Modelos Bioinspirados y Heurísticas de Búsqueda

Practica 3

"Algoritmos heurísticos no constructivos poblacionales

Algoritmos Genéticos"

Antonio José Morano Moriña

Índice

1-Funcion de evaluación

El objetivo de esta práctica es estudiar el funcionamiento de los algoritmos genéticos: básico, multimodal y CHC, para el problema de la practica1. Los cuales deberán compararse con los resultados de la búsqueda local del mejor vecino de la práctica 1.

Por lo tanto, la función de evaluación que vamos a usar sigue siendo la misma:

```
# FUNCIÓN DE EVALUACIÓN
def fEvaluacion(sActual):
    beneficio = 0
    bateria = 0
    venta = 0
    compra = 0
    for i in range(0, len(sActual)):
        if sActual[i] > 0:  # Venta de energía sobre la c
            energia = generacion[i] + bateria
            venta = energia * sActual[i] / 100
            bateria = energia - venta # Carga de la bater
            if bateria > capacidad: # Si hay más batería
                diferencia = bateria - capacidad
                venta += diferencia # Se vende el exceso
                bateria = capacidad
            beneficio += pVenta[i] * venta # Se calcula e
        elif sActual[i] < 0:  # Compra de energía sobre e
            energia = bateria + generacion[i] # Energia t
            if energia > capacidad:
                bateria = capacidad # Se actualiza la bat
                venta = energia - capacidad # Se vende el
                beneficio += pVenta[i] * venta
            else:
                espacio = capacidad - energia
                compra = espacio * -1 * sActual[i] / 100
                bateria = capacidad - espacio + compra
                beneficio -= pCompra[i] * compra
        else:
            energia = generacion[i] + bateria
            if energia > capacidad:
                venta = energia - capacidad
                bateria = capacidad
                beneficio += pVenta[i] * venta
            else:
                bateria = energia
    venta = bateria
    beneficio += pVenta[23] * venta
    return beneficio
```

2-Funciones auxiliares

Para explicar los códigos de los diferentes algoritmos genéticos implementados, es importante comentar primero las diferentes funciones auxiliares implementadas para el correcto funcionamiento de estos.

```
def creaPoblacion(tamPoblacion):
  poblacion = np.random.randint(-100,100,(tamPoblacion,24) ,dtype=int)
  return poblacion

def evaluaPoblacion(poblacion):
    beneficios = np.zeros(len(poblacion))
    for i in range (len(poblacion)):
        beneficios[i] = fEvaluacion(poblacion[i])
    return beneficios
```

Estas funciones se encargan de crear una población aleatoria con el tamaño pasado por parámetro y de evaluar los beneficios de cada uno de los individuos de la población.

```
def genVecino(sActual, pos, g):
    sVecina = sActual.copy()
    valorPos = sVecina[pos] + g
    if valorPos <= -100:  # No puede pasar de
        sVecina[pos] = -100
    elif valorPos >= 100:
        sVecina[pos] = 100
    else:
        sVecina[pos] = valorPos
    return sVecina

def mutacionIndividuo(individuo):
    mutado = individuo.copy()
    if np.random.random() <= probMutacion:
        # Obtenemos los valores aleatorios
        pos = np.random.randint(0, 24)
        g = np.random.randint(0,10)
        if np.random.uniform() < 0.5:
            mutado = genVecino(mutado,pos,g)
        else:
            mutado = genVecino(mutado,pos,-g)
    return mutado
```

La mutación usa la función genVecino de la practica 1, para crear un individuo mutado, modificando una posición aleatoria en un valor aleatorio con un rango máximo del 10%.

Algoritmo genético generacional

cruce2puntos:

```
def cruce2puntos(padre1,padre2):
    pos1 = np.random.randint(0, len(padre1) - 1)
    pos2 = np.random.randint(0, len(padre2) - 1)

    if pos1 > pos2:
        pos1,pos2 = pos2,pos1

    hijo1 = np.concatenate((padre1[:pos1], padre2[pos1:pos2], padre1[pos2:]))
    hijo2 = np.concatenate((padre2[:pos1], padre1[pos1:pos2], padre2[pos2:]))

    return hijo1, hijo2
```

Consiste en coger dos posiciones aleatorias, que se usan como puntos de corte entre los cromosomas pasados como parámetros, para concatenarlos en función de los puntos de corte y generar unos hijos resultantes del cruce.

seleccionElite:

```
def seleccionElite(poblacion,costesPob, numSeleccionados):
  Ordenados = np.argsort(costesPob)
  seleccionados = poblacion[Ordenados[0:numSeleccionados]]

  return seleccionados
```

Ordenamos la población en función de los costes de sus individuos, dejando a los mejores los primeros, después, seleccionamos simplemente el número que queramos (numSeleccionados) de individuos.

torneo:

```
def torneo(poblacion,costesPoblacion,tamTorneo):
    indices = np.random.choice(len(poblacion), size=tamTorneo, replace=False)
    ganador = indices[np.argmax(costesPoblacion[indices])]
    return poblacion[ganador]
```

Esta función crea un subconjunto de individuos, del tamaño seleccionado para el torneo (tamTorneo), los individuos del concurso son seleccionados aleatoriamente y se selecciona el mejor entre estos, esta función se usará como método de selección para los padres a cruzar.

