Trabajo Práctico N°3

Algoritmos Evolutivos (2024)

Tema: PSO con restricciones

Alumno: Paz, Martin

Link del repositorio: https://github.com/freischarler/ceia_algevo/tree/main/tp3

Ejercicio 1:

1.A - Algoritmo: https://github.com/freischarler/ceia_algevo/blob/main/tp3/a.py

```python
# Parámetros del problema
fabrication_capacity = 640
finishing_capacity = 960

# Función objetivo a maximizar
def f(x):
    return 375 * x[0] + 275 * x[1] + 475 * x[2] + 325 * x[3]

# Restricciones
def g1(x):
    return 2.5 * x[0] + 1.5 * x[1] + 2.75 * x[2] + 2 * x[3] <= fabrication_capacity

def g2(x):
    return 3.5 * x[0] + 3 * x[1] + 3 * x[2] + 2 * x[3] <= finishing_capacity

# Parámetros del PSO
n_particles = 20 # numero de particulas en el enjambre
n_dimensions = 4 # dimensiones del espacio de busqueda (x1 y x2)
max_iterations = 50 # numero máximo de iteraciones para la optimizacion
c1 = c2 = 1.4944 # coeficientes de aceleracion
w = 0.6  # Factor de inercia

def pso_optimization_with_constraints(constraints):
    # matriz para las posiciones de las particulas
    x = np.zeros((n_particles, n_dimensions))
    # matriz para las velocidades de las particulas
    v = np.zeros((n_particles, n_dimensions))
    # matriz para los mejores valores personales
    pbest = np.zeros((n_particles, n_dimensions))
    # vector para las mejores aptitudes personales (inicialmente -infinito)
    pbest_fit = -np.inf * np.ones(n_particles)
    # mejor solución global
    gbest = np.zeros(n_dimensions)
    # mejor aptitud global (inicialmente -infinito)
    gbest_fit = -np.inf

    # inicializacion de particulas factibles
    for i in range(n_particles):
        while True:  # bucle para asegurar que la particula sea factible
            # inicializacion posicion aleatoria en el rango [0, 10]
            x[i] = np.random.uniform(0, 10, n_dimensions)
            # se comprueba si la posicion cumple las restricciones
            if g1(x[i]) and g2(x[i]):
                break  # Salir del bucle si es factible
        # inicializar velocidad aleatoria
        v[i] = np.random.uniform(-1, 1, n_dimensions)
        # se establece el mejor valor personal inicial como la posicion actual
        pbest[i] = x[i].copy()
        fit = f(x[i])  # calculo la aptitud de la posicion inicial
        if fit > pbest_fit[i]:  # si la aptitud es mejor que la mejor conocida
            pbest_fit[i] = fit  # se actualiza el mejor valor personal

    # Optimización
    for _ in range(max_iterations):
        for i in range(n_particles):
            fit = f(x[i])
            if fit > pbest_fit[i] and g1(x[i]) and g2(x[i]):
                pbest_fit[i] = fit
                pbest[i] = x[i].copy()
                if fit > gbest_fit:
                    gbest_fit = fit
                    gbest = x[i].copy()

            # Actualización de la velocidad y posición de la partícula
```

```
            v[i] = w * v[i] + c1 * np.random.rand() * (pbest[i] - x[i]) + c2 *
np.random.rand() * (gbest - x[i])
            x[i] += v[i]

            # Asegurar que la posición esté dentro de las restricciones y no negativa
            if not (g1(x[i]) and g2(x[i])) or np.any(x[i] < 0):
                x[i] = pbest[i].copy()
```

1.B - Solución óptima y valor objetivo óptimo:
https://github.com/freischarler/ceia_algevo/blob/main/tp3/b.py

```
# Llamada a la función de optimización
gbest, gbest_fit = pso_optimization()

# Se imprime la mejor solucion encontrada y también su valor optimo
print(f"Mejor solucion: [{gbest[0]:.4f}, {gbest[1]:.4f}, {gbest[2]:.4f}, {gbest[3]:.4f}]")
print(f"Valor optimo: {gbest_fit}")
```

Mejor solucion: [85.4831, 85.7415, 83.1323, 34.5331]
Valor optimo: 106346.1718373614

Esto sería:

- 85.48 unidades del producto A
- 85.74 unidades del producto B
- 83.13 unidades del producto C
- 34.53 unidades del producto D

Para producir una utilidad de $ 106346.17

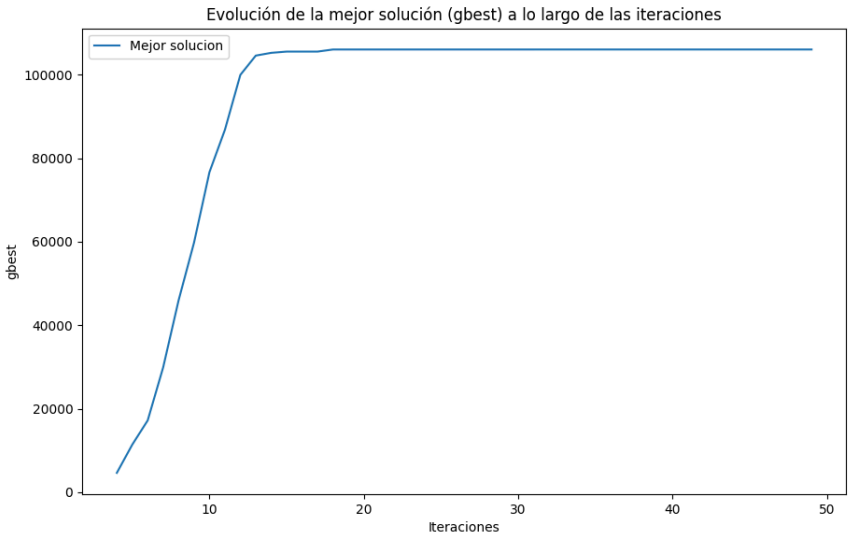
1.C - URL del repositorio: https://github.com/freischarler/ceia_algevo/blob/main/tp3/

1.D - Grafico gbest: https://github.com/freischarler/ceia_algevo/blob/main/tp3/d.py

Para obtener un gráfico de gbest según las iteraciones, dentro del algoritmo realizamos lo siguiente:

```
#creamos un vector para guardar los gbest/iter
gbest_history = []
…
gbest_history.append(gbest_fit)
```

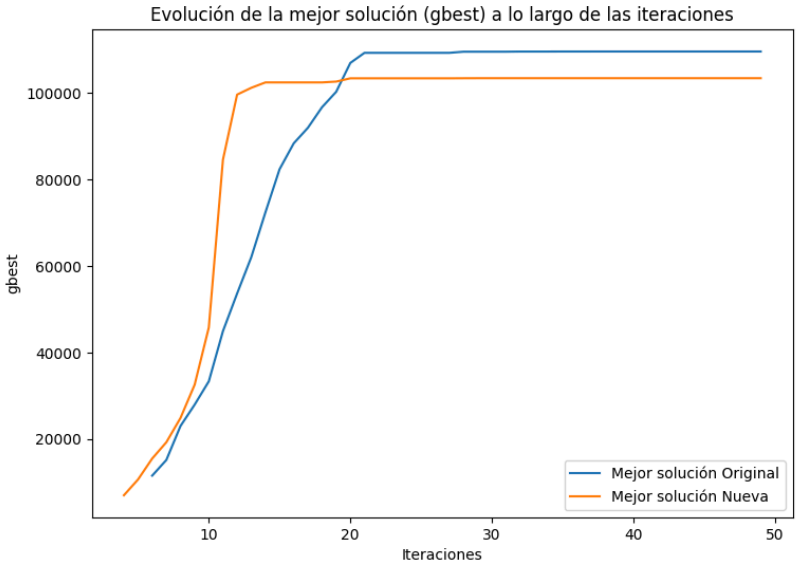
Luego graficamos con pyplot



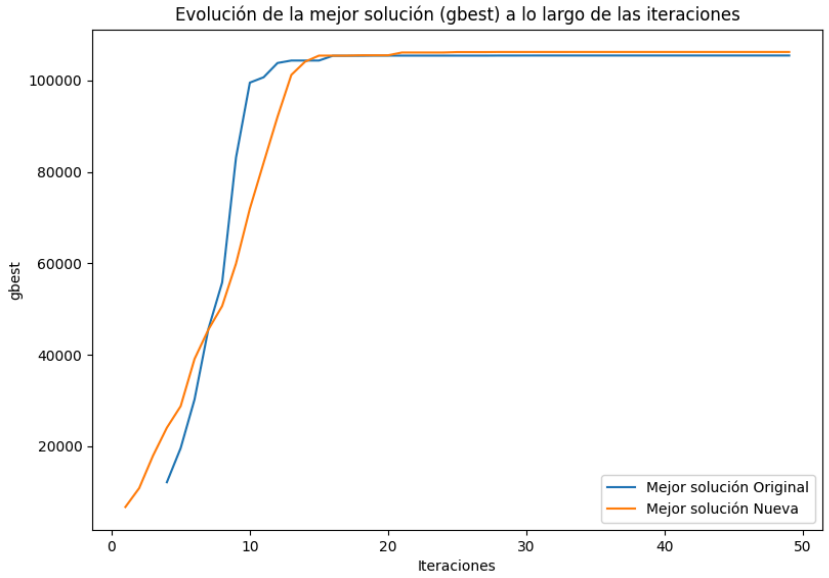
1.E - Reduccion: https://github.com/freischarler/ceia_algevo/blob/main/tp3/e.py

Al reducir una unidad de tiempo, creamos una nueva restricción y corremos el algoritmo y vemos la diferencia:

Mejor solución: [40.27466184 168.36461444 57.6868606   64.06376802]
Valor óptimo: 109625.25055225167
Mejor solución con nueva restricción: [151.96687739 106.65078247 31.00249837   7.42488095]
Valor óptimo con nueva restricción: 103455.81723291895
Reducción en el valor óptimo: 6169.433319332718
Porcentaje de reducción: 5.63%



Mejor solución: [77.79463732 79.12530476 50.2115278 94.37187362]
Valor óptimo: 105453.7824388942
Mejor solución con nueva restricción: [83.12028324 65.44764801 98.93153021 30.98305565]
Valor óptimo con nueva restricción: 106230.17935542625
Reducción en el valor óptimo: -776.3969165320595
Porcentaje de reducción: -0.74%

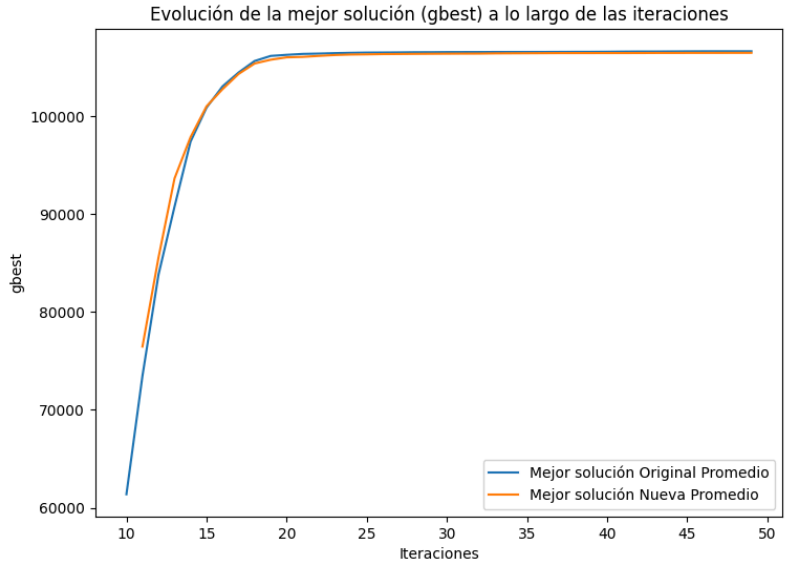


Vemos resultados con y sin reducción de valor óptimo, esto es, debido a que PSO utiliza inicializaciones aleatorias y actualizaciones basadas en valores aleatorios, los resultados pueden variar significativamente entre ejecuciones.

Si realizamos varias ejecuciones y luego realizamos un promedio podemos ver lo siguiente:

```
n_runs = 50  # Número de ejecuciones para promediar
```

Mejor solución promedio: [63.56820905 71.20617848 86.53613872 68.13155359]
Valor óptimo promedio: 106667.19828521009
Mejor solución promedio con nueva restricción: [58.06510829 74.64081623 86.42813146 71.23599594]
Valor óptimo promedio con nueva restricción: 106505.70119669728
Reducción en el valor óptimo promedio: 161.4970885128132
Porcentaje de reducción promedio: 0.15%

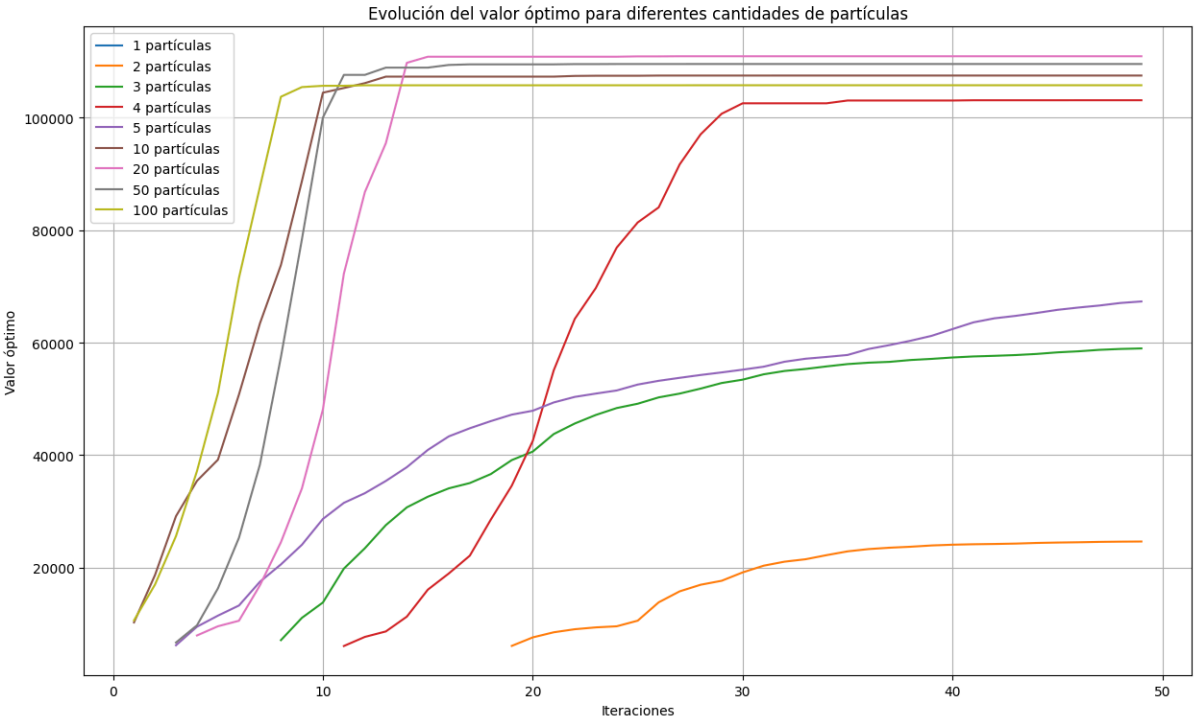


El resultado muestra que la reducción en el valor óptimo promedio es muy pequeña (0.15%), lo que indica que la nueva restricción no tiene un impacto significativo en la solución óptima. Esto puede deberse a que la nueva restricción no es mucho más restrictiva que las restricciones originales, o que el espacio de búsqueda ya estaba bien explorado por el algoritmo.

1.F - Variacion de particulas: https://github.com/freischarler/ceia_algevo/blob/main/tp3/f.py

Vamos a realizar varias simulaciones con diferente partículas

```
# Valores de partículas a probar
particle_counts = [1, 2, 3, 4, 5, 10, 20, 50, 100]
```



En la gráfica vemos que, a partir de 10 partículas, se llega a un rango de resultados óptimos con bajas iteraciones (respecto a menos partículas).

Un enjambre inferior, puede no explorar suficientemente el espacio de búsqueda y puede quedar atrapado en óptimos locales. En cambio a partir de un cierto número de partículas (en este caso, alrededor de 10), el equilibrio entre exploración y explotación mejora, permitiendo al enjambre encontrar soluciones más cercanas al óptimo global. Además, con más partículas, la varianza en los resultados tiende a disminuir, esto significa que los resultados son más consistentes y menos dependientes de las posiciones iniciales de las partículas.

ANEXO ARCHIVOS DE FUENTE

Ejercicio 1.F

```
import numpy as np
import matplotlib.pyplot as plt

# Parámetros del problema
fabrication_capacity = 640
finishing_capacity = 960

# Función objetivo a maximizar
def f(x):
    return 375 * x[0] + 275 * x[1] + 475 * x[2] + 325 * x[3]

# Restricciones
def g1(x):
    return 2.5 * x[0] + 1.5 * x[1] + 2.75 * x[2] + 2 * x[3] <= fabrication_capacity

def g2(x):
    return 3.5 * x[0] + 3 * x[1] + 3 * x[2] + 2 * x[3] <= finishing_capacity

# Parámetros del PSO
n_dimensions = 4  # dimensiones del espacio de búsqueda (x1 y x2)
max_iterations = 50  # número máximo de iteraciones para la optimización
c1 = c2 = 1.4944  # coeficientes de aceleración
w = 0.6  # Factor de inercia

def pso_optimization_with_constraints(n_particles, constraints):
    x = np.zeros((n_particles, n_dimensions))  # matriz para las posiciones de las
partículas
    v = np.zeros((n_particles, n_dimensions))  # matriz para las velocidades de las
partículas
    pbest = np.zeros((n_particles, n_dimensions))  # matriz para los mejores valores
personales
    pbest_fit = -np.inf * np.ones(n_particles)  # vector para las mejores aptitudes
personales (inicialmente -infinito)
    gbest = np.zeros(n_dimensions)  # mejor solución global
    gbest_fit = -np.inf  # mejor aptitud global (inicialmente -infinito)

    # inicialización de partículas factibles
    for i in range(n_particles):
        while True:  # bucle para asegurar que la partícula sea factible
            x[i] = np.random.uniform(0, 10, n_dimensions)  # inicialización posición
aleatoria en el rango [0, 10]
            if g1(x[i]) and g2(x[i]):  # se comprueba si la posición cumple las
restricciones
                break  # Salir del bucle si es factible
        v[i] = np.random.uniform(-1, 1, n_dimensions)  # inicializar velocidad aleatoria
```

```
        pbest[i] = x[i].copy()  # se establece el mejor valor personal inicial como la
posición actual
        fit = f(x[i])  # cálculo la aptitud de la posición inicial
        if fit > pbest_fit[i]:  # si la aptitud es mejor que la mejor conocida
            pbest_fit[i] = fit  # se actualiza el mejor valor personal

    # Optimización
    gbest_history = []
    for _ in range(max_iterations):
        for i in range(n_particles):
            fit = f(x[i])
            if fit > pbest_fit[i] and g1(x[i]) and g2(x[i]):
                pbest_fit[i] = fit
                pbest[i] = x[i].copy()
                if fit > gbest_fit:
                    gbest_fit = fit
                    gbest = x[i].copy()

            # Actualización de la velocidad y posición de la partícula
            v[i] = w * v[i] + c1 * np.random.rand() * (pbest[i] - x[i]) + c2 *
np.random.rand() * (gbest - x[i])
            x[i] += v[i]

            # Asegurar que la nueva posición esté dentro de las restricciones y no negativa
            if not (g1(x[i]) and g2(x[i])) or np.any(x[i] < 0):
                x[i] = pbest[i].copy()

        gbest_history.append(gbest_fit)

    return gbest, gbest_fit, gbest_history

# Valores de partículas a probar
particle_counts = [1, 2, 3, 4, 5, 10, 20, 50, 100]

# Graficar la evolución del valor óptimo para diferentes cantidades de partículas
plt.figure(figsize=(10, 6))

for n_particles in particle_counts:
    _, _, gbest_history = pso_optimization_with_constraints(n_particles, [g1, g2])
    plt.plot(gbest_history, label=f'{n_particles} partículas')

plt.xlabel('Iteraciones')
plt.ylabel('Valor óptimo')
plt.title('Evolución del valor óptimo para diferentes cantidades de partículas')
plt.legend()
plt.grid(True)
plt.show()
```