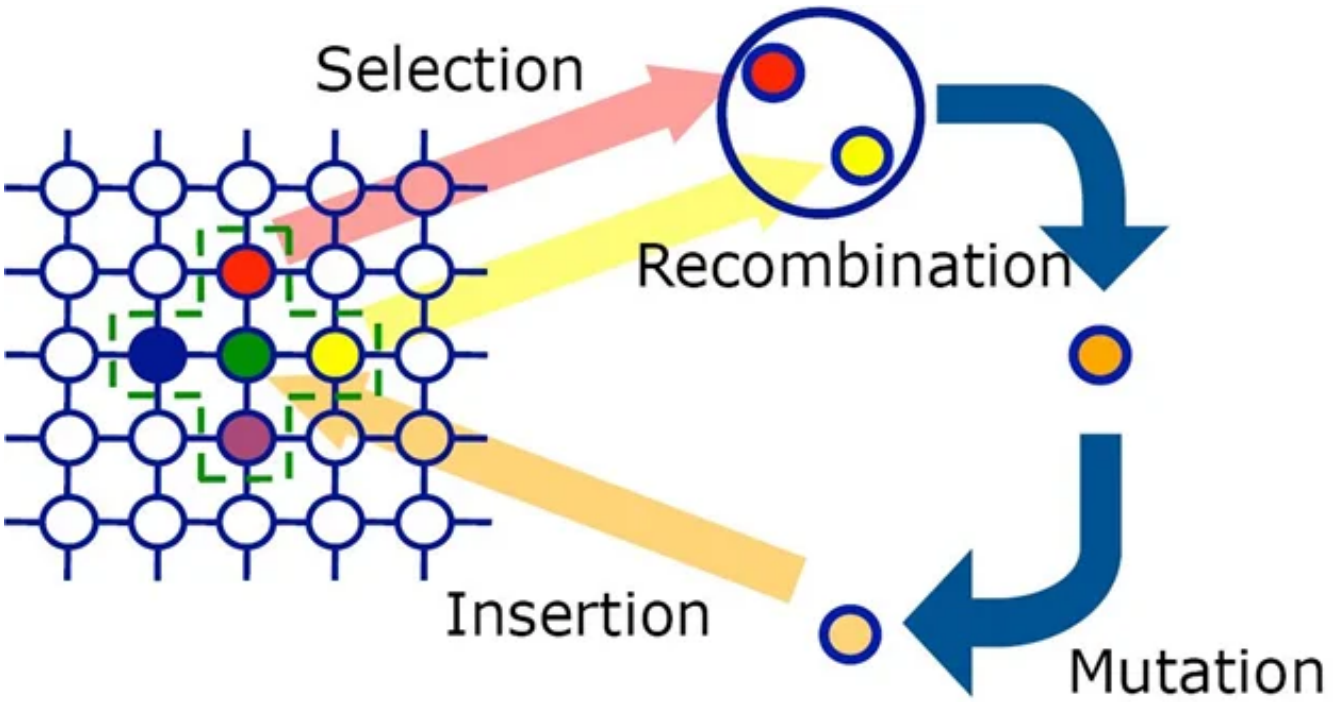
Algoritmos Genéticos



Adrián Yared Armas de la Nuez

Contenido

1. Enunciado

La tarea consiste en encontrar el subconjunto de películas cuyo tamaño mejor se ajusta a uno pedido (sin pasarse).
Se implementará una aplicación en Python de un algoritmo GENÉTICO que BUSQUE SOLUCIONES para grabar películas en DVD´s de DISTINTOS TAMAÑOS realizando el mejor ajuste para no desaprovechar el espacio de almacenamiento.

(Se probará con los tamaños 4.7, 8.5, 13.3 Y 15.9 GBytes mostrando las mejores soluciones obtenidas).

Las películas están catalogadas por distintos géneros, ACCIÓN, TERROR, COMEDIA y SUSPENSE.