Algoritmo genético CHC

cruceParentCentered:

```
def cruceParentCentered(padre1, padre2):
    hijo1 = np.zeros_like(padre1)
    hijo2 = np.zeros_like(padre2)
    # Calculamos la media de los genes de los dos padres.
    media = (padre1 + padre2) / 2
    # Generamos los genes de los hijos utilizando la media de los
    for i in range(len(padre1)):
        rand = np.random.rand()
        if rand < 0.5:
            hijo1[i] = media[i] - abs(padre1[i] - padre2[i]) / 2
            hijo2[i] = media[i] + abs(padre1[i] - padre2[i]) / 2
        else:
            hijo1[i] = media[i] + abs(padre1[i] - padre2[i]) / 2
            hijo2[i] = media[i] - abs(padre1[i] - padre2[i]) / 2
        # Ajustamos los valores de los hijos si superan los límit
        hijo1[i] = max(min(hijo1[i], 100), -100)
        hijo2[i] = max(min(hijo2[i], 100), -100)
    return hijo1, hijo2
```

La función toma como entrada `padre1` y `padre2`; y devuelve `hijo1` y `hijo2` generados por el operador de cruce.

El operador se basa en la media de los genes de los dos padres. Para generar los genes de los hijos, se utiliza la media de los padres y se realiza un ajuste en función de la diferencia entre los valores de cada gen de los padres. Si un gen es mayor en un padre que en el otro, entonces el gen correspondiente en uno de los hijos será mayor que la media y el otro será menor que la media.

dEuclidea:

```
def dEuclidea(x, y):
    d = math.sqrt(sum((x[i] - y[i])**2 for i in range(len(x))))
    return d
```

combinarMejores:

```
def combinarMejores(padres, hijos):
    costesP = evaluaPoblacion(padres)
    costesH = evaluaPoblacion(hijos)
    while np.any(costesH > costesP):
        padresOrdenados = np.argsort(costesP)
        hijosOrdenados = np.argsort(costesH)[::-1]
        for i in range(len(hijosOrdenados)):
            if costesH[hijosOrdenados[i]] > costesP[padresOrdenados[i]]:
                padres[padresOrdenados[i]] = hijos[hijosOrdenados[i]]
                costesP[padresOrdenados[i]] = costesH[hijosOrdenados[i]]
    return padres
```

Esta función es el operador de recombinación del algoritmo CHC, y se encarga de recibir 2 poblaciones (padres e hijos), y devuelve la población de los padres, cambiando aquellas posiciones que obtuvieron peor beneficio que sus hijos, por sus hijos.

La función repite este proceso hasta que los costos de los individuos en la población cruzada sean iguales o peores que los de la población original.

Algoritmo genético multimodal

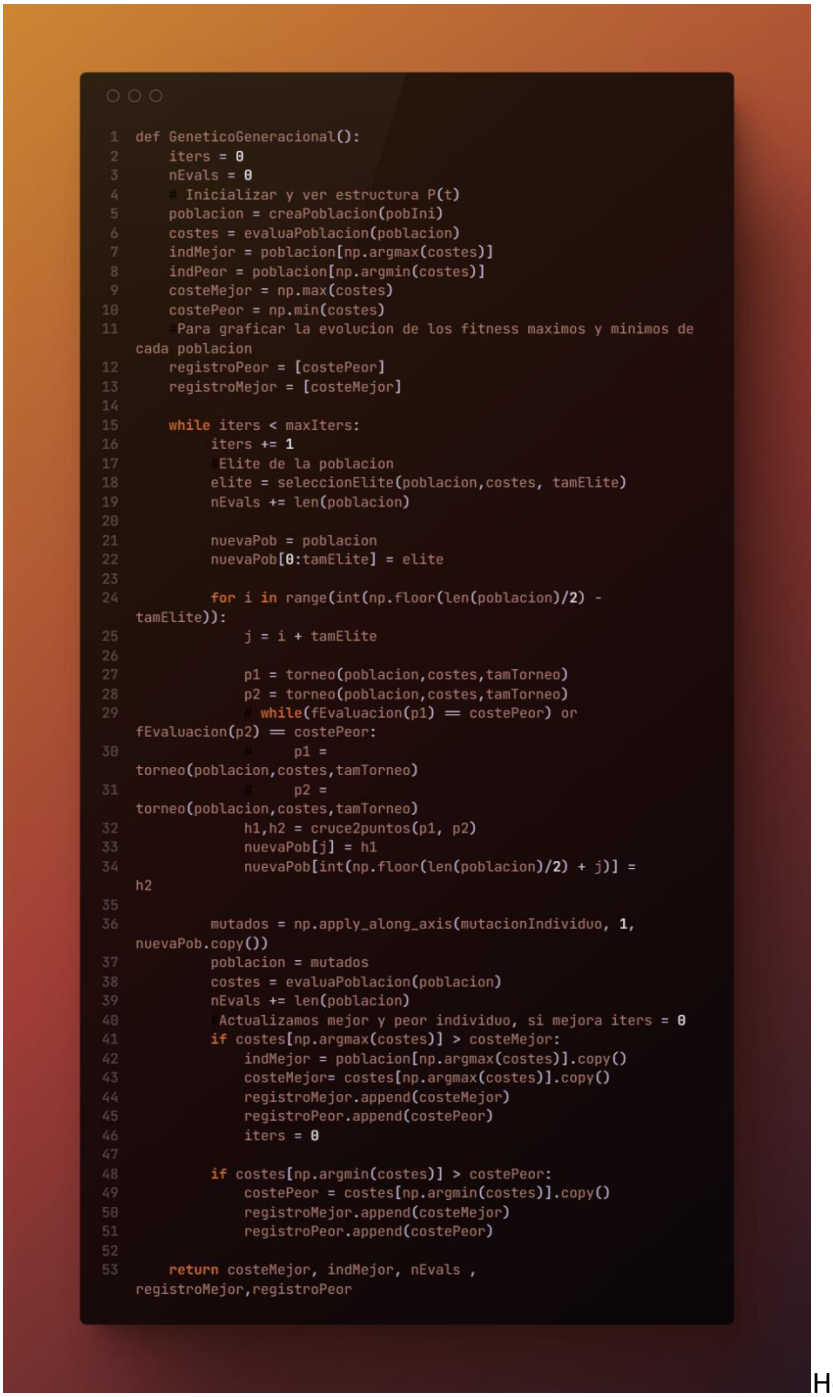
clearing:

```
def clearing(pob, kappa, dist):
    tamPob = len(pob)
    costes = evaluaPoblacion(pob)
    orden = np.argsort(costes)[::-1]
    for i in range(tamPob):
        if costes[orden[i]] > 0:
            numGanadores = 1
            for j in range(i + 1, tamPob):
                if costes[orden[j]] > 0 and dEuclidea(pob[orden[i]], pob[orden[j]]) < dist:
                    if numGanadores < kappa:
                        numGanadores += 1
                    else:
                        costes[orden[j]] = 0
    nuevaPob = []
    for i in orden:
        if costes[i] != 0:
            nuevaPob.append(pob[i])

    print(len(nuevaPob))
    return nuevaPob
```

En esta función se implementa la técnica de clearing, en la cual primero ordenamos la población en función de los beneficios de cada individuo. Luego, itera sobre los individuos en orden descendente y asigna un número de "ganadores" igual a 1 al primer individuo. Luego, se compara este individuo con los siguientes en la lista ordenada y se asigna un número de ganadores adicional si la distancia euclidiana entre los individuos es menor que `dist`. Si el número de ganadores para un individuo supera `kappa`, entonces se le asigna un coste de 0, lo que indica que no será incluido en la población resultante. Finalmente, se devuelve una población filtrada que incluye solo los individuos cuyo coste no es 0.

3-Algoritmos genéticos

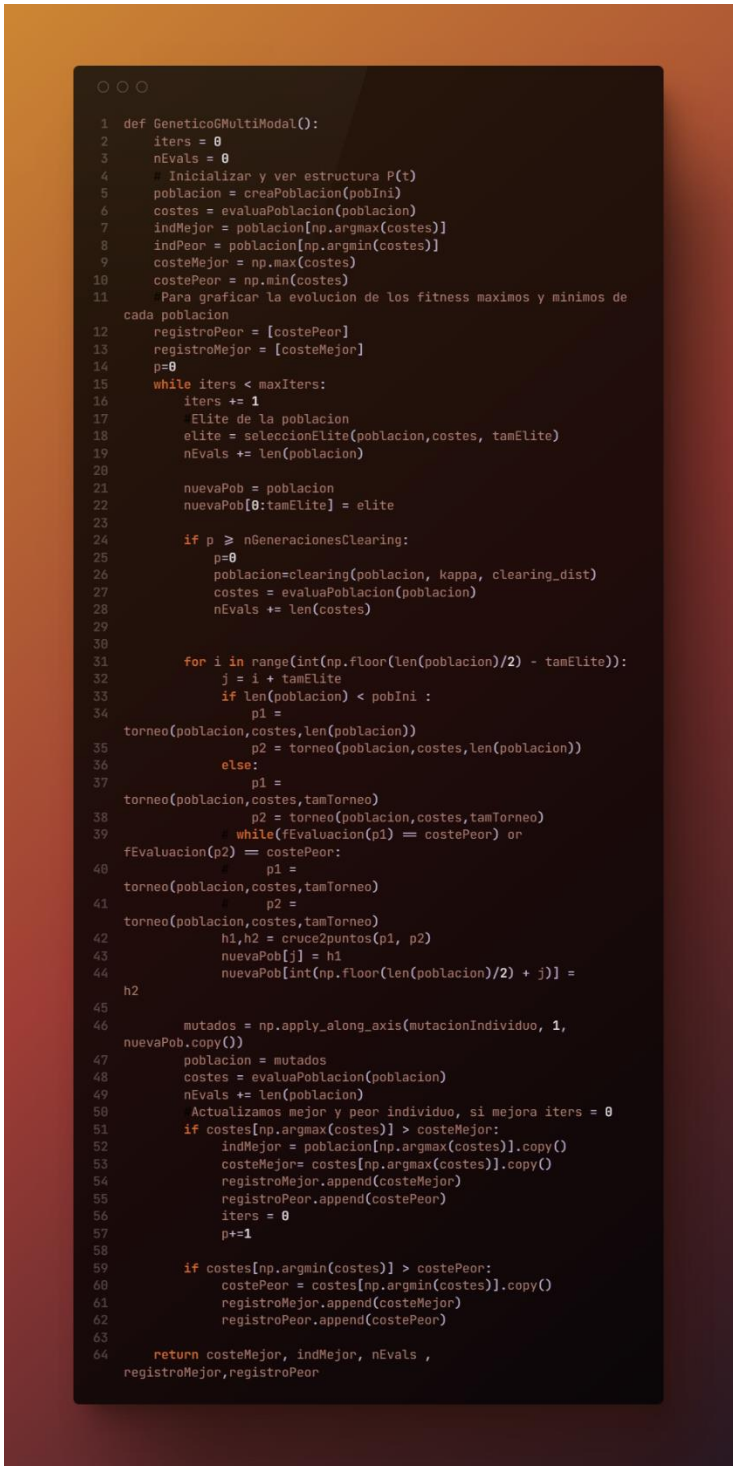
Básico Generacional



```python
def GeneticoGeneracional():
    iters = 0
    nEvals = 0
    # Inicializar y ver estructura P(t)
    poblacion = creaPoblacion(pobIni)
    costes = evaluaPoblacion(poblacion)
    indMejor = poblacion[np.argmax(costes)]
    indPeor = poblacion[np.argmin(costes)]
    costeMejor = np.max(costes)
    costePeor = np.min(costes)
    # Para graficar la evolucion de los fitness maximos y minimos de cada poblacion
    registroPeor = [costePeor]
    registroMejor = [costeMejor]

    while iters < maxIters:
        iters += 1
        # Elite de la poblacion
        elite = seleccionElite(poblacion,costes, tamElite)
        nEvals += len(poblacion)

        nuevaPob = poblacion
        nuevaPob[0:tamElite] = elite

        for i in range(int(np.floor(len(poblacion)/2) - tamElite)):
            j = i + tamElite

            p1 = torneo(poblacion,costes,tamTorneo)
            p2 = torneo(poblacion,costes,tamTorneo)
            # while(fEvaluacion(p1) == costePeor) or fEvaluacion(p2) == costePeor:
            #     p1 = torneo(poblacion,costes,tamTorneo)
            #     p2 = torneo(poblacion,costes,tamTorneo)
            h1,h2 = cruce2puntos(p1, p2)
            nuevaPob[j] = h1
            nuevaPob[int(np.floor(len(poblacion)/2) + j)] = h2

        mutados = np.apply_along_axis(mutacionIndividuo, 1, nuevaPob.copy())
        poblacion = mutados
        costes = evaluaPoblacion(poblacion)
        nEvals += len(poblacion)
        # Actualizamos mejor y peor individuo, si mejora iters = 0
        if costes[np.argmax(costes)] > costeMejor:
            indMejor = poblacion[np.argmax(costes)].copy()
            costeMejor= costes[np.argmax(costes)].copy()
            registroMejor.append(costeMejor)
            registroPeor.append(costePeor)
            iters = 0

        if costes[np.argmin(costes)] > costePeor:
            costePeor = costes[np.argmin(costes)].copy()
            registroMejor.append(costeMejor)
            registroPeor.append(costePeor)

    return costeMejor, indMejor, nEvals , registroMejor,registroPeor
```

He implementado el generacional, cada iteración se crea una población nueva, a partir de los 5 mejores de la anterior (elitismo) y los cruces formados por los padres elegidos mediante torneo, añadimos estos hijos resultantes de los cruces a los individuos de la elite anterior, y aplicamos el operador de mutación. Comprobamos si mejora el mejor de la nueva población, si no lo hace volvemos a iterar, y si lo hace iters=0. Haciendo así que solo acabe cuando llegue al límite de iteración sin mejorar

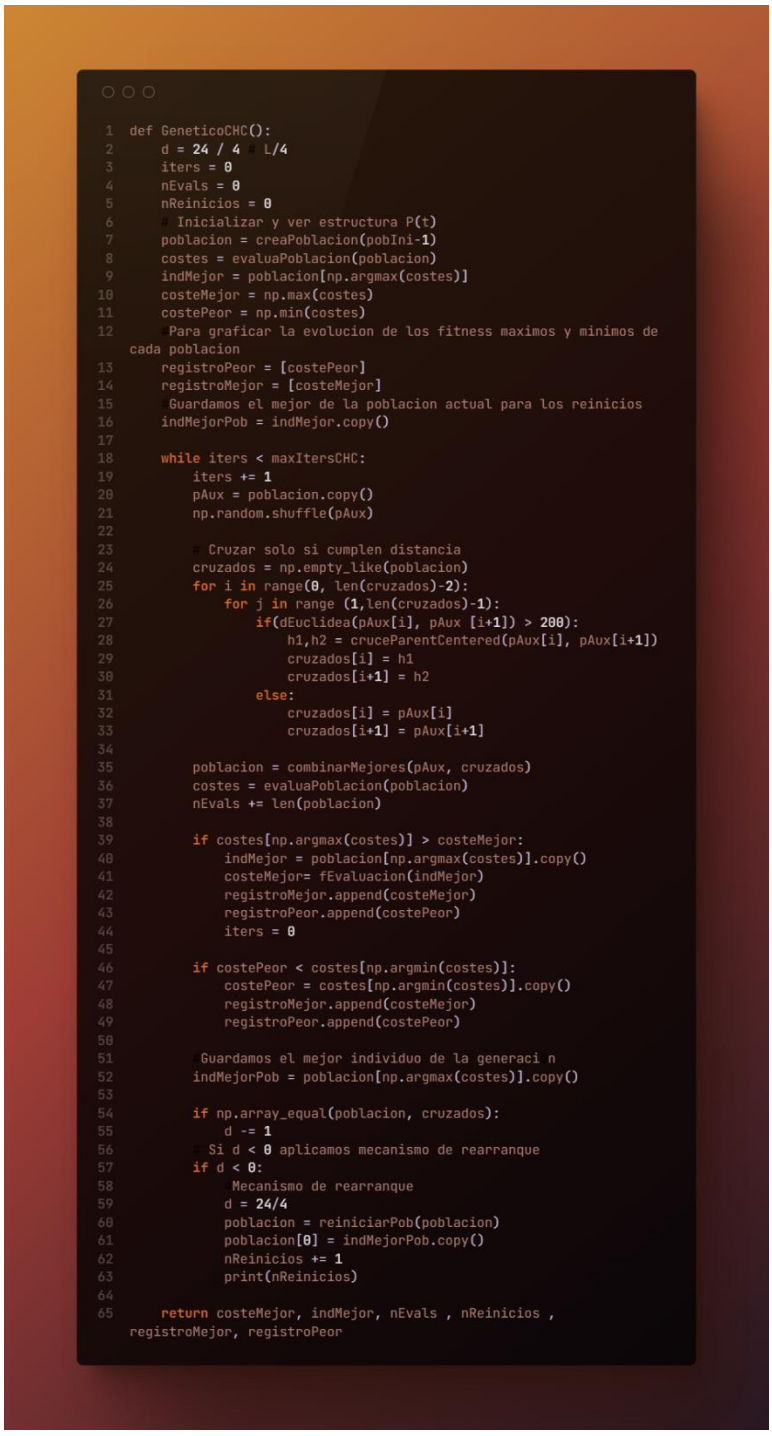
MultiModal



```
def GeneticoGMultiModal():
    iters = 0
    nEvals = 0
    # Inicializar y ver estructura P(t)
    poblacion = creaPoblacion(pobIni)
    costes = evaluaPoblacion(poblacion)
    indMejor = poblacion[np.argmax(costes)]
    indPeor = poblacion[np.argmin(costes)]
    costeMejor = np.max(costes)
    costePeor = np.min(costes)
     Para graficar la evolucion de los fitness maximos y minimos de
cada poblacion
    registroPeor = [costePeor]
    registroMejor = [costeMejor]
    p=0
    while iters < maxIters:
        iters += 1
         Elite de la poblacion
        elite = seleccionElite(poblacion,costes, tamElite)
        nEvals += len(poblacion)

        nuevaPob = poblacion
        nuevaPob[0:tamElite] = elite

        if p ≥ nGeneracionesClearing:
            p=0
            poblacion=clearing(poblacion, kappa, clearing_dist)
            costes = evaluaPoblacion(poblacion)
            nEvals += len(costes)


        for i in range(int(np.floor(len(poblacion)/2) - tamElite)):
             j = i + tamElite
             if len(poblacion) < pobIni :
                 p1 =
torneo(poblacion,costes,len(poblacion))
                 p2 = torneo(poblacion,costes,len(poblacion))
             else:
                 p1 =
torneo(poblacion,costes,tamTorneo)
                 p2 = torneo(poblacion,costes,tamTorneo)
             # while(fEvaluacion(p1) == costePeor) or
fEvaluacion(p2) == costePeor:
             #     p1 =
torneo(poblacion,costes,tamTorneo)
             #     p2 =
torneo(poblacion,costes,tamTorneo)
             h1,h2 = cruce2puntos(p1, p2)
             nuevaPob[j] = h1
             nuevaPob[int(np.floor(len(poblacion)/2) + j)] =
h2

        mutados = np.apply_along_axis(mutacionIndividuo, 1,
nuevaPob.copy())
        poblacion = mutados
        costes = evaluaPoblacion(poblacion)
        nEvals += len(poblacion)
         Actualizamos mejor y peor individuo, si mejora iters = 0
        if costes[np.argmax(costes)] > costeMejor:
             indMejor = poblacion[np.argmax(costes)].copy()
             costeMejor= costes[np.argmax(costes)].copy()
             registroMejor.append(costeMejor)
             registroPeor.append(costePeor)
             iters = 0
             p+=1

        if costes[np.argmin(costes)] > costePeor:
             costePeor = costes[np.argmin(costes)].copy()
             registroMejor.append(costeMejor)
             registroPeor.append(costePeor)

    return costeMejor, indMejor, nEvals ,
registroMejor,registroPeor
```

