Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Práctica 3. Algoritmos heurísticos no constructivos poblacionales. Algoritmos Genéticos

Autor: Alberto Fernández Merchán
Asignatura: Modelos Bioinspirados y Heurísticas de Búsquedas

Índice

Índice de cuadros

1. Introducción

En esta práctica, estudiaremos el funcionamiento de algoritmos heurísticos no constructivos para resolver el problema de administración de energía de una central solar, que abordamos en la práctica 1. Estos algoritmos mejoran soluciones candidatas a través de un conjunto de operaciones, a partir de un conjunto de soluciones iniciales.

Analizaremos tres tipos de algoritmos: el algoritmo genético básico, CHC y el algoritmo genético multimodal. Además, compararemos sus resultados con el algoritmo de búsqueda local primero el mejor. Realizaremos la ejecución de los algoritmos con tres semillas diferentes (1) y aplicaremos los algoritmos tanto para datos reales como para datos aleatorios.

Luego, analizaremos los resultados obtenidos en función de la convergencia, el grado de adaptación (fitness), y simularemos el problema con el mejor individuo de cada ejecución. Asimismo, compararemos los resultados utilizando una tabla, en la que mediremos el número de veces que se llama a la función de evaluación y el dinero que se ha conseguido.

Por último, mostraremos algunos artículos en los que se resuelven problemas reales mediante el uso de algoritmos genéticos.

Las semillas que se han utilizado han sido las siguientes:

Semillas		
123456	654321	456789

Cuadro 1: Semillas que se han utilizado para estudiar el funcionamiento de los algoritmos.

2. Algoritmo Genético Básico

En primer lugar, analizaremos un algoritmo genético básico generacional con una élite de 5 individuos y una población de 17 inicializada de forma aleatoria.

2.1. Código

En esta sección veremos los códigos utilizados para realizar el cruce entre individuos, la mutación de cada uno y, por último, el código del algoritmo genético propiamente dicho.

2.1.1. Función de Cruce

Para el operador de cruce he utilizado el cruce en dos puntos. Este operador consiste en seleccionar dos puntos aleatorios de un cromosoma y generar los hijos concatenando cada parte del cromosoma de cada padre de la siguiente manera:

```
def cruce(individuo1, individuo2):

    pos1 = np.random.randint(0, len(individuo1) - 1)
    pos2 = np.random.randint(0, len(individuo1) - 1)
    min_pos = pos2 if pos2 < pos1 else pos1
    max_pos = pos2 if pos2 > pos1 else pos1
    hijo1 = np.append(individuo1[0:min_pos].copy(),
            np.append(individuo2[min_pos:max_pos].copy(),
                      individuo1[max_pos:len(individuo1)].copy()))

    hijo2 = np.append(individuo2[0:min_pos].copy(),
            np.append(individuo1[min_pos:max_pos].copy(),
                      individuo2[max_pos:len(individuo2)].copy()))
    return hijo1, hijo2
```

2.1.2. Función de Mutación

El operador de mutación consiste en cambiar un gen del cromosoma un 1 % de las veces que se genera un hijo. Este cambio consiste en sumar o restar, aleatoriamente, un valor de 10.
Este operador se utiliza para aumentar la diversidad entre los individuos de la población.

```
def mutacion(individuo):
    ind_mutado = individuo.copy()
    if np.random.random() <= Utils.PORCENTAJE_MUTACION:
        # Obtiene el valor de la posición asignada
        posicion = np.random.randint(0, 24)
        valor = 10
        # Sumar o Restar
        if np.random.uniform() < 0.5:
            ind_mutado[posicion] += valor
        else:
            ind_mutado[posicion] -= valor

        # Comprobamos que no supere los límites
        if ind_mutado[posicion] > 100:
            ind_mutado[posicion] = 100
        elif ind_mutado[posicion] < -100:
            ind_mutado[posicion] = -100
    return ind_mutado
```

2.1.3. Algoritmo Genético

```python
def algoritmo_genetico_generacional(semilla, isRandom):
    np.random.seed(semilla)
    t = 0
    # Inicializar P(t)
    poblacion = Utils.inicializar_poblacion(Utils.POBLACION_INICIAL)
    # Evaluar P(t)
    valores_poblacion, dinero_acumulado, bateria_acumulada = Utils.fitnessPoblacion(poblacion, isRandom)
    indice_maximo = np.argmax(valores_poblacion)
    # Obtenemos el mejor individuo de la población inicial.
    mejor_individuo = poblacion[indice_maximo]
    mejor_valor = valores_poblacion[indice_maximo]
    while t < Utils.NUM_ITERACIONES:
        t = t + 1
        # Seleccionamos la élite de la población:
        elite = elitismo(poblacion, Utils.ELITE, isRandom)
        numero_evaluaciones += len(poblacion) # Evaluamos a toda la poblacion para escoger la élite.

        # Seleccionar los índices de los padres (K=3)
        candidatos = np.array([torneo(valores_poblacion, 3) for _ in
                               range(Utils.POBLACION_INICIAL*2)])

        # Elegimos a los L=2 mejores de cada trío de padres
        padres = np.zeros(shape=(Utils.POBLACION_INICIAL-Utils.ELITE, 2), dtype=int)
        # Generamos el numero de la población menos el número de individuos
        # de la élite parejas.
        for i in range(0, Utils.POBLACION_INICIAL - Utils.ELITE):
            padres[i][0] = candidatos[i][-1]
            padres[i][1] = candidatos[i + 1][-1]

        # Recombinar P(t)
        hijos = np.zeros(shape=(Utils.POBLACION_INICIAL, 24), dtype=int)
        # Añado la élite a la siguiente generación:
        hijos[0:Utils.ELITE, :] = elite
        for i in range(Utils.ELITE, Utils.POBLACION_INICIAL, 2):
            h1, h2 = cruce(poblacion[padres[i-Utils.ELITE][0].copy()],
                           poblacion[padres[Utils.ELITE][1].copy()])
            hijos[i] = h1
            hijos[i + 1] = h2

        # Mutar P(t)
        hijos_mutados = np.apply_along_axis(mutacion, 1, hijos.copy())
        poblacion = hijos_mutados
        # Evaluar P(t)
        valores_poblacion, dinero_acumulado, bateria_acumulada = Utils.fitnessPoblacion(poblacion, isRandom)
        numero_evaluaciones += len(poblacion)
        indice_maximo = np.argmax(valores_poblacion)

        # Obtenemos el mejor individuo de la población.
        if mejor_valor < valores_poblacion[indice_maximo]:
            mejor_individuo = poblacion[indice_maximo].copy()
            mejor_valor = valores_poblacion[indice_maximo].copy()
            t = 0   # Cuando mejora, reiniciamos el contador de iteraciones.

    return mejor_valor
```

2.2. Parámetros

Los parámetros que se han utilizado son los siguientes:

Parámetro	**Valor**
Población Inicial	25
Porcentaje de Mutación	30 %
Número de Iteraciones sin mejora	100
Individuos de Élite	5
Ganadores de Torneo	3

Cuadro 2: Parámetros del algoritmo genético generacional básico.

2.3. Análisis del algoritmo

Los algoritmos genéticos generacionales trabajan mediante la generación iterativa de nuevas poblaciones de soluciones candidatas. Utilizan operadores genéticos como la selección, el cruce y la mutación para explorar el espacio de búsqueda y encontrar soluciones óptimas.

Una de las características clave de estos algoritmos es la capacidad de explotación. Esta se refiere a la capacidad del algoritmo para mejorar gradualmente la calidad de la población al seleccionar y reproducir las mejores soluciones en cada generación. Esta estrategia permite que el algoritmo converja hacia una solución óptima y se acerque a la región del espacio de búsqueda donde se encuentra la mejor solución.

