 1, 5, 2, 1, 2, 10, 20, 2, 10, 10, 20, 5, 10, 1, 2, 1, 5, 20, 2, 5, 1, 5, 10, 2, 5, 10, 2, 1, 1, 1, 10, 20, 10, 20, 2, 2, 10, 20, 10, 1, 1, 5, 2]

Implementación del Algoritmo Genético:

1. Representación del individuo: Cada individuo en la población es una posible disposición de monedas en la fila. Representaremos una solución como una lista de 0s y 1s, donde un 1 indica que una moneda se toma y un 0 indica que no se toma. La longitud de la lista es igual al número de monedas en la fila.

2. Función de fitness: La función de fitness evaluará la calidad de una disposición de monedas. Cuanto mayor sea la suma de los valores de las monedas tomadas, mejor será la solución. Sin embargo, debemos tener en cuenta la restricción de que no se pueden tomar dos monedas adyacentes. Por lo tanto, si encontramos dos monedas adyacentes tomadas, penalizaremos la solución reduciendo su valor de fitness.

3. Proceso de evolución: Utilizaremos operadores genéticos como la selección, el cruce y la mutación para evolucionar la población. Después de evaluar el fitness de cada individuo, seleccionaremos a los mejores individuos para reproducirse mediante cruce. Además, aplicaremos una pequeña tasa de mutación para introducir variabilidad en la población.

Implementación de la Función de Fitness:

La función de fitness recorre la disposición de monedas y calcula la suma de los valores de las monedas tomadas. Además, penaliza la solución si encuentra dos monedas adyacentes tomadas, reduciendo su valor de fitness. Esto se hace restando la mitad del valor de la moneda adyacente si ambas monedas están tomadas.

In [ ]:
```python
import numpy as np

# Definir la lista de monedas
coins = [1, 20, 5, 1, 2, 5, 5, 1, 5, 2, 2, 1, 10, 5, 10, 5, 20, 20, 20, 5, 1, 1, 20, 20, 1, 10, 2, 10, 5, 2, 10, 1, 20,

# Función de fitness
def fitness(solution):
    total_value = 0
    for i in range(len(solution)):
        if solution[i] == 1:
            total_value += coins[i]
            # Penalizar soluciones con monedas adyacentes tomadas
            if i > 0 and solution[i - 1] == 1:
                total_value -= coins[i] // 2
    return total_value

# Algoritmo genético
def genetic_algorithm(population_size, generations):
    # Inicializar la población con soluciones aleatorias
    population = np.random.randint(2, size=(population_size, len(coins)))

    for generation in range(generations):
        # Calcular el fitness de cada individuo en la población
        fitness_values = [fitness(individual) for individual in population]

        # Seleccionar los mejores individuos para la reproducción
        selected_indices = np.argsort(fitness_values)[-population_size // 2:]
        selected_individuals = population[selected_indices]

        # Aplicar cruce
        offspring = []
        for _ in range(population_size - len(selected_individuals)):
            indices = np.random.randint(len(selected_individuals), size=2)
            parent1, parent2 = selected_individuals[indices[0]], selected_individuals[indices[1]]
            crossover_point = np.random.randint(len(coins))
            child = np.concatenate([parent1[:crossover_point], parent2[crossover_point:]])
```

```
            offspring.append(child)
        offspring = np.array(offspring)

        # Aplicar mutación
        mutation_rate = 0.05
        mutation_mask = np.random.rand(*offspring.shape) < mutation_rate
        offspring ^= mutation_mask

        # Reemplazar la población con la descendencia
        population = np.vstack([selected_individuals, offspring])

    # Obtener la mejor solución encontrada
    best_solution_index = np.argmax([fitness(individual) for individual in population])
    best_solution = population[best_solution_index]
    best_fitness = fitness_values[best_solution_index]

    return best_solution, best_fitness

# Parámetros del algoritmo genético
population_size = 100
generations = 1000

# Ejecutar el algoritmo genético
best_solution, best_fitness = genetic_algorithm(population_size, generations)

print("Mejor solución encontrada:", best_solution)
print("Fitness de la mejor solución:", best_fitness)
```

```
Mejor solución encontrada: [0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1
 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 1 0 1 0 1 0 1 0 1 1 0
 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1]
Fitness de la mejor solución: 518
```

Conclusión

El problema de la fila de monedas se resolvió de manera efectiva utilizando un enfoque de algoritmo genético. El enfoque de algoritmo genético proporcionó una manera intuitiva de buscar soluciones óptimas al problema, permitiendo la exploración de un amplio espacio de soluciones. La implementación de la función de fitness fue crucial para evaluar la calidad de las soluciones y garantizar que se cumpliera la restricción de que no se pueden tomar dos monedas adyacentes. En general, los algoritmos genéticos son herramientas poderosas para abordar problemas de optimización combinatoria como este.

In [ ]:

In [ ]: