Practica 3. Algoritmo Evolutivos: Estrategias Evolutivas y Evolución Diferencial.

Algoritmos Bioinspirados

Diego Castillo Reyes
Marthon Leobardo Yañez Martinez
Aldo Escamilla Resendiz

28 de abril de 2024

Índice general

1. Introducción

Los **Algoritmos Evolutivos** son una familia de métodos de optimización inspirados en el proceso de selección natural y evolución biológica. Dos enfoques destacados dentro de esta familia son las **Estrategias Evolutivas** y la **Evolución Diferencial**, cada uno con sus particularidades y aplicaciones.

Estrategias Evolutivas (ES)

Las Estrategias Evolutivas son técnicas que se centran principalmente en la optimización numérica continua. En estas estrategias, la población de soluciones evoluciona a través de la mutación y la selección. La mutación se realiza generalmente mediante la adición de ruido normalmente distribuido a los vectores de parámetros, lo que permite una exploración efectiva del espacio de búsqueda. Una característica distintiva de las ES es el uso de mecanismos de autoadaptación para ajustar los parámetros de la mutación, como la tasa de mutación, basándose en el éxito de las generaciones anteriores.

Evolución Diferencial (DE)

La Evolución Diferencial es otro método robusto para optimizar problemas de optimización numérica. Se caracteriza por su sencillez y eficacia, especialmente en problemas multimodales (con múltiples óptimos locales). En DE, la nueva generación se crea añadiendo la diferencia ponderada entre dos o más soluciones de la población actual a una tercera solución. Este método se basa en operadores simples como la mutación diferencial, la recombinación y la selección. La mutación diferencial es particularmente útil para mantener la diversidad genética dentro de la población, lo que ayuda a explorar de manera efectiva el espacio de búsqueda.

Ambos métodos, aunque similares en su inspiración evolutiva, difieren en sus mecanismos de mutación y adaptación, lo que los hace adecuados para diferentes tipos de problemas de optimización. La elección entre Estrategias Evolutivas y Evolución Diferencial a menudo depende del problema específico, la naturaleza del espacio de búsqueda y las preferencias del investigador o ingeniero.

2. Desarrollo

Para el desarrollo de esta práctica se programaron los siguientes codigos.

2.1 Estrategia Evolutiva (μ, λ)

Usa una estrategia evolutiva en la que solo los descendientes (no los padres) son considerados para la generación siguiente, lo que puede ayudar a evitar la convergencia prematura hacia mínimos locales subóptimos.

```
# La estrategia evolutiva de este programa es (  ,   ) donde solo los descendientes son considerados para la siguiente generacion.
from matplotlib import pyplot as plt
import numpy as np
from functools import reduce
from operator import mul

# Funciones a optimizar
def rosenbrock(x):
    result = 0
    for i in range(len(x) - 1):
        result += 100 * ((x[i] ** 2) - x[i + 1]) ** 2 + (1 - x[i]) ** 2
    return result

def ackley(x):
    summation1 = sum(xi**2 for xi in x)
    summation2 = sum(np.cos(2 * np.pi * xi) for xi in x)
    exp1 = np.exp(-0.2 * np.sqrt((1 / len(x)) * summation1))
    exp2 = np.exp((1 / len(x)) * summation2)
    result = (- 20) * exp1 - exp2 + np.e + 20
    return result

def griewank(x):
    summation = sum(xi**2 for xi in x)
    producer = reduce(mul, np.cos(x / np.sqrt(np.arange(1, len(x) + 1))))
    result = (1 / 4000) * summation - producer + 1
    return result

```

```python
def rastrigin(x):
    summation = sum(xi**2 - 10 * np.cos(2 * np.pi * xi) for xi in x)
    result = summation + 10 * len(x)
    return result


# Estrategias evolutivas
def randSolution(interval, dimension):
    limInf, limSup = interval
    solution = np.round(np.random.uniform(limInf, limSup, dimension), 2)
    return solution

def clip(x, interval): # revisa los limites de las variables de decision
    limInf, limSup = interval
    return max(min(x, limSup), limInf)

def mutation(sigmaT, x, interval):
    newX = []
    for i in range(len(x)):
        newX.append(clip(np.round(x[i] + sigmaT * np.random.normal(0, 1), 2), interval))
    return newX

def crossover(parents, dimension):
    newX = []
    for i in range(dimension):
        index = np.random.randint(0, len(parents))
        newX.append(parents[index][i])
    return newX


def estrategiaEvolutiva(Gmax, dimension, interval, fun, mu, lamb, c, sigma):
    x = randSolution(interval, dimension) #solucion base/solución inicial (la podria tomar del conocimiento del problema)
    fx = fun(x)
    print('SOLUCION INICIAL', x, fx)
    bestSolution = []
    sigmas = []
    successes = 0
    ps = 0


    for gen in range(Gmax): # Numero maximo de generaciones
        if gen == 0:
            # Crea    padres
            parents = []
            for _ in range(mu):
                individual = mutation(sigma, x, interval)
                fitness = np.round(fun(individual), 4)
                parents.append([individual, fitness])
            print(f"Padres: {parents}")
        else:
            # Selecciona los nuevos padres a partir de los hijos generados en la generación anterior
            # Ordena los hijos por su fitness y selecciona los mejores    para ser padres
            parents = sorted(offspring, key=lambda child: child[1])[:mu]

        # Crear    hijos a partir de los    padres
        offspring = []
        for _ in range(lamb):
            # Seleccionar aleatoriamente a dos padres para el cruce
            # Generar índices aleatorios
            parent_indices = np.random.choice(len(parents), 2, replace=False)
            # Usar índices para obtener los individuos
            p1, p2 = parents[parent_indices[0]][0], parents[parent_indices[1]][0]
            child = crossover([p1, p2], dimension)
            child = mutation(sigma, child, interval)
            offspring.append([child, np.round(fun(child), 4)])

            # Actualizar la mejor solución
            if offspring[-1][1] < fun(x):
                x = offspring[-1][0]
```

```
                successes += 1

        print(f"Generacion {gen} Descendientes:\n{offspring}")

        bestSolution.append(fun(x))
        sigmas.append(sigma)

        #Actualizar ps: frecuencia relativa de mutaciones exitosas.
        if gen % (10 * dimension) == 0: # calcula la proporción de éxito cada 10*n generaciones
            ps = successes / (gen + 1)
            #if gen%n == 0: # n mas grande  ps se mantiene mas generaciones
            if ps > 1/5:
                sigma = sigma / c # no hay tantos éxitos por lo tanto explora regiones con tamaños de paso más grande
            elif ps < 1/5:
                sigma = sigma * c  # encuentra región prometedora por lo tanto refina la solución actual(explotacion)
            elif ps == 1/5: #caso contrario sigma queda con el mismo valor
                sigma = sigma

    return bestSolution, sigmas

Gmax = 1000
np.random.seed(123)
dimension = 10
fun = rosenbrock

interval = (-2.048, 2.048) if fun == rosenbrock else \
           (-32.768, 32.768) if fun == ackley else \
           (-600, 600) if fun == griewank else \
           (-5.12, 5.12) if fun == rastrigin else None

mu = 20
lamb = 30 # Debe ser mayor que mu
sigma = 0.5 if fun == rosenbrock else \
           2.0 if fun == ackley else \
           20 if fun == griewank else \
           1.0 if fun == rastrigin else None
c = 0.817

best, sigmas = estrategiaEvolutiva(Gmax, dimension, interval, fun, mu, lamb, c, sigma)

print('BEST', best)
print('SIGMAS', sigmas)
plt.plot(range(0, len(best)), best, color = 'green', label='mejores')
plt.legend()
plt.show()
plt.plot(range(0, len(sigmas)), sigmas, label='sigmas')
plt.legend()

plt.show()
```

Este código es una implementación directa de una estrategia evolutiva que facilita la optimización en espacios de búsqueda complejos mediante la adaptación continua de los parámetros y el uso de operadores genéticos como la mutación y el crossover.

2.2 Estrategia Evolutiva $(\mu + \lambda)$

```
# La estrategia evolutiva de este programa es (   +   ) donde solo los descendientes son considerados para la siguiente generacion.
from matplotlib import pyplot as plt
import numpy as np
from functools import reduce
from operator import mul

# Funciones a optimizar
def rosenbrock(x):
    result = 0
    for i in range(len(x) - 1):
        result += 100 * ((x[i] ** 2) - x[i + 1]) ** 2 + (1 - x[i]) ** 2
```

```
    return result

def ackley(x):
    summation1 = sum(xi**2 for xi in x)
    summation2 = sum(np.cos(2 * np.pi * xi) for xi in x)
    exp1 = np.exp(-0.2 * np.sqrt((1 / len(x)) * summation1))
    exp2 = np.exp((1 / len(x)) * summation2)
    result = (- 20) * exp1 - exp2 + np.e + 20
    return result

def griewank(x):
    summation = sum(xi**2 for xi in x)
    producer = reduce(mul, np.cos(x / np.sqrt(np.arange(1, len(x) + 1))))
    result = (1 / 4000) * summation - producer + 1
    return result

def rastrigin(x):
    summation = sum(xi**2 - 10 * np.cos(2 * np.pi * xi) for xi in x)
    result = summation + 10 * len(x)
    return result


# Estrategias evolutivas
def randSolution(interval, dimension):
    limInf, limSup = interval
    solution = np.round(np.random.uniform(limInf, limSup, dimension), 2)
    return solution

def clip(x, interval): # revisa los limites de las variables de decision
    limInf, limSup = interval
    return max(min(x, limSup), limInf)

def mutation(sigmaT, x, interval):
    newX = []
    for i in range(len(x)):
        newX.append(clip(np.round(x[i] + sigmaT * np.random.normal(0, 1), 2), interval))
    return newX

def crossover(parents, dimension):
    newX = []
    for i in range(dimension):
        index = np.random.randint(0, len(parents))
        newX.append(parents[index][i])
    return newX


def estrategiaEvolutiva(Gmax, dimension, interval, fun, mu, lamb, c, sigma):
    x = randSolution(interval, dimension) #solucion base/solución inicial (la podria tomar del
    conocimiento del problema)
    fx = fun(x)
    print('SOLUCION INICIAL', x, fx)
    bestSolution = []
    sigmas = []
    successes = 0
    ps = 0


    for gen in range(Gmax): # Numero maximo de generaciones
        if gen == 0:
            # Crea    padres
            parents = []
            for _ in range(mu):
                individual = mutation(sigma, x, interval)
                fitness = np.round(fun(individual), 4)
                parents.append([individual, fitness])
            print(f"Padres: {parents}")
        else:
            # Selecciona los nuevos padres a partir de los mejores individuos en la generación
    anterior
            # Ordena los individuos por su fitness y selecciona los mejores    para ser padres
            parents = sorted(population, key=lambda child: child[1])[:mu]
```

```
81
82          # Crear    hijos a partir de los    padres
83          offspring = []
84          for _ in range(lamb):
85              # Seleccionar aleatoriamente a dos padres para el cruce
86              # Generar índices aleatorios
87              parent_indices = np.random.choice(len(parents), 2, replace=False)
88              # Usar índices para obtener los individuos
89              p1, p2 = parents[parent_indices[0]][0], parents[parent_indices[1]][0]
90              child = crossover([p1, p2], dimension)
91              child = mutation(sigma, child, interval)
92              offspring.append([child, np.round(fun(child), 4)])
93
94          population = parents + offspring
95          for _ in range(len(population)):
96              # Actualizar la mejor solución
97              if population[-1][1] < fun(x):
98                  x = population[-1][0]
99                  successes += 1
100
101         print(f"Generacion {gen} Descendientes:\n{offspring}")
102
103         bestSolution.append(fun(x))
104         sigmas.append(sigma)
105
106         #Actualizar ps: frecuencia relativa de mutaciones exitosas.
107         if gen % (10 * dimension) == 0: # calcula la proporción de éxito cada 10*n generaciones
108             ps = successes / (gen + 1)
109             #if gen%n == 0: # n mas grande  ps se mantiene mas generaciones
110             if ps > 1/5:
111                 sigma = sigma / c # no hay tantos éxitos por lo tanto explora regiones con tamaños
     de paso más grande
112             elif ps < 1/5:
113                 sigma = sigma * c  # encuentra región prometedora por lo tanto refina la solución
    actual(explotacion)
114             elif ps == 1/5: # caso contrario sigma queda con el mismo valor
115                 sigma = sigma
116
117     return bestSolution, sigmas
118
119 Gmax = 1000
120 np.random.seed(38)
121 dimension = 10
122 fun = rosenbrock
123
124 interval = (-2.048, 2.048) if fun == rosenbrock else \
125             (-32.768, 32.768) if fun == ackley else \
126             (-600, 600) if fun == griewank else \
127             (-5.12, 5.12) if fun == rastrigin else None
128
129 mu = 20
130 lamb = 30
131 sigma = 0.5 if fun == rosenbrock else \
132             2.0 if fun == ackley else \
133             20 if fun == griewank else \
134             1.0 if fun == rastrigin else None
135 c = 0.817
136
137 best, sigmas = estrategiaEvolutiva(Gmax, dimension, interval, fun, mu, lamb, c, sigma)
138
139 print('BEST', best)
140 print('SIGMAS', sigmas)
141 plt.plot(range(0, len(best)), best, color = 'green', label='mejores')
142 plt.legend()
143 plt.show()
144 plt.plot(range(0, len(sigmas)), sigmas, label='sigmas')
145 plt.legend()
146
147 plt.show()
```

2.3 Evolución Diferencial

```python
import numpy as np
from functools import reduce
from operator import mul
import os
# import estrategiaEvolutiva1 as ee1

def clear_screen():
    # Comprueba si el sistema operativo es Windows
    if os.name == 'nt':
        os.system('cls')  # cls es el comando para limpiar la consola en Windows
    else:
        os.system('clear')  # clear es el comando para limpiar la consola en Unix/Linux

# Funciones a optimizar
def rosenbrock(x):
    result = 0
    for i in range(len(x) - 1):
        result += 100 * ((x[i] ** 2) - x[i + 1]) ** 2 + (1 - x[i]) ** 2
    return result

def ackley(x):
    summation1 = sum(xi**2 for xi in x)
    summation2 = sum(np.cos(2 * np.pi * xi) for xi in x)
    exp1 = np.exp(-0.2 * np.sqrt((1 / len(x)) * summation1))
    exp2 = np.exp((1 / len(x)) * summation2)
    result = (- 20) * exp1 - exp2 + np.e + 20
    return result

def griewank(x):
    summation = sum(xi**2 for xi in x)
    producer = reduce(mul, np.cos(x / np.sqrt(np.arange(1, len(x) + 1))))
    result = (1 / 4000) * summation - producer + 1
    return result

def rastrigin(x):
    summation = sum(xi**2 - 10 * np.cos(2 * np.pi * xi) for xi in x)
    result = summation + 10 * len(x)
    return result


# Definir el algoritmo de Evolución Diferencial
def differential_evolution(strategy, objective_function, dimensions, population_size, F, R, Gmax,
    L, U):
    # Ejemplo para 'rand/1/bin':
    if strategy == 'rand/1/bin':
        X, best = rand_1_bin(objective_function, dimensions, population_size, F, R, Gmax, L, U)
        print(X)
        print(f"Best: {best}")

    # Ejemplo para 'rand/1/exp':
    elif strategy == 'rand/1/exp':
        X, best = rand_1_exp(objective_function, dimensions, population_size, F, R, Gmax, L, U)
        print(X)
        print(f"Best: {best}")

    # Ejemplo para 'best/1/bin':
    elif strategy == 'best/1/bin':
        X, best = best_1_bin(objective_function, dimensions, population_size, F, R, Gmax, L, U)
        print(X)
        print(f"Best: {best}")

    # Ejemplo para 'best/1/exp':
    elif strategy == 'best/1/exp':
        X, best = best_1_exp(objective_function, dimensions, population_size, F, R, Gmax, L, U)
        print(X)
        print(f"Best: {best}")

    return