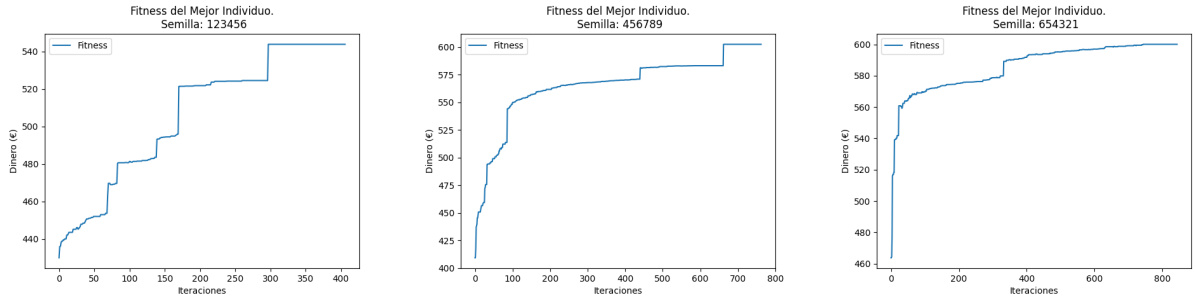
Sin embargo, a la hora de explorar nuevas soluciones, este algoritmo genético básico no consigue obtener una diversidad poblacional alta. Para intentar mejorar este aspecto, se introduce el operador de mutación, que realiza una perturbación en uno de los genes de un individuo para aumentar dicha diversidad. Según vemos en las tablas 4 y 7, la convergencia se realiza en una etapa temprana de la ejecución del algoritmo, lo que nos puede indicar que existe una convergencia prematura y, por tanto, poca diversidad genética.

2.4. Resultados

2.4.1. Problema Real

Fitness del mejor individuo

En las gráficas de la tabla 3 podemos observar el fitness del mejor individuo.

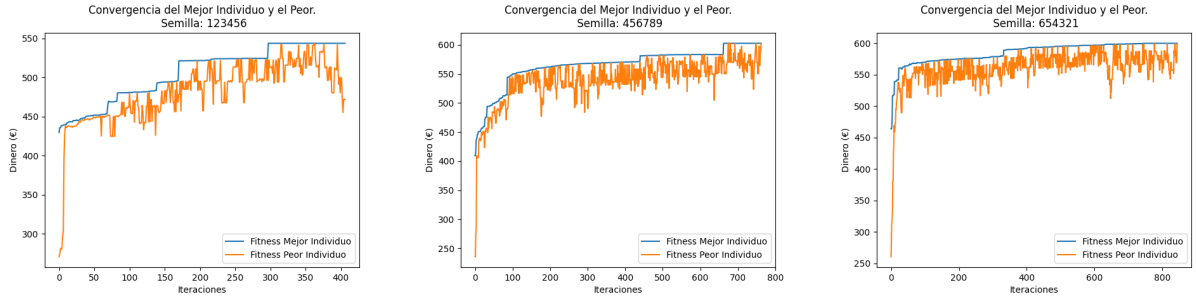


Cuadro 3: *Fitness* del mejor individuo en cada iteración del algoritmo sobre los datos del problema real con el algoritmo genético básico.

En las gráficas podemos ver que la adaptación de los individuos se realiza de forma escalonada y no continua. Además, tanto en el primer ejemplo como en el tercero, vemos que la función objetivo tiene una mejora considerable dejando como marca un codo en el perfil de la gráfica.

Convergencia del mejor y peor individuo

En las siguientes gráficas veremos el *fitness* del mejor y peor individuo en cada iteración que mejora la solución global del problema.

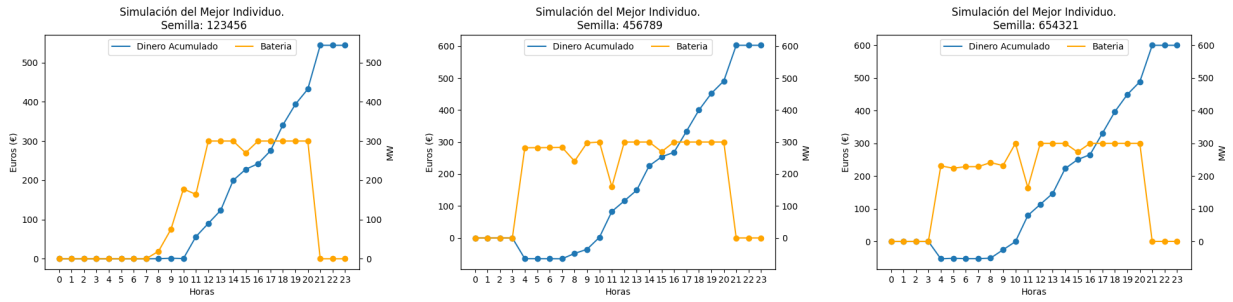


Cuadro 4: Convergencia del *fitness* del mejor y peor individuo en cada iteración del algoritmo sobre los datos del problema real con el algoritmo genético básico.

Podemos observar como, al comienzo de la ejecución del algoritmo, el peor individuo tiene un valor muy inferior al del mejor individuo. Sin embargo, conforme van sucediendo las iteraciones, comprobamos que el *fitness* de ambos individuos comienza a converger. EL *fitness* del peor individuo oscila por debajo del *fitness* del mejor cromosoma. Esto es normal debido a que el peor individuo no tiene por qué mejorar en cada iteración. Sin embargo, vemos que se acerca al del mejor individuo.

Simulación del problema

Por último visualizaremos las gráficas de simulación del mejor individuo.



Cuadro 5: Simulación del mejor individuo sobre los datos del problema real con el algoritmo genético básico.

En las gráficas del cuadro 5 podemos ver que la solución obtenida decide comprar energía al comienzo del día para llenar la batería y venderla más adelante junto al sobrante.

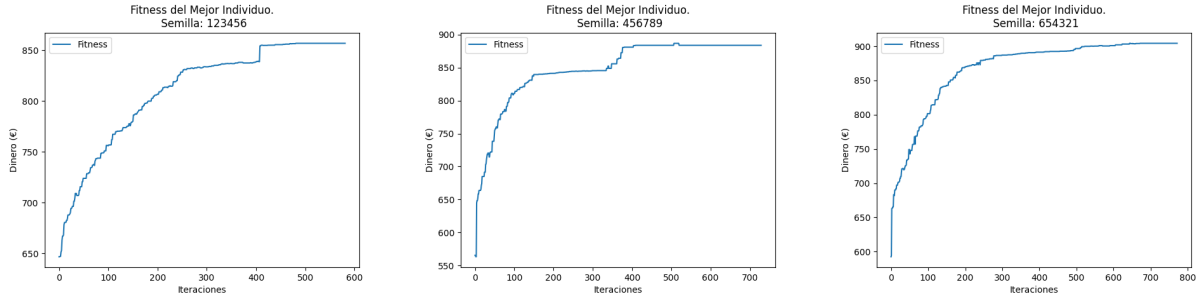
Si visualizamos las gráficas de la práctica 1, se parecen mucho a las que obteníamos con la búsqueda tabú, incluso, en el caso del algoritmo genético, llega a superar el valor máximo de ganancia obtenida.

En este caso, se llega a obtener una cantidad superior a los 500e en todos los casos, cuando en la búsqueda tabú obteníamos un beneficio medio inferior a esa cifra.

2.4.2. Problema Aleatorio

Fitness del mejor individuo

A continuación se muestran las gráficas del *fitness* del mejor individuo durante cada iteración del algoritmo sobre los datos del problema aleatorio.

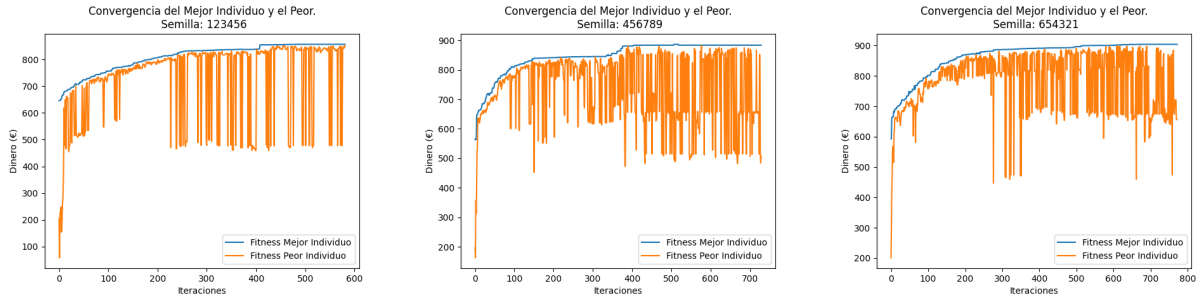


Cuadro 6: *Fitness* del mejor individuo en cada iteración del algoritmo sobre los datos del problema aleatorio con el algoritmo genético básico.

Al igual que en el problema de los datos reales, el *fitness* del mejor individuo se incrementa de forma escalonada hasta llegar a una planicie donde no consigue mejorar más. En este caso, no se marca tanto el codo de la mejora del algoritmo genético en el perfil de la gráfica. Sin embargo, los resltados obtenidos son mucho mejores con estos datos que con los del problema anterior.

Convergencia del mejor y peor individuo

En este párrafo se muestran las gráficas de convergencia del mejor y peor individuo en cada iteración que cambia el mejor individuo.

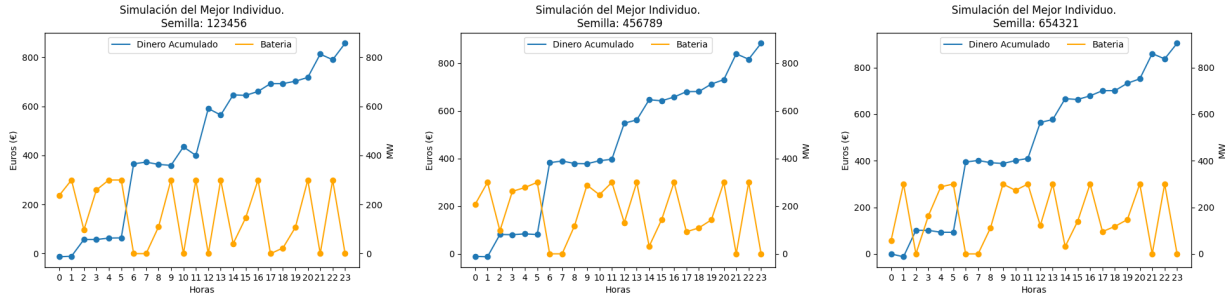


Cuadro 7: Convergencia del *fitness* del mejor y peor individuo en cada iteración del algoritmo sobre los datos del problema aleatorio con el algoritmo genético básico.

En cuanto a la convergencia de la población en este problema, sucede parecido al anterior problema. Sin embargo, la oscilación del *fitness* del peor individuo es mucho más pronunciada que con los datos reales. Esto puede ser debido a que el porcentaje de mutación es muy alto. Sin embargo, como se ha dicho anteriormente, se ha elegido este porcentaje ya que ofrece mejores resultados en el problema de datos reales.

Simulación del problema

Por último, mostraremos la simulación de la batería y la ganancia obtenida para el problema con los datos aleatorios.



Cuadro 8: Simulación del mejor individuo sobre los datos del problema aleatorio con el algoritmo genético básico.

Como ocurría con el anterior problema, las gráficas de simulación se parecen mucho a las que obteníamos con el algoritmo de búsqueda tabú. Estas gráficas son erráticas debido a la naturaleza aleatoria de los datos de venta y compra de energía.

2.5. Experimentación

En esta sección he tratado de experimentar con algunos parámetros para conseguir mejorar los resultados obtenidos por el algoritmo genético básico. Los parámetros que he seleccionado para experimentar son los siguientes:

2.5.1. Número de Iteraciones

Respecto al número de iteraciones, he escogido los siguientes valores para ejecutar el algoritmo y tratar de ver cuál es la mejor cantidad para colocar en la condición de parada del algoritmo:

Iteraciones sin Mejora			
100	200	300	**500**

Cuadro 9: Número de iteraciones que se han utilizado para estudiar el funcionamiento de los algoritmos.

2.5.2. Porcentaje de Mutación

En cuanto al porcentaje de mutación se han elegido los siguientes valores:

Porcentaje de Mutación				
0.01	0.05	0.1	0.2	**0.3**

Cuadro 10: Porcentaje de mutación que se han utilizado para estudiar el funcionamiento de los algoritmos.

2.5.3. Población Inicial

Por último, he tomado los siguientes valores para experimentar con el número de individuos que tendría la población inicial:

Población Inicial							
15	17	19	21	23	**25**	27	29

Cuadro 11: Número de individuos iniciales que se han utilizado para estudiar el funcionamiento de los algoritmos.

Finalmente he elegido un número de iteraciones de 2000, un porcentaje de mutación de 0,3 y una población inicial de 25 individuos. Estos valores son los que mejor media en el resultado me proporcionan. Cabe destacar que el porcentaje de mutación es alto, sin embargo, tras realizar la experimentación con los valores anteriormente señalados en la Tabla 10, es el que mejor resultado da con el resto de parámetros.

3. CHC

El algoritmo CHC fue diseñado para cromosomas binarios, sin embargo, en este caso utilizaremos un operador de cruce conocido como *BLX-α* que nos servirá para generar hijos con codificación real.

3.1. Código

3.1.1. Operador de Cruce

El operador de cruce en este algoritmo consiste en obtener un hijo lo suficientemente diferente de los padres y lo suficientemente parecido para que mantenga la diversidad y pueda alcanzar una buena solución.

```python
def blx_alpha(parent_1, parent_2, alpha):
    """
    Operador de cruce BLX-alpha para dos padres con valores reales.
    :param parent_1: Cromosoma padre 1.
    :param parent_2: Cromosoma padre 2.
    :param alpha: Valor de alpha para el operador BLX-alpha.
    :return: Dos hijos generados a partir de los padres mediante el operador BLX-alpha.
    """
    # Creamos dos hijos vacíos con el mismo tamaño que los padres.
    child_1 = np.zeros_like(parent_1)
    child_2 = np.zeros_like(parent_2)
    # Recorremos los genes de los padres y generamos los genes de los hijos.
    for i in range(len(parent_1)):
        # Calculamos los límites del intervalo de cruce para el gen i.
        cmin = min(parent_1[i], parent_2[i])
        cmax = max(parent_1[i], parent_2[i])
        I = cmax - cmin
        # Generamos los genes de los hijos utilizando el operador BLX-alpha.
        delta = alpha * I
        rand = np.random.uniform(-delta, 1 + delta)
        child_1[i] = 0.5 * ((1 + rand) * parent_1[i] + (1 - rand) * parent_2[i])
        child_2[i] = 0.5 * ((1 - rand) * parent_1[i] + (1 + rand) * parent_2[i])

        # Ajustamos los valores de los hijos si superan los límites permitidos.
        child_1[i] = max(min(child_1[i], 100), -100)
        child_2[i] = max(min(child_2[i], 100), -100)

    return child_1, child_2
```

3.1.2. Operador de Selección

El operador de selección del algoritmo CHC consiste en reemplazar a los peores individuos de los padres por los mejores individuos de los hijos, aumentando así la calidad de los individuos de la población resultante.

```python
def select_s(padres, hijos, isRandom):
    """
    Reemplaza los peores miembros de padres por los mejores miembros de hijos mientras haya algún miembro en
    hijos mejor que en padres.

    Args:
        padres (numpy.ndarray): La matriz de padres.
        hijos (numpy.ndarray): La matriz de hijos.

    Returns:
        numpy.ndarray: La matriz de padres actualizada.
    """
    valores_padres = Utils.fitnessPoblacion(padres, isRandom)[0]
    valores_hijos = Utils.fitnessPoblacion(hijos, isRandom)[0]

    while np.any(valores_hijos > valores_padres):
        indices_padres = np.argsort(valores_padres)
        indices_hijos = np.argsort(valores_hijos)[::-1]
        for i in range(len(indices_hijos)):
            if valores_hijos[indices_hijos[i]] > valores_padres[indices_padres[i]]:
                padres[indices_padres[i]] = hijos[indices_hijos[i]]
                valores_padres[indices_padres[i]] = valores_hijos[indices_hijos[i]]

    return padres
```

3.1.3. Algoritmo CHC

```python
def CHC(semilla, israndom):
    np.random.seed(semilla)
    t = 0
    d = 24 / 4
    # Inicializar Poblacion
    poblacion = Utils.inicializar_poblacion(Utils.POBLACION_INICIAL - 1)
    # Evaluar Poblacion
    valores_poblacion = Utils.fitnessPoblacion(poblacion, israndom)[0]
    indice_maximo = np.argmax(valores_poblacion)

    # Mejor Valor Inicial
    mejor_individuo = poblacion[indice_maximo]
    mejor_valor = valores_poblacion[indice_maximo]

    # Mientras no se cumpla la condicion
    while t < Utils.NUM_ITERACIONES_CHC:
        t += 1
        # Seleccionar_r
        parejas = poblacion.copy()
        np.random.shuffle(parejas)

        # Cruzar
        hijos = np.empty_like(poblacion)
        for i in range(0, len(parejas), 2):
            hijo1, hijo2 = blx_alpha(parejas[i], parejas[i + 1], Utils.ALPHA)
            hijos[i] = hijo1
            hijos[i + 1] = hijo2

        # Seleccionar
        poblacion = select_s(parejas, hijos, israndom)
        valores_poblacion, _, _ = Utils.fitnessPoblacion(poblacion, israndom)
        indice_maximo = np.argmax(valores_poblacion)
        indice_minimo = np.argmin(valores_poblacion)

        # Actualizamos el mejor valor de toda la historia
        if valores_poblacion[indice_maximo] > mejor_valor:
            mejor_valor = valores_poblacion[indice_maximo].copy()

        # Obtenemos el mejor individuo de la generación
        mejorIndividuoGenAnterior = poblacion[indice_maximo].copy()

        # Si P(t) == P(t-1)
        if np.array_equal(poblacion, hijos):
            d -= 1
        # Si d < 0 -> diverge P(t) y d = 24/4
        if d < 0:
            # En el arranque los valores de un cromosoma corresponden al mejor
            # individuo de la generación anterior y el resto serán aleatorios
            d = 24/4
            poblacion = Utils.inicializar_poblacion(Utils.POBLACION_INICIAL - 1)
            poblacion[0] = mejorIndividuoGenAnterior.copy()

    return mejor_valor
```

3.2. Parámetros

Los parámetros que utilizaremos para este algoritmo son los siguientes:

Parámetro	**Valor**
Población Inicial	16
Alpha (α)	0.1
Número de Iteraciones sin mejora	1000

Cuadro 12: Parámetros del algoritmo genético CHC.

3.3. Análisis del algoritmo

Este algoritmo utiliza la técnica de cruce *BLX-*α para introducir diversidad en la población y evitar la convergencia prematura. El algoritmo CHC mantiene un equilibrio entre diversidad y convergencia utilizando varias reinicializaciones de la población.

En cuanto a la capacidad de exploración, el algoritmo CHC consigue potenciar esta característica debido a su mecanismo de reinicialización. Este permite volver a reiniciar la población de individuos cuando convergen todos ellos un número determinado de veces.

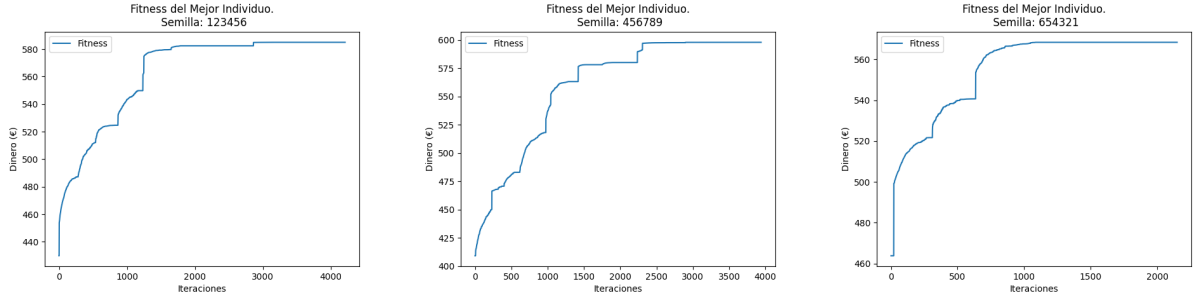
Por otro lado, la capacidad de explotación del algoritmo viene dada por el operador de cruce que permite explorar el entorno de las soluciones ya encontradas generando hijos similares a los padres. Además, para aumentar esta explotación, al reiniciar también se copia el mejor individuo en la población siguiente para conservar una especie de heurística sobre la mejor solución encontrada en la iteración anterior.

3.4. Resultados

3.4.1. Problema Real

Fitness del mejor individuo

A continuación se observa el *fitness* en las diferentes semillas probadas con el algoritmo CHC sobre los datos del problema real.

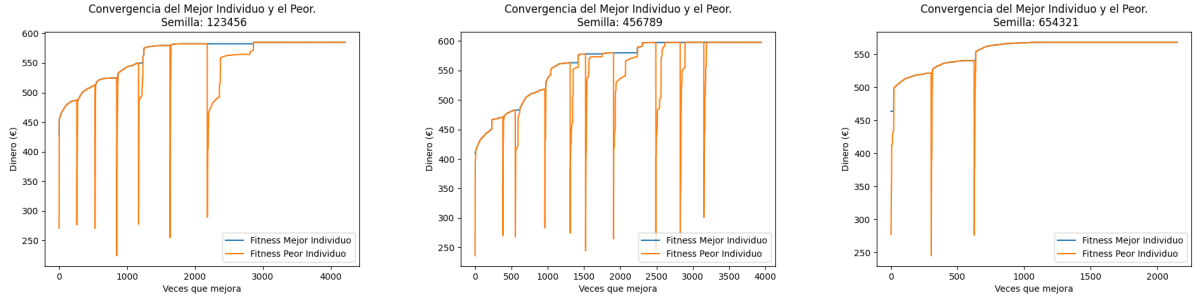


Cuadro 13: *Fitness* del mejor individuo en cada iteración del algoritmo sobre los datos del problema real con el algoritmo genético CHC.

Podemos ver que, como en el anterior algoritmo, la mejora del *fitness* es escalonada. Sin embargo, en este caso podemos ver como existen periodos de una mayor estabilidad tras cada mejora. Esto puede ser debido al contador de reinicios del algoritmo CHC, que reinicia la población solo cuando han pasado un número concreto de poblaciones iguales.

Convergencia del mejor y peor individuo

En este párrafo veremos el *fitness* del mejor y peor individuo de la población en cada iteración en la que el mejor individuo se actualiza con los datos del problema real.

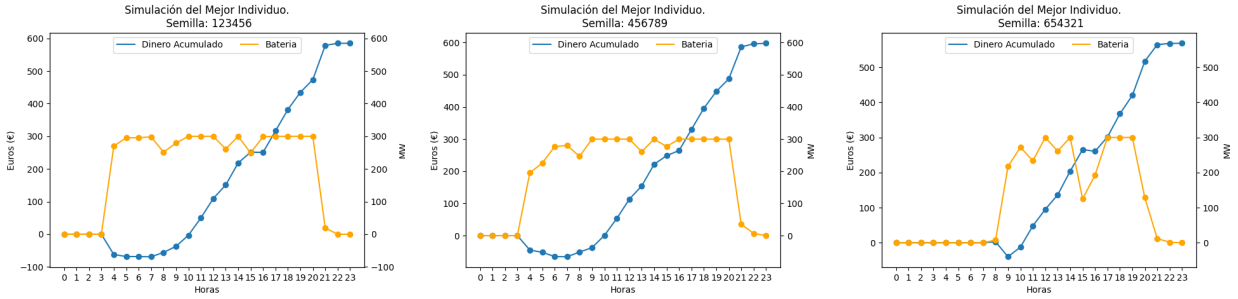


Cuadro 14: Convergencia del *fitness* del mejor y peor individuo en cada iteración del algoritmo sobre los datos del problema real con el algoritmo CHC.

En estas gráficas podemos ver que, en el algoritmo CHC, el peor individuo se adapta demasiado bien al *fitness* del mejor individuo de la población. Sin embargo, al reiniciar, pierde el valor que tenía anteriormente y cae a valores muy inferiores. Sin embargo, con unas pocas iteraciones más, el peor individuo llega a converger rápidamente con el mejor.
Además, en las gráficas podemos ver, perfectamente, cuando el algoritmo decide reinicializar la población, ya que se pueden notar los picos en el *fitness* del peor individuo alcanzando valores muy bajos, ya que, al reiniciar la población de manera aleatoria, siempre habrá un individuo peor que el mejor obtenido hasta entonces.

Simulación del problema

Por último, veremos como se muestra la capacidad de la batería y el dinero obtenido después de cada hora del día en la simulación de la solución obtenida con los datos del problema real.



Cuadro 15: Simulación del mejor individuo sobre los datos del problema real con el algoritmo genético CHC.

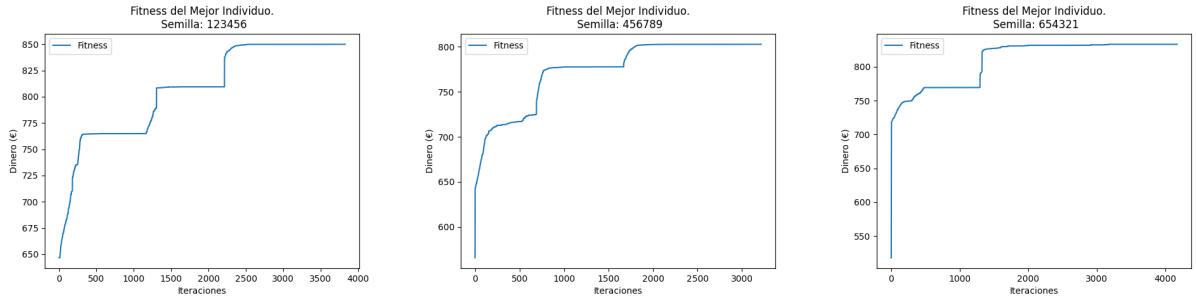
Observamos que, como en el algoritmo anterior, las gráficas son similares a las de los algoritmos de búsqueda tabú y enfriamiento simulado que vimos en la práctica 1.

Podemos notar una diferencia con la gráfica del algoritmo anterior ya que la segunda semilla no comienza comprando energía y llenando la batería. Por lo demás, el algoritmo CHC llega a conseguir un resultado un poco mejor que el genético básico con un número de llamadas a la función de evaluación mucho menor.

3.4.2. Problema Aleatorio

Fitness del mejor individuo

En las siguientes gráficas veremos el *fitness* del mejor individuo para el problema de datos aleatorios. Podemos observar que, en el problema aleatorio, los periodos de estabilidad del algoritmo son mucho más



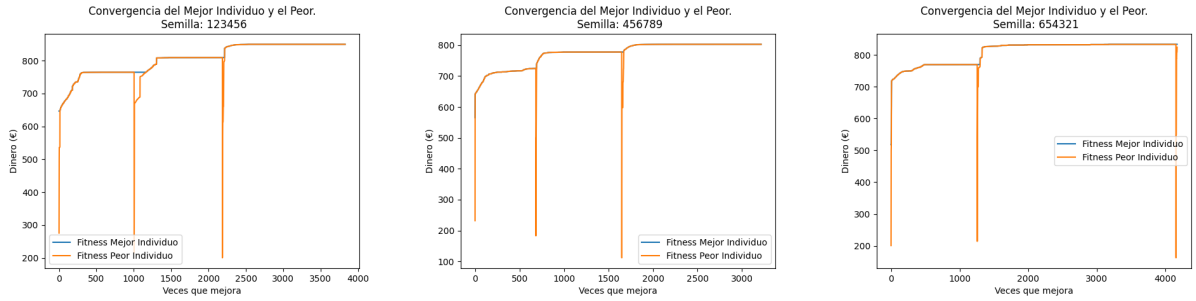
Cuadro 16: *Fitness* del mejor individuo en cada iteración del algoritmo sobre los datos del problema aleatorio con el algoritmo genético CHC.

prolongados que en el de datos reales. Comparándolo con el algoritmo genético, podemos ver que la mejora del mejor individuo se realiza, sobre todo, en unos pocos escalones. Esto puede ser debido a la reinicialización de la población del algoritmo.

Cabe destacar el borde redondeado de los codos de cada gráfica, esta característica puede ser debida a la pequeña mejora que proporciona el cruce *blx*-α. Puede indicarnos que el cruce que hemos elegido está mal escogido o tiene un α demasiado pequeño.

Convergencia del mejor y peor individuo

A continuación se muestran los *fitness* del mejor y peor individuo para el problema de datos aleatorios.

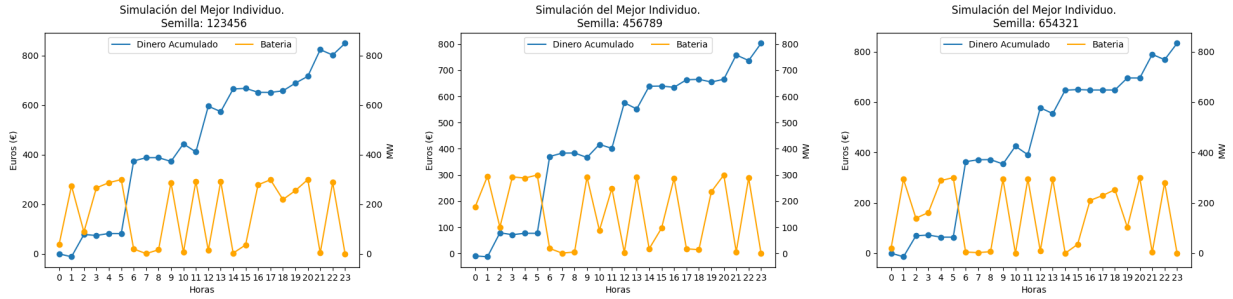


Cuadro 17: Convergencia del *fitness* del mejor y peor individuo en cada iteración del algoritmo sobre los datos del problema aleatorio con el algoritmo genético CHC.

En estas gráficas podemos ver que, como en el problema de datos reales, el peor individuo converge con el mejor individuo muy rápidamente.
También podemos observar que, en este caso, se ha reinicializado menos veces que para el problema real. Esto es porque la población anterior de cada iteración no ha conseguido igualar a la actual más que 2 veces. Estos reinicios se pueden notar en los picos del *fitness* del peor individuo, que llegan a alcanzar valores de 200e.

Simulación del problema

Por último, veremos el estado de la batería y la ganancia obtenida cada hora por la solución obtenida para el problema de datos aleatorios.



Cuadro 18: Simulación del mejor individuo sobre los datos del problema aleatorio con el algoritmo genético CHC.

En estas gráficas podemos ver que, como en el algoritmo anterior, la capacidad de la batería no sigue un patrón determinado debido a la naturaleza aleatoria de los datos del problema. Las gráficas se parecen, también, a las obtenidas por el algoritmo de búsqueda tabú de la práctica 1.

3.5. Experimentación

En esta sección anotaremos los valores de los diferentes parámetros escogidos para el algoritmo CHC. Experimentaremos sobre los siguientes parámetros:

3.5.1. Parámetro α del operador de cruce

El operador de cruce que hemos escogido es el BLX-α, este operador tiene un parámetro (α) que permite aumentar o disminuir la diversidad de la población escogida. He probado los siguientes valores para él:

Parámetro Alpha					
0.001	0.01	0.05	0.075	**0.1**	0.2

Cuadro 19: Parámetros α escogidos para el operador de cruce para estudiar el funcionamiento del algoritmo CHC.

3.5.2. Número de Iteraciones sin mejora

En este apartado experimentaremos con el número de iteraciones sin mejora del algoritmo CHC. Los valores escogidos han sido los siguientes:

Número de Iteraciones sin mejorar						
100	200	300	500	**1000**	1500	2000

Cuadro 20: Número de iteraciones que se han utilizado para estudiar el funcionamiento del algoritmo CHC.

Finalmente, he elegido $\alpha = 0{,}05$ y el número de iteraciones sin mejora será de 300 ya que esta ha sido la combinación que mejores resultados ha obtenido (30).