- En un mismo DVD no se podrán mezclar películas de COMEDIA y TERROR.
- Tampoco se quiere poner en el mismo DVD las películas Jugada salvaje y El cuerpo ni Furia de Titanes y El hombre de acero.

	TÍTULO	GÉNERO	TAMAÑO EN GB
1	La última casa a la izquierda	TERROR	1,830
2	Saw IV	TERROR	1,435
3	La huérfana	TERROR	2,163
4	Furia de Titanes	ACCIÓN	1,746
5	El hombre de acero	ACCIÓN	0,964
6	Los Vengadores	ACCIÓN	2,032
7	American Pie: El reencuentro	COMEDIA	1,257
8	El lado bueno de las cosas	COMEDIA	3,139
9	Los tres chiflados	COMEDIA	0,750
10	Jugada salvaje	SUSPENSE	2,275
11	El cuerpo	SUSPENSE	2,082
12	15 años y un día	SUSPENSE	2,321

Criterios de calificación:
(4 Puntos) El algoritmo muestra soluciones aceptables
(4 Puntos) Se crean y comentan los métodos de mutación, fitness, y recombinación.
(2 Puntos) Uso del vocabulario correcto (población, generación, cromosoma etc) así como explicación de los hiperparámetos (probabilidad de recombinación, de mutación, población inicial etc).

2. Explicación del código por partes

2.1 Datos de Películas y Restricciones

2.1.1 Código

```
# List of movies with attributes: title, genre, and size in GB.
movies = [
    {"title": "The Last House on the Left", "genre": "HORROR", "size":
1.830},
    {"title": "Saw IV", "genre": "HORROR", "size": 1.435},
    {"title": "Orphan", "genre": "HORROR", "size": 2.163},
    {"title": "Clash of the Titans", "genre": "ACTION", "size": 1.746},
    {"title": "Man of Steel", "genre": "ACTION", "size": 0.964},
    {"title": "The Avengers", "genre": "ACTION", "size": 2.032},
    {"title": "American Reunion", "genre": "COMEDY", "size": 1.257},
    {"title": "Silver Linings Playbook", "genre": "COMEDY", "size":
3.139},
    {"title": "The Three Stooges", "genre": "COMEDY", "size": 0.750},
    {"title": "Wild Card", "genre": "THRILLER", "size": 2.275},
    {"title": "The Body", "genre": "THRILLER", "size": 2.082},
    {"title": "15 Years and One Day", "genre": "THRILLER", "size":
2.321}
]

# Restrictions: certain movies cannot be selected together, represented
as pairs.
movie_restrictions = [
    ("Wild Card", "The Body"),
    ("Clash of the Titans", "Man of Steel")
]

# Genre restrictions: restrict selecting both genres in a single DVD.
genre_restrictions = ("COMEDY", "HORROR")
```

2.1.1 Explicación

Esta es la lista de películas dictadas por el enunciado e introducidas en una lista de diccionarios.

`movie_restrictions`: Algunas parejas de películas no pueden estar en el mismo DVD.

genre_restrictions: Evita que Comedia y Terror estén juntos en el mismo DVD

2.2 Tamaños de DVD

2.2.1 Código

```
# Available DVD sizes in GB.
dvd_sizes = [4.7, 8.5, 13.3, 15.9]
```

2.2.1 Explicación

Esta lista contiene las capacidades en GB de diferentes tamaños de DVD.

2.3 Parámetros del Algoritmo Genético

2.3.1 Código

```
# Genetic algorithm parameters.
NUM_INDIVIDUALS = 100  # Population size
GENERATIONS = 100      # Number of generations
MUTATION_PROBABILITY = 0.1  # Probability of mutation
```

2.3.1 Explicación

NUM_INDIVIDUOS: Número de individuos en cada generación.
GENERACIONES: Cantidad de generaciones a evaluar.
PROB_MUTACION: Probabilidad de que ocurra una mutación en un individuo

2.4 Función calculate_fitness

2.4.1 Código

```
def calculate_fitness(individual, dvd_size):
    # Total size of selected movies.
    total_size = sum(m["size"] for i, m in enumerate(movies) if
individual[i] == 1)

    # If total size exceeds DVD capacity, return a fitness score of 0.
    if total_size > dvd_size:
        return 0

    # Check genre restrictions: no selection should include both
restricted genres.
```

```
    genres = {movies[i]["genre"] for i in range(len(movies)) if
individual[i] == 1}
    if genre_restrictions[0] in genres and genre_restrictions[1] in
genres:
        return 0

    # Check pair restrictions: restricted pairs should not be selected
together.
    for restriction in movie_restrictions:
        if any(m["title"] == restriction[0] for i, m in
enumerate(movies) if individual[i] == 1) and \
           any(m["title"] == restriction[1] for i, m in
enumerate(movies) if individual[i] == 1):
            return 0

    return total_size  # Return the total size if all restrictions are
met.
```

2.4.1 Explicación

Esta función evalúa a los individuos y si se pasa del máximo su valor fitness será 0. En total_size calcula el tamaño total de las películas con un 1, si se supera el tamaño se pone 0, además comprueba que no se mezclen géneros o títulos prohibidos por el enunciado.

2.5 Función create_individual

2.5.1 Código

```
def create_individual():
    # Create a random individual (a random binary selection of movies).
    return [random.randint(0, 1) for _ in movies]
```

2.5.1 Explicación

Crea un individuo aleatorio con una lista binaria en la que cada posición representa si una película está seleccionada (1) o no (0).

2.6 Función Selection

2.6.1 Código

```
def selection(population, dvd_size):
    # Calculate fitness for each individual and sort by fitness.
    scores = [(ind, calculate_fitness(ind, dvd_size)) for ind in
population]
    scores = sorted(scores, key=lambda x: x[1], reverse=True)
    # Select top 50% individuals for mating.
    selected = [s[0] for s in scores[:NUM_INDIVIDUALS // 2]]
    return selected
```

2.6.1 Explicación

Selecciona los mejores individuos de la población basándose en su fitness, para ello calcula el fitness de cada individuo, a continuación ordena la población de mayor a menor fitness y finalmente, selecciona la mitad superior de la población para la próxima generación.

2.7 Función Crossover

2.7.1 Código

```
def crossover(ind1, ind2):
    # Single-point crossover between two parents.
    point = random.randint(1, len(ind1) - 2)
    return ind1[:point] + ind2[point:], ind2[:point] + ind1[point:]
```

2.7.1 Explicación

Realiza el cruce entre dos individuos (padres) en un punto aleatorio, generando dos nuevos individuos (hijos). Para ello escoge un punto de cruce al azar e intercambia los genes (películas seleccionadas) de ambos padres desde ese punto para crear dos hijos.

2.8 Función Mutation

2.8.1 Código

```
def mutation(individual):
    # Apply mutation with a given probability.
    if random.random() < MUTATION_PROBABILITY:
        point = random.randint(0, len(individual) - 1)
        individual[point] = 1 - individual[point]  # Flip the binary
bit at the mutation point.
```

2.8.1 Explicación

Aplica una mutación aleatoria a un individuo para agregar variabilidad.
Si ocurre la mutación (de acuerdo a MUTATION_PROBABILITY, cambia el estado de un gen aleatorio (0 a 1 o viceversa).

2.9 Función genetic_algorithm

2.9.1 Código

```
def genetic_algorithm(dvd_size):
    # Initialize population with random individuals.
    population = [create_individual() for _ in range(NUM_INDIVIDUALS)]

    for _ in range(GENERATIONS):
        # Selection of the top individuals for reproduction.
        selected = selection(population, dvd_size)
        new_population = []
        # Generate new individuals by crossover and mutation.
        while len(new_population) < NUM_INDIVIDUALS:
            parent1 = random.choice(selected)
            parent2 = random.choice(selected)
            child1, child2 = crossover(parent1, parent2)
            mutation(child1)
            mutation(child2)
            new_population.extend([child1, child2])
        population = new_population  # Update population with the new
generation.

    # Find the best individual with the highest fitness value.
    best_individual = max(population, key=lambda ind:
calculate_fitness(ind, dvd_size))
```

```
    best_fitness = calculate_fitness(best_individual, dvd_size)

    # Print the best solution for the given DVD size.
    selected_titles = [movies[i]["title"] for i in range(len(movies))
if best_individual[i] == 1]
    print(f"Best fit for DVD of {dvd_size} GB:")
    print(f"Selected movies: {selected_titles}")
    print(f"Used space: {best_fitness:.3f} GB\n")
```

2.9.1 Explicación

Ejecuta el ciclo completo del algoritmo genético, para ello inicializa una población aleatoria de individuos, a continuación para cada generación, selecciona los mejores individuos, luego realiza cruces y mutaciones para generar una nueva población y al final de todas las generaciones, identifica el mejor individuo de la población (el que usa mejor el espacio en el DVD sin violar restricciones) y para terminar imprime las películas seleccionadas para el tamaño de DVD dado.

2.10 Ejecución por cada tamaño de DVD

2.10.1 Código

```
# Run the genetic algorithm for each DVD size.
for size in dvd_sizes:
    genetic_algorithm(size)
```

2.10.1 Explicación

Ejecuta el algoritmo genético para cada tamaño de DVD de la lista e imprime el mejor ajuste para cada uno.

3. Resultado

```
Best fit for DVD of 4.7 GB:
Selected movies: ['Wild Card', '15 Years and One Day']
Used space: 4.596 GB

Best fit for DVD of 8.5 GB:
Selected movies: ['Clash of the Titans', 'American Reunion', 'Silver Linings Playbook', '15 Years and One Day']
Used space: 8.463 GB

Best fit for DVD of 13.3 GB:
Selected movies: ['Man of Steel', 'The Avengers', 'American Reunion', 'Silver Linings Playbook', 'The Three Stooges', 'Wild Card', '15 Years and One Day']
Used space: 12.738 GB

Best fit for DVD of 15.9 GB:
Selected movies: ['Clash of the Titans', 'The Avengers', 'American Reunion', 'Silver Linings Playbook', 'The Three Stooges', 'Wild Card', '15 Years and One Day']
Used space: 13.520 GB
```