```

```python
#crea poblacion dentro de los intervalos de la funcion objetivo
def start_poblation(population_size, dimension, lower_limit, upper_limit):
    poblation = []

    for _ in range(population_size):
        individual = []
        for _ in range(dimension):
            individual.append(np.random.uniform(lower_limit, upper_limit))

        poblation.append(individual)

    return poblation

# Generacion de r1 != r2 != r3
def random_numbers(population_size, n):
    indexes = np.random.randint(0, population_size, n)
    while len(set(indexes)) != n:
        indexes = np.random.randint(0, population_size, n)
    return indexes

def clip(x, interval): # revisa los limites de las variables de decision
    limInf, limSup = interval
    return max(min(x, limSup), limInf)

# Ejecutar el experimento
def run_experiment(strategy, objective_function, dimensions, population_size, F, R, Gmax, runs, L,
     U):
    for _ in range(runs):
        differential_evolution(strategy, objective_function, dimensions, population_size, F, R,
    Gmax, L, U)
    return

def rand_1_bin(objective_function, dimensions, population_size, F, R, Gmax, L, U):
    X = start_poblation(population_size, dimensions, L, U)
    print("============================Rand/1/bin============================")
    print(f"==================Funcion objetivo: {objective_function.__name__}==================
    ")
    print("========================Poblacion inicial========================")
    print(f"{X} tamaño: {len(X)}")
    best_individual = min(X, key=objective_function)
    best = objective_function(best_individual)
    for g in range(Gmax):
        for i in range(population_size):
            r1, r2, r3 = random_numbers(population_size, 3)
            jrand = np.random.randint(0, dimensions)
            ui = []
            for j in range(dimensions):
                if np.random.rand() < R or j == jrand:
                    u = X[r1][j] + F * (X[r2][j] - X[r3][j])
                    u = clip(u, (L, U))
                    ui.append(u)
                else:
                    ui.append(X[i][j])

            if objective_function(ui) <= objective_function(X[i]):
                X[i] = ui

            if objective_function(X[i]) < best:
                best = objective_function(X[i])
                best_individual = X[i]
        print(f"Generacion {g} Mejor individuo: {best_individual} Fitness: {best}")

    return X, best

def rand_1_exp(objective_function, dimensions, population_size, F, R, Gmax, L, U):
    X = start_poblation(population_size, dimensions, L, U)
    print("============================Rand/1/exp============================")
    print(f"==================Funcion objetivo: {objective_function.__name__}==================
    ")
    print("========================Poblacion inicial========================")
    print(f"{X} tamaño: {len(X)}")
```

```
    best_individual = min(X, key=objective_function)
    best = objective_function(best_individual)
    for g in range(Gmax):
        for i in range(population_size):
            r1, r2, r3 = random_numbers(population_size, 3)
            jrand = np.random.randint(0, dimensions)
            ui = X[i][:]
            j = 0
            # n = 0
            while True:
                u = X[r1][j] + F * (X[r2][j] - X[r3][j])
                u = clip(u, (L, U))
                ui[j] = u
                j += 1 #% dimensions  # Incrementa j de manera circular
                # jrand = (jrand + 1) % dimensions  # Asegura que al menos uno cambie
                # n += 1
                if j < dimensions and np.random.rand() < R or j == jrand:
                    break

            if objective_function(ui) <= objective_function(X[i]):
                X[i] = ui

            if objective_function(X[i]) < best:
                best = objective_function(X[i])
                best_individual = X[i]
        print(f"Generacion {g} Mejor individuo: {best_individual} Fitness: {best}")

    return X, best

def best_1_bin(objective_function, dimensions, population_size, F, R, Gmax, L, U):
    X = start_poblation(population_size, dimensions, L, U)
    print("============================Best/1/bin============================")
    print(f"==================Funcion objetivo: {objective_function.__name__}==================
    ")
    print("========================Poblacion inicial========================")
    print(f"{X} tamaño: {len(X)}")
    best_individual = min(X, key=objective_function)
    best = objective_function(best_individual)
    for g in range(Gmax):
        for i in range(population_size):
            r2, r3 = random_numbers(population_size, 2)
            jrand = np.random.randint(0, dimensions)
            ui = []
            for j in range(dimensions):
                if np.random.rand() < R or j == jrand:
                    u = best_individual[j] + F * (X[r2][j] - X[r3][j])
                    u = clip(u, (L, U))
                    ui.append(u)
                else:
                    ui.append(X[i][j])

            if objective_function(ui) <= objective_function(X[i]):
                X[i] = ui

            if objective_function(X[i]) < best:
                best = objective_function(X[i])
                best_individual = X[i]
        print(f"Generacion {g} Mejor individuo: {best_individual} Fitness: {best}")

    return X, best

def best_1_exp(objective_function, dimensions, population_size, F, R, Gmax, L, U):
    X = start_poblation(population_size, dimensions, L, U)
    print("============================Best/1/exp============================")
    print(f"==================Funcion objetivo: {objective_function.__name__}==================
    ")
    print("========================Poblacion inicial========================")
    print(f"{X} tamaño: {len(X)}")
    best_individual = min(X, key=objective_function)
    best = objective_function(best_individual)
    for g in range(Gmax):
```

```
        for i in range(population_size):
            r2, r3 = random_numbers(population_size, 2)
            jrand = np.random.randint(0, dimensions)
            ui = X[i][:]
            j = 0
            # n = 0
            while True:
                u = best_individual[j] + F * (X[r2][j] - X[r3][j])
                u = clip(u, (L, U))
                ui[j] = u
                j = (j + 1) #% dimensions  # Incrementa j de manera circular
                # jrand = (jrand + 1) % dimensions  # Asegura que al menos uno cambie
                # n += 1
                if np.random.rand() < R or j == jrand:
                    break

            if objective_function(ui) <= objective_function(X[i]):
                X[i] = ui

            if objective_function(X[i]) < best:
                best = objective_function(X[i])
                best_individual = X[i]
        print(f"Generacion {g} Mejor individuo: {best_individual} Fitness: {best}")

    return X, best

# Configuración de parámetros según la imagen
population_size = 50
dimensions = 10
F = 0.6
R = 0.9
Gmax = 1000
runs = 1

np.random.seed(7)

# Estrategias de evolución diferencial
strategies = ['rand/1/bin', 'rand/1/exp', 'best/1/bin', 'best/1/exp']

# Funciones objetivo
objective_functions = [rosenbrock, ackley, griewank, rastrigin]

# Correr los experimentos
for strategy in strategies:
    for objective_function in objective_functions:
        if objective_function == rosenbrock:
            interval = (-2.048, 2.048)
        elif objective_function == ackley:
            interval = (-32.768, 32.768)
        elif objective_function == griewank:
            interval = (-600, 600)
        elif objective_function == rastrigin:
            interval = (-5.12, 5.12)
        lower_limit, upper_limit = interval
        run_experiment(strategy, objective_function, dimensions, population_size, F, R, Gmax, runs
    , lower_limit, upper_limit)
        input("Presione Enter para continuar...")
        clear_screen()
```