La implementación es prácticamente igual que la del básico, solo que en este caso, llamaremos a clearing() que se encarga de reducir la población dejando solo aquellos que estén cercanos entre si y exploran regiones del espacio más interesantes, y eliminando aquellos que se encuentren lejos (ya que están en una región del espacio de búsqueda lejana y sus beneficios son peores).

CHC



```python
def GeneticoCHC():
    d = 24 / 4 # L/4
    iters = 0
    nEvals = 0
    nReinicios = 0
    # Inicializar y ver estructura P(t)
    poblacion = creaPoblacion(pobIni-1)
    costes = evaluaPoblacion(poblacion)
    indMejor = poblacion[np.argmax(costes)]
    costeMejor = np.max(costes)
    costePeor = np.min(costes)
    #Para graficar la evolucion de los fitness maximos y minimos de cada poblacion
    registroPeor = [costePeor]
    registroMejor = [costeMejor]
    #Guardamos el mejor de la poblacion actual para los reinicios
    indMejorPob = indMejor.copy()

    while iters < maxItersCHC:
        iters += 1
        pAux = poblacion.copy()
        np.random.shuffle(pAux)

        # Cruzar solo si cumplen distancia
        cruzados = np.empty_like(poblacion)
        for i in range(0, len(cruzados)-2):
            for j in range (1,len(cruzados)-1):
                if(dEuclidea(pAux[i], pAux [i+1]) > 200):
                    h1,h2 = cruceParentCentered(pAux[i], pAux[i+1])
                    cruzados[i] = h1
                    cruzados[i+1] = h2
                else:
                    cruzados[i] = pAux[i]
                    cruzados[i+1] = pAux[i+1]

        poblacion = combinarMejores(pAux, cruzados)
        costes = evaluaPoblacion(poblacion)
        nEvals += len(poblacion)

        if costes[np.argmax(costes)] > costeMejor:
            indMejor = poblacion[np.argmax(costes)].copy()
            costeMejor= fEvaluacion(indMejor)
            registroMejor.append(costeMejor)
            registroPeor.append(costePeor)
            iters = 0

        if costePeor < costes[np.argmin(costes)]:
            costePeor = costes[np.argmin(costes)].copy()
            registroMejor.append(costeMejor)
            registroPeor.append(costePeor)

        #Guardamos el mejor individuo de la generaci n
        indMejorPob = poblacion[np.argmax(costes)].copy()

        if np.array_equal(poblacion, cruzados):
            d -= 1
        # Si d < 0 aplicamos mecanismo de rearranque
        if d < 0:
            # Mecanismo de rearranque
            d = 24/4
            poblacion = reiniciarPob(poblacion)
            poblacion[0] = indMejorPob.copy()
            nReinicios += 1
            print(nReinicios)

    return costeMejor, indMejor, nEvals , nReinicios , registroMejor, registroPeor
```

El algoritmo CHC, no aplica una mutación, si no que genera 2 poblaciones, una con los padres y otra resultante de los cruces de estos padres (aquellos que estén a cierta distancia entre sí solamente), una vez tenemos esas dos población, usamos la función combinarMejores(), para obtener una nueva población compuesta por los mejores individuos de las poblaciones anteriores.

Si la población generada es igual que la obtenida en el cruce, restaremos uno a d, una vez esta sea menor que 0, usaremos el mecanismo de rearranque, donde llamaremos a la función crearPoblacion() para crear una nueva población desde 0, a la que le tenemos que añadir el mejor individuo de la generación anterior.

4-Valores parámetros seleccionados

```
#Parametros
pobIni = 30
probMutacion = 0.05
maxIters = 5000
tamElite = 5
tamTorneo = 15
# CHC
maxItersCHC = 2500
# Multimodal
clearing_dist = 25
kappa = 2
nGeneracionesClearing = 100
```

Estos han sido los mejores parámetros, tras realizar diferentes experimentaciones.