4. Algoritmo Genético Multimodal

4.1. Código

4.1.1. Función de Clearing

```
def clearing(poblacion, israndom):
    # Realizamos la seleccion sobre los individuos dominantes, por tanto, hay que limpiar a los individuos
    # de cada nicho. Obtenemos los valores del fitness de la poblacion
    fitness_poblacion = Utils.fitnessPoblacion(poblacion,israndom)[0]
    # Clasificación de la población de forma descendente
    indices_ordenados = np.argsort(-fitness_poblacion)
    # Se escoge al mejor individuo y se compara con el resto de forma descendente.
    # Aquellos individuos que estén dentro de su radio, quedan eliminados.
    for i in range(len(fitness_poblacion)):
        if fitness_poblacion[indices_ordenados[i]] > 0:
            numGanadores = 1
            for j in range(i+1,len(fitness_poblacion)):
                if fitness_poblacion[indices_ordenados[j]] > 0 and \
                    Utils.distanciaEuclidea(poblacion[indices_ordenados[i]],
                                            poblacion[indices_ordenados[j]]) < Utils.RADIO_CLEARING:
                    if numGanadores < Utils.KAPPA:
                        numGanadores += 1
                    else:
                        fitness_poblacion[indices_ordenados[j]] = 0
    # Devuelve los individos que no han sido eliminados.
    return poblacion[fitness_poblacion[indices_ordenados] != 0]
```

4.2. Parámetros

Como este algoritmo es una modificación del algoritmo genético básico, algunos parámetros son los mismos:

Parámetro	Valor
Población Inicial	27
Porcentaje de Mutación	10 %
Número de Iteraciones sin mejora	500
Individuos de Élite	5
Porcentaje de la población para el cruce	80 %
Generaciones antes de hacer clearing	10
Radio de Clearing	5
Kappa	2

Cuadro 21: Parámetros del algoritmo genético multimodal.

4.3. Análisis del algoritmo

Este algoritmo es una modificación del algoritmo genético básico. Consiste en cruzar tan solo los individuos dominantes de la población, es decir, los que sobrevivan a la función de *clearing*. Este cruce se producirá con una probabilidad del 80 %, en caso de que no se produzca, se copian los individuos directamente a la siguiente población. Este mecanismo, al matar a los individuos que son parecidos entre sí, favorece la diversidad de la población. Sin embargo, más adelante veremos que el algoritmo padece de convergencia prematura.

En teoría, al mantener a un subconjunto de individuos en cada solución lo suficientemente distinta entre sí, favorecemos a la diversidad genética y, consecuentemente, aumentaríamos el grado de exploración del algoritmo al cruzar los individuos. Por otro lado, el grado de explotación de las soluciones se mantiene como en el algoritmo genético básico, ya que el operador de cruce es el mismo (cruce en dos puntos).

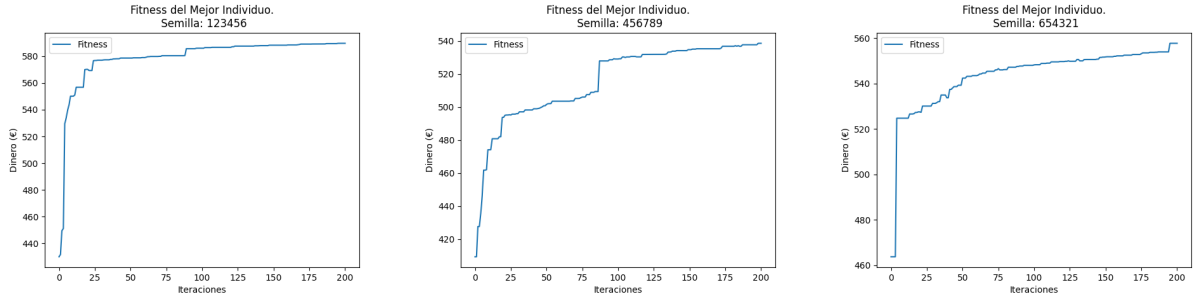
Para evitar la convergencia prematura, se han añadido 2 individuos aleatorios a la población después de hacer el cruce. Esto permite mantener la diversidad genética y obtener mejores resultados.

4.4. Resultados

4.4.1. Problema Real

Fitness del mejor individuo

A continuación observamos las gráficas del *fitness* del mejor individuo sobre las diferentes ejecuciones del problema de datos reales.



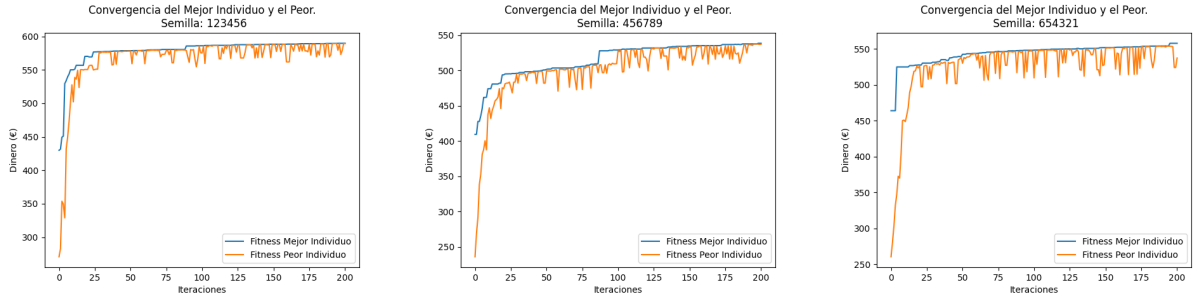
Cuadro 22: *Fitness* del mejor individuo en cada iteración del algoritmo sobre los datos del problema real con el algoritmo genético Multimodal.

En estas gráficas podemos observar que, como en el algoritmo genético básico, el grado de adaptación del mejor individuo mejora considerablemente. Podemos observar que la mejora del individuo se realiza de forma escalonada, al igual que en el algoritmo básico.

El resultado es el esperado, puesto que la base del algoritmo es la misma que el algoritmo genético básico, es normal que la forma de la gráfica sea parecida. Sin embargo, al añadir dos individuos aleatorios cada vez que se generan los hijos, el resultado mejora considerablemente llegando a superar los 600e de beneficio.

Convergencia del mejor y peor individuo

Ahora mostraré las gráficas relativas al *fitness* del mejor y peor individuo de las diferentes ejecuciones del algoritmo multimodal sobre los datos del problema real.

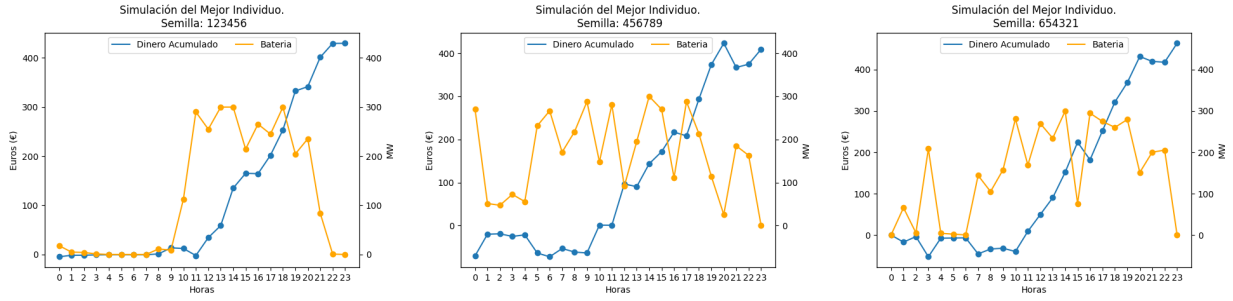


Cuadro 23: Convergencia del *fitness* del mejor y peor individuo en cada iteración del algoritmo sobre los datos del problema real con el algoritmo Multimodal.

En esta variación del algoritmo básico, podemos observar que el *fitness* del peor individuo oscila por debajo del mejor individuo de la población. Esto puede ser debido al método de *clearing* que elimina a los individuos que están muy cercanos entre sí.
Cabe destacar, que en la última semilla, el peor individuo no llega a converger con el mejor. Sin embargo, en las otras dos sí que termina haciéndolo al final. Esto puede ser debido a falta de tiempo para que el peor individuo converja.

Simulación del problema

Por último, se muestran las gráficas que realizan la simulación de los beneficios y estado de la batería por cada hora que proporciona la solución dada por el mejor individuo de cada ejecución del algoritmo sobre los datos del problema real.



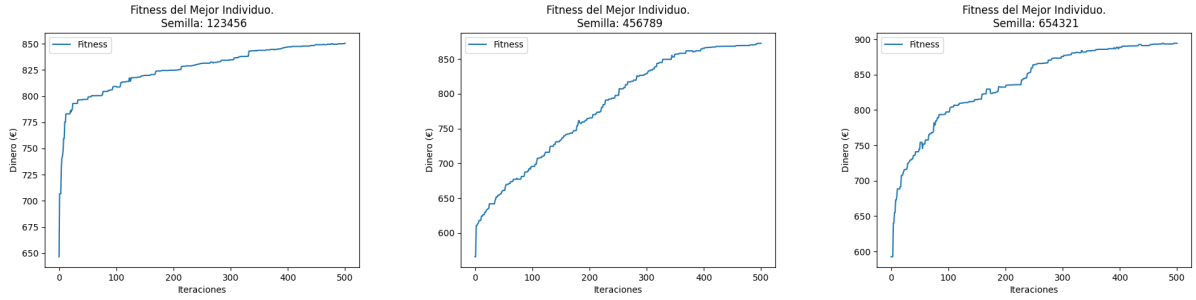
Cuadro 24: Simulación del mejor individuo sobre los datos del problema real con el algoritmo genético Multimodal.

En la simulación de la mejor solución generada por el mejor individuo de cada ejecución podemos ver que se asemeja mucho a las obtenidas por el algoritmo genético básico. Sin embargo, podemos observar que el algoritmo decide llenar la batería mucho antes que en la otra versión del algoritmo. Esto le permite obtener un resultado superior a los otros dos algoritmos que hemos visto en esta práctica.

4.4.2. Problema Aleatorio

Fitness del mejor individuo

En este párrafo se muestra el *fitness* del mejor individuo de las tres ejecuciones realizadas sobre los datos del problema aleatorio.

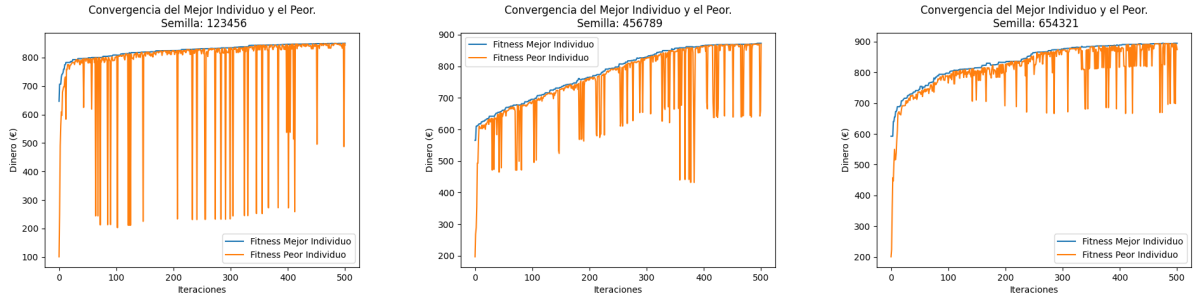


Cuadro 25: *Fitness* del mejor individuo en cada iteración del algoritmo sobre los datos del problema aleatorio con el algoritmo genético Multimodal.

Las gráficas son muy parecidas a las del problema de datos reales. El mejor individuo mejora escalonadamente hasta llegar a converger a un valor cercano a $900e$.

Convergencia del mejor y peor individuo

A continuación se muestran las gráficas del *fitness* del peor y mejor individuo de cada ejecución del algoritmo sobre los datos del problema aleatorio.

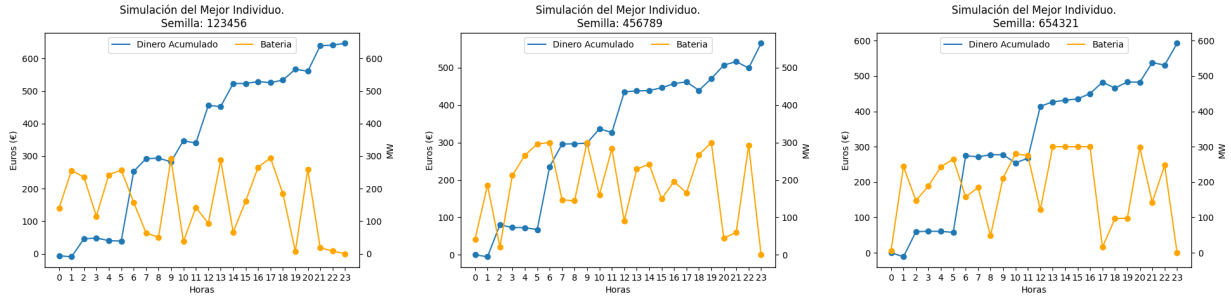


Cuadro 26: Convergencia del *fitness* del mejor y peor individuo en cada iteración del algoritmo sobre los datos del problema aleatorio con el algoritmo genético Multimodal.

En este problema, el *fitness* del peor individuo tiene una mayor oscilación. Sin embargo, la población acaba convergiendo con el mejor individuo de la misma.

Simulación del problema

Finalmente, mostraremos la simulación del estado de la batería y de las ganancias obtenidas en cada ejecución con los datos del problema aleatorio.



Cuadro 27: Simulación del mejor individuo sobre los datos del problema aleatorio con el algoritmo genético Multimodal.

Las gráficas de simulación son similares a las de los otros algoritmos que hemos visto en esta práctica al igual que a las de los algoritmos de búsqueda tabú y enfriamiento simulado. Podemos observar que la ganancia de la mejor solución llega a superar los 800 euros, siendo esta la solución más alta obtenida en la práctica.

4.5. Experimentación

Para este algoritmo dejaremos los parámetros que hemos obtenido con la experimentación del algoritmo genético básico y nos centraremos en experimentar con los valores siguientes:

4.5.1. Número de generaciones para hacer clearing

Los valores que hemos escogido para iterar para encontrar el mejor número de generaciones tras el que hacer *clearing* han sido:

Cantidad de generaciones antes de hacer Clearing						
10	20	50	100	150	200	300

Cuadro 28: Cantidad de generaciones que se han utilizado para estudiar el funcionamiento del algoritmo multimodal.

4.5.2. Kappa

Por último, el número de individuos que dominan cada nicho se ha escogido entre los siguientes valores:

Número de Kappas								
2	3	4	5	6	7	8	9	10

Cuadro 29: Número de kappas que se han utilizado para estudiar el funcionamiento del algoritmo CHC.

Finalmente, se han escogido hacer clearing tras 10 generaciones y hacerlo con un radio de 5 y un *kappa* de 2. He utilizado la distancia euclídea para calcular el radio de clearing. El valor del radio es 5 porque, si se aumenta, no se generan demasiados nichos, lo que puede perjudicar a la diversidad de la población.

5. Análisis de Resultados

5.1. Problema Real

Problema Real						
Algoritmo	**Ev. Medias**	**Ev. Mejor**	**Desv. Evaluaciones**	**Mejor Fitness**	**Media Fitness**	**Desv. Fitness**
Búsqueda Local	53.8	52	1.79	361.1	309.1	31.54
G. Básico	50300	30475	17435.57	602.62	582.2	33.16
G. Multimodal	16988.66	16965	20.79	589.78	562.04	25.87
CHC	54949.33	34400	17925.88	597.95	583.75	14.83

Cuadro 30: Comparación de métricas para los datos del problema real.

En la tabla 30 podemos observar los resultados obtenidos en los algoritmos genéticos implementados en esta práctica aplicados al problema de datos reales. Además, también se compara con los resultados obtenidos por la búsqueda local de primero el mejor que vimos en la práctica 1.