¿Cuáles fueron las versiones de algoritmos que no convergieron a la solución optima?
En la función Griewank, todas las ejecuciones tienden a converger hacia soluciones con valores de fitness relativamente bajos. Esto sugiere que no existen evidencias claras que indiquen la falta de convergencia en alguna de las variantes del algoritmo evaluadas. Por tanto, aunque el rendimiento puede no ser óptimo, todos los enfoques del algoritmo parecen alcanzar un cierto nivel de estabilidad en sus resultados.
¿A qué versión de estretegia evolutiva le fue mejor?
En la función Rastrigin, todas las estrategias evolutivas exhiben un rendimiento parecido, convergiendo hacia soluciones con valores de fitness bajos. Sin embargo, se observa una excepción notable en la segunda ejecución utilizando la Evolución Diferencial con la estrategia (rand/1/exp), que logró alcanzar un fitness significativamente superior. Esto sugiere que esta configuración particular del algoritmo puede ser más efectiva bajo ciertas condiciones, des-

tacándose del resto en términos de eficacia.
¿Qué versión de evolucion diferencial fue la mejor e indique por que cree que le fue mejor (en qué problemas).
En la función Rastrigin, las estrategias de Evolución Diferencial (DE) utilizando tanto (rand/1/bin) como (best/1/bin) mostraron un desempeño destacado, logrando en varias ejecuciones converger hacia soluciones con un fitness cercano a cero, que representa el óptimo para esta función. Esto indica que estas estrategias podrían ser especialmente efectivas para abordar problemas que presentan múltiples óptimos locales, como es el caso de Rastrigin, debido a su eficiente balance entre exploración y explotación del espacio de búsqueda.

Por otro lado, en la función Griewank, todas las estrategias evaluadas parecen converger hacia soluciones con fitness bajo, lo que no permite identificar diferencias significativas en el rendimiento entre las distintas variantes de evolución diferencial aplicadas. Sin embargo, la estrategia DE (best/1/exp) mostró un rendimiento ligeramente superior en algunas de las ejecuciones, sugiriendo que podría tener ciertas ventajas en determinadas configuraciones o escenarios de este problema.

Considerando todas las pruebas, cuál es el algoritmo que converge mas rápido (respecto a suvalor de media)
El algoritmo de Evolución Diferencial (DE) utilizando la estrategia (rand/1/bin) se destaca por su rapidez en la convergencia, especialmente cuando se aplica a la función Rastrigin. En múltiples ejecuciones, este algoritmo ha demostrado su capacidad para converger eficientemente hacia soluciones con un fitness cercano a cero, el cual es el valor óptimo para esta función. Esta eficacia en alcanzar rápidamente soluciones de alta calidad lo convierte en una opción atractiva para problemas de optimización complejos con múltiples óptimos locales.

Cuadro 1: Comparación de estrategias en diferentes funciones

Problema	**(m, λ)**	**(m + λ)**	**(r/1/b)**	**(r/1/e)**	**(b/1/b)**	**(b/1/e)**
Ackley	–	–	–	–	–	–
Griewank	–	–	–	–	–	–
Rastrigin	–	–	–	–	–	–
Rosenbrock	–	–	–	–	–	–

3. Resultados

4. Conclusiones

Escamilla Resendiz Aldo
Los algoritmos de Evolución Diferencial (DE) han mostrado ser herramientas potentes y versátiles para abordar funciones de optimización complejas como Rastrigin y Griewank. Específicamente, la estrategia DE (rand/1/bin) se ha destacado por su rapidez en la convergencia hacia el óptimo en la función Rastrigin, mientras que las estrategias DE (rand/1/bin) y DE (best/1/bin) también han demostrado ser efectivas en enfrentar problemas con múltiples óptimos locales gracias a su capacidad para explorar y explotar eficientemente el espacio de búsqueda.
Castillo Reyes Diego
Todas las estrategias tendieron a converger a soluciones de baja fitness en la función Griewank, la DE (best/1/exp) ha mostrado un rendimiento ligeramente superior en ciertas ejecuciones. Esto sugiere que la elección de la estrategia DE puede ser crucial dependiendo de la naturaleza específica del problema de optimización a resolver.
Yañez Martinez Marthon
la importancia de elegir la estrategia adecuada de DE, adaptada a las características específicas del problema, para lograr un equilibrio óptimo entre exploración y explotación.