Para la selección de los que creo valores óptimos simplemente me he basado en los que me hicieran a los algoritmos encontrar soluciones mejores.

pobIni -> Los beneficios encontrados mejoran conforme se aumenta el tamaño, entre 25 y 30 los cambios ya son menores. Aun así, la diferencia tampoco es muy grande en cuanto al mejor valor encontrado.

maxIters y *maxItersCHC* -> Para ambos parámetros he seleccionado el valor con el cual la ejecución tardaba menos, siempre y cuando los cambios en el mejor beneficio encontrado no fueran notables.

La experimentación de estos parámetros es importante, ya que con esto buscamos encontrar un equilibrio entre la eficiencia y la efectividad del algoritmo, hacemos esto buscando un valor que permita que el algoritmo converja en un numero razonable de iteraciones, ni muy pronto ni muy tarde.

probMutacion -> Para determinar el valor de este parámetro, he tenido en cuenta que, a menor probabilidad de mutación, es mas posible que es algoritmo tengo dificultades para escapar de óptimos locales, mientras que si es muy alta es mas probable que se aleje de la solución actual, explorando demasiado el espacio y perdiendo información sobre las poblaciones pasadas.

El valor seleccionado es el que mejores resultados daba, bajarlo mas no varía apenas las soluciones y aumentarlo solamente empeora los beneficios obtenidos.

5-Experimentacion

Ejemplo de salida de los algoritmos:

```
(535.5437712960002, array([ -78,  -44,  -39,  -21,
-67,   -9,   98,   42,  -78,  -92,  -54,
       -100,   68,  -89,   72,   -3,  -51,  -26,
-17,  -37,  -33,   82,
         14,   60]), 73602, 362,
[409.56857371868125, 409.56857371868125,
409.56857371868125, 427.05426085423545,
434.759358232, 465.8486943676801, 478.4674014017,
478.4674014017, 508.4726160134126,
508.4726160134126, 508.4726160134126,
512.283157155, 529.0679217930998,
530.4133781112289, 534.6833121354471,
535.5437712960002], [135.1790495657284,
138.22057188080004, 203.24233552788374,
203.24233552788374, 203.24233552788374,
203.24233552788374, 203.24233552788374,
225.60735974500002, 225.60735974500002,
228.67432644915715, 255.18103737600006,
255.18103737600006, 255.18103737600006,
255.18103737600006, 255.18103737600006,
255.18103737600006])
```

Esta es la salida al hacer un print de la funcion que desarrolla el algoritmo ejecutado.

Subrayado en rojo se encuentra el mejor valor encontrado, seguido por el individuo que genera dicho valor.

En verde el numero de evaluaciones realizadas, los dos arrays largos siguientes corresponden a los históricos del mejor y el peor individuo.

Tabla de resultados – Problema Real

Algoritmo	Ev.Medias	Ev.Mejor	σ EV	Mejor Fitness	Media Fitness	σ fitness
B.Local	5689	4654	1436	331,81	321,3	18,19
G. Básico	521720	409080	81065	333,96	333,15	0,446
G.Multimodal	160128	97320	88217	333,75	331,18	4,32
CHC	89129	75150	7440	332,41	330,39	2,30

G.Básico, es el algoritmo que encuentra mejores resultados de forma media, también encuentra el mayor de los beneficios y tiene una desviación mínima, por lo que estamos ante un algoritmo muy robusto y estable. Esto quiere decir que tiene un comportamiento predecible y es probable que produzca soluciones similares.

G.MultiModal, tiene un desempeño muy similar al genético básico, encuentra soluciones muy parecidas en calidad, solo que es menos consistente en estas mismas. Esto podemos verlo en su desviación típica que aumenta con respecto a la del genético básico.

G.CHC, es el que peor resultados da de los algoritmos genéticos, de media da mejores soluciones que la búsqueda local, la desviación del algoritmo CHC es bastante menor que la de la búsqueda local..

Por último, me gustaría comparar los 3 algoritmos genéticos con la búsqueda local de la practica1, todos encuentran mejores soluciones que la BL de forma general y todos tienen una desviación típica bastante menor. Por lo tanto, podemos decir que los algoritmos genéticos son bastantes más estables y consistentes para el problema dado que la búsqueda local, en este caso, la búsqueda local del mejor vecino

Tabla de resultados – Problema aleatorizado

Algoritmo	Ev.Medias	Ev.Mejor	σ EV	Mejor Fitness	Media Fitness	σ fitness
B.Local	8281	7691	1194	538,2	521,33	19,98
G. Básico	325641	242049	98065	626,2	623,1	3,119
G.Multimodal	89562	84534	7219	603,16	595,146	12,12
CHC	102192	75120	10527	661,58	635,467	32,62

Como es comprensible los resultados han variado, así como el orden de aptitud de cada uno de los algoritmos.