En cuanto a los resultados obtenidos, podemos ver que el **CHC** obtiene la mayor ganancia media. Por otro lado, el **algoritmo genético básico** consigue el mejor resultado máximo de las tres semillas. Además, la desviación típica en cuanto a llamadas a la función de evaluación es muy parecida en ambos algoritmos. Esto denota una robustez muy inferior respecto al algoritmo genético multimodal, ya que su desviación típica es de 20,79 llamadas a la función de evaluación.

En cuanto a la robustez de estos algoritmos, podemos observar que, para los resultados, generan soluciones muy parecidas entre sí. El algoritmo genético multimodal tiene la peor desviación típica en relación con los resulados obtenidos. Este algoritmo tiene una desviación típica de 25,87e que es superior a la de los algoritmos genético básico y CHC (33,16 y 14,83e respectivamente).

El algoritmo multimodal, acaba con 13, 17 y 18 nichos en cada una de las semillas respectivamente. Esto es lo esperado, ya que el radio de clearing es muy pequeño para que pueda realizar correctamente los nichos. Además, el número de nichos obtenidos es superior a 2 e inferior al tamaño de la población escogido (27).

5.2. Problema Aleatorio

Problema Aleatorio						
Algoritmo	**Ev. Medias**	**Ev. Mejor**	**Desv. Evaluaciones**	**Mejor Fitness**	**Media Fitness**	**Desv. Fitness**
Búsqueda Local	53.6	53	0.55	523.69	430.15	71.55
G. Básico	52050.0	43600	7485.48	904.36	881.66	23.86
G. Multimodal	42369	42350	27.07	894.4	872.5	22.01
CHC	59829.33	51456	7756.16	849.89	828.66	23.87

Cuadro 31: Comparación de métricas para los datos del problema aleatorio.

Por otro lado, en la tabla 31 podemos observar los resultados obtenidos en los algoritmos genéticos implementados en esta práctica aplicados al problema de datos aleatorios. También se compara con los resultados obtenidos por la búsqueda local de la primera práctica.

En cuanto a los resultados obtenidos, podemos ver que, a diferencia del problema anterior, el **algoritmo genético básico** consigue mejores resultados que sus otros competidores.
En cuanto a la robustez de los resultados de los algoritmos genéticos, podemos observar que son muy parecidas entre sí, a diferencia del algoritmo de búsqueda local que lo supera en más del triple de desviación típica.

Por otro lado, si observamos el número de llamadas a la función de evaluación, notamos que el CHC realiza, de media, un número mayor de evaluaciones. En este aspecto, el algoritmo multimodal realiza un número inferior de evaluaciones respecto a los otros dos algoritmos genéticos.

La mayor robustez en cuanto al número de llamadas a la función de evaluación la consigue el algoritmo multimodal, ya que tiene una desviación típica de 27,07 llamadas a la función.

En el algoritmo multimodal obtenemos 13, 14 y 18 nichos para cada semilla respectivamente. Esto es, como hemos mencionado en el problema anterior, lo esperado, ya que genera un número de nichos que se encuentra en el rango de 2 y el número de individuos totales de la población.

6. Artículos Relacionados

6.1. Genético Básico

Este artículo ([Shopova and Vaklieva-Bancheva, 2006]) presenta en detalle un algoritmo genético llamado BASIC, diseñado para aprovechar los esquemas genéticos conocidos y ser capaz de resolver numerosos problemas de optimización. BASIC GA sigue los pasos comunes de los algoritmos genéticos e incluye esquemas de representación real para variables enteras y reales. También se aplican tres esquemas de selección sesgada para la reproducción, cuatro para la recombinación y tres para la mutación. Además, se aborda un nuevo esquema de selección para la sustitución.

BASIC GA puede ajustarse fácilmente a problemas concretos mediante la adaptación de sus parámetros globales y locales, y ofrece la posibilidad de ampliar los operadores genéticos con nuevos esquemas. Se han resuelto una serie de problemas de optimización para probar su capacidad, utilizando funciones de penalización estáticas y dinámicas para manejar todo tipo de restricciones. Las soluciones obtenidas son comparables con otros algoritmos genéticos y técnicas de solución.

Por último, menciona algunos ejemplos de aplicaciones de algoritmos genéticos en la optimización de problemas de ingeniería, como la optimización de la fermentación de cerveza o la programación de trabajos en talleres.

6.2. CHC

En este artículo [Rathee and Ratnoo, 2020] se presenta un algoritmo genético multiobjetivo para la selección de características, llamado MOCHC-FS. El objetivo es seleccionar características no redundantes e informativas para el preprocesamiento de conjuntos de datos antes de aplicar algoritmos de minería de datos.

El algoritmo utiliza una combinación de clasificación no dominada y algoritmos genéticos CHC para obtener un conjunto de soluciones no dominadas que cumplen con ambos objetivos de precisión y tasa de reducción.
Este algoritmo se ha validado en veinte conjuntos de datos disponibles en el repositorio de datos de UCI y se demuestra que puede encontrar soluciones óptimas que cumplen ambos objetivos.

Por último, se extrae un subconjunto de características de las soluciones no dominadas y se registran las tasas de precisión y reducción para varios conjuntos de datos experimentales utilizando el algoritmo de clasificación KNN en las características seleccionadas.

6.3. Multimodal

El artículo [Dilettoso and Salerno, 2006] propone un nuevo algoritmo genético de nicho llamado ***Self-adaptive NGA*** (SANGA), que utiliza una variable adicional en el problema de optimización para estimar el radio del nicho en lugar de asignarlo a priori, como se hace en otros algoritmos genéticos del mismo estilo.

SANGA se combina con el método de búsqueda de patrones determinista para formar un método híbrido de optimización que funciona bien en la optimización de funciones multimodales y en el diseño de dispositivos electromagnéticos. La capacidad de estos algoritmos para localizar múltiples óptimos puede ser útil para los problemas de optimización del mundo real que a menudo exhiben múltiples óptimos. Este algoritmo consigue identificar correctamente los nichos y estimar sus radios con un bajo número de evaluaciones de la función objetivo, y se puede combinar con el método de búsqueda de patrones determinista.

7. Conclusiones

En esta práctica hemos visto como implementar tres algoritmos genéticos diferentes: un algoritmo genético básico, el algoritmo CHC y, por último, mejorar el algoritmo genético básico añadiéndole una función de *clearing* para obtener mayor diversidad genética.

Atendiendo a la tabla de resultados, podemos ver una clara mejora en relación a la primera práctica donde vimos las búsquedas locales. Los algoritmos genéticos mejoran los resultados obtenidos por la búsqueda local drásticamente.

Entre los algoritmos genéticos, destacan los resultados del algoritmo genético básico, ya que obtiene los mejores resultados en ambos problemas. Sin embargo, el número de llamadas a la función de evaluación que realiza es superior al de los otros dos algoritmos poblacionales.
El algoritmo multimodal no ha conseguido alcanzar los resultados obtenidos por sus rivales. Sin embargo, queda cerca de las ganancias generadas por los otros dos algoritmos genéticos.

Finalmente, podemos concluir que, para problemas de optimización, es mejor utilizar algoritmos genéticos que algoritmos de búsqueda local, ya que obtienen mejores resultados. En este problema, concretamente, hemos visto que es mejor utilizar el algoritmo genético básico con un número de evaluaciones muy superior a los otros dos. En caso de querer optimizar el problema y, realizar un número de evaluaciones inferior, deberíamos decantarnos por el algoritmo CHC, que obtiene resultados similares y mucho más rápido.

Referencias

[Dilettoso and Salerno, 2006] Dilettoso, E. and Salerno, N. (2006). A self-adaptive niching genetic algorithm for multimodal optimization of electromagnetic devices. *IEEE Transactions on Magnetics*, 42(4):1203–1206.

[Rathee and Ratnoo, 2020] Rathee, S. and Ratnoo, S. (2020). Feature selection using multi-objective chc genetic algorithm. *Procedia Computer Science*, 167:1656–1664.

[Shopova and Vaklieva-Bancheva, 2006] Shopova, E. G. and Vaklieva-Bancheva, N. G. (2006). Basic—a genetic algorithm for engineering problems solution. *Computers & chemical engineering*, 30(8):1293–1309.