4. Enlace a Colab

5. Código completo

```
import random
import numpy as np

# List of movies with attributes: title, genre, and size in GB.
movies = [
    {"title": "The Last House on the Left", "genre": "HORROR", "size":
1.830},
    {"title": "Saw IV", "genre": "HORROR", "size": 1.435},
    {"title": "Orphan", "genre": "HORROR", "size": 2.163},
    {"title": "Clash of the Titans", "genre": "ACTION", "size": 1.746},
    {"title": "Man of Steel", "genre": "ACTION", "size": 0.964},
    {"title": "The Avengers", "genre": "ACTION", "size": 2.032},
    {"title": "American Reunion", "genre": "COMEDY", "size": 1.257},
    {"title": "Silver Linings Playbook", "genre": "COMEDY", "size":
3.139},
    {"title": "The Three Stooges", "genre": "COMEDY", "size": 0.750},
    {"title": "Wild Card", "genre": "THRILLER", "size": 2.275},
    {"title": "The Body", "genre": "THRILLER", "size": 2.082},
    {"title": "15 Years and One Day", "genre": "THRILLER", "size":
2.321}
]

# Restrictions: certain movies cannot be selected together, represented
as pairs.
movie_restrictions = [
    ("Wild Card", "The Body"),
    ("Clash of the Titans", "Man of Steel")
]
```

```
# Genre restrictions: restrict selecting both genres in a single DVD.
genre_restrictions = ("COMEDY", "HORROR")

# Available DVD sizes in GB.
dvd_sizes = [4.7, 8.5, 13.3, 15.9]

# Genetic algorithm parameters.
NUM_INDIVIDUALS = 100  # Population size
GENERATIONS = 100      # Number of generations
MUTATION_PROBABILITY = 0.1  # Probability of mutation

def calculate_fitness(individual, dvd_size):
    # Total size of selected movies.
    total_size = sum(m["size"] for i, m in enumerate(movies) if
individual[i] == 1)

    # If total size exceeds DVD capacity, return a fitness score of 0.
    if total_size > dvd_size:
        return 0

    # Check genre restrictions: no selection should include both
restricted genres.
    genres = {movies[i]["genre"] for i in range(len(movies)) if
individual[i] == 1}
    if genre_restrictions[0] in genres and genre_restrictions[1] in
genres:
        return 0

    # Check pair restrictions: restricted pairs should not be selected
together.
    for restriction in movie_restrictions:
        if any(m["title"] == restriction[0] for i, m in
enumerate(movies) if individual[i] == 1) and \
           any(m["title"] == restriction[1] for i, m in
enumerate(movies) if individual[i] == 1):
            return 0

    return total_size  # Return the total size if all restrictions are
met.

def create_individual():
```

```
    # Create a random individual (a random binary selection of movies).
    return [random.randint(0, 1) for _ in movies]

def selection(population, dvd_size):
    # Calculate fitness for each individual and sort by fitness.
    scores = [(ind, calculate_fitness(ind, dvd_size)) for ind in
population]
    scores = sorted(scores, key=lambda x: x[1], reverse=True)
    # Select top 50% individuals for mating.
    selected = [s[0] for s in scores[:NUM_INDIVIDUALS // 2]]
    return selected

def crossover(ind1, ind2):
    # Single-point crossover between two parents.
    point = random.randint(1, len(ind1) - 2)
    return ind1[:point] + ind2[point:], ind2[:point] + ind1[point:]

def mutation(individual):
    # Apply mutation with a given probability.
    if random.random() < MUTATION_PROBABILITY:
        point = random.randint(0, len(individual) - 1)
        individual[point] = 1 - individual[point]  # Flip the binary
bit at the mutation point.

def genetic_algorithm(dvd_size):
    # Initialize population with random individuals.
    population = [create_individual() for _ in range(NUM_INDIVIDUALS)]

    for _ in range(GENERATIONS):
        # Selection of the top individuals for reproduction.
        selected = selection(population, dvd_size)
        new_population = []
        # Generate new individuals by crossover and mutation.
        while len(new_population) < NUM_INDIVIDUALS:
            parent1 = random.choice(selected)
            parent2 = random.choice(selected)
            child1, child2 = crossover(parent1, parent2)
            mutation(child1)
            mutation(child2)
            new_population.extend([child1, child2])
```

```
        population = new_population  # Update population with the new
generation.

    # Find the best individual with the highest fitness value.
    best_individual = max(population, key=lambda ind:
calculate_fitness(ind, dvd_size))
    best_fitness = calculate_fitness(best_individual, dvd_size)

    # Print the best solution for the given DVD size.
    selected_titles = [movies[i]["title"] for i in range(len(movies))
if best_individual[i] == 1]
    print(f"Best fit for DVD of {dvd_size} GB:")
    print(f"Selected movies: {selected_titles}")
    print(f"Used space: {best_fitness:.3f} GB\n")

# Run the genetic algorithm for each DVD size.
for size in dvd_sizes:
    genetic_algorithm(size)
```

6. Bibliografía

https://jarroba.com/algoritmos-geneticos-ejemplo/

https://www.youtube.com/watch?v=3Kzj2FNaua8