En este problema el mejor algoritmo en cuanto a calidad de las soluciones es el genético CHC, sin embargo, la desviación sube bastante, esta vez no hay un algoritmo que sea mejor en todo, ya que CHC encuentra buenas soluciones, pero es poco robusto y da soluciones un poco alejadas entre ellas; lo mismo pasa con el algoritmo Básico, el cual da soluciones algo más consistentes pero encuentra soluciones de peor calidad.

El multimodal es el que baja sus resultados en este problema, no hasta el punto asemejarse a la búsqueda local, pero si baja bastante en comparación con los valores medios encontrados por los otros dos algoritmos genéticos.

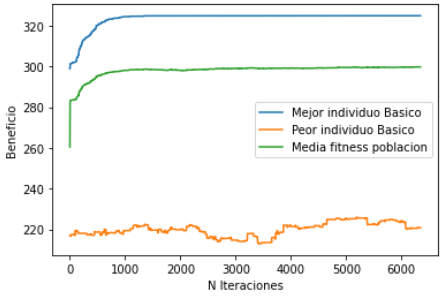
En cuanto a la eficiencia computacional de los algoritmos, podemos ver que es igual para ambos problemas, el algoritmo genético básico tiene un gran coste ya que llama muchas mas veces que los demás a la función de evaluación, seguido por el genético multimodal y el CHC, que no están muy lejos entre ellos con respecto a sus valores medios, aunque la desviación de estos es bastante distinta, teniendo el multimodal un coste computacional mas estable.

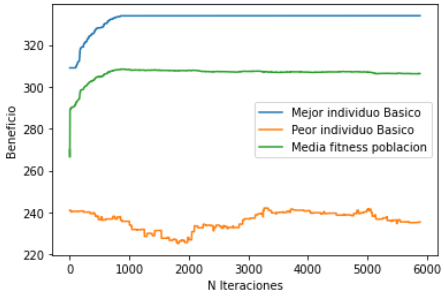
6-Estudio convergencia

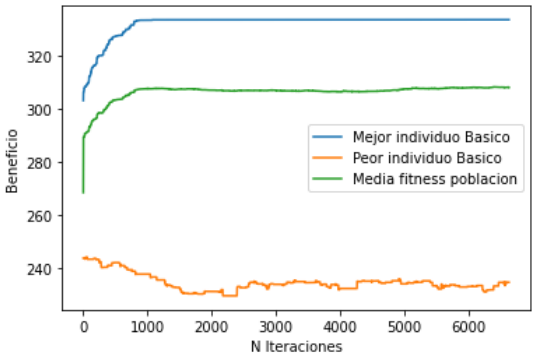
En la práctica se pide un estudio sobre la convergencia de los individuos en los diferentes algoritmos, los resultados obtenidos han sido los siguientes:

GENETICO GENERACIONAL

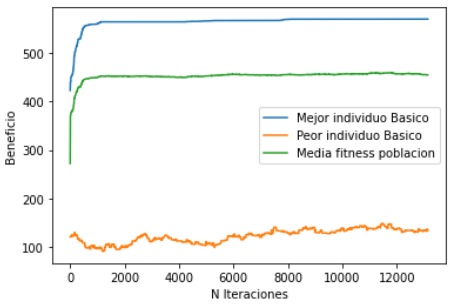
Problema con datos reales



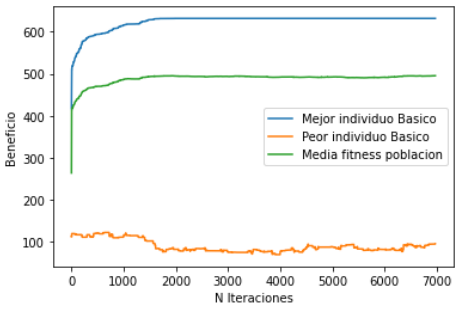




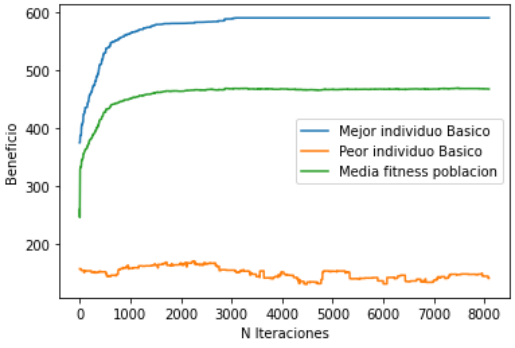
Problema con datos aleatorizado



semilla 10



semilla 5



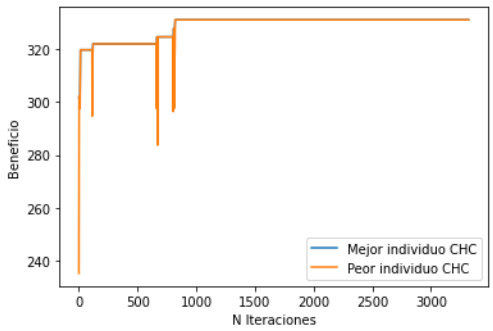
semilla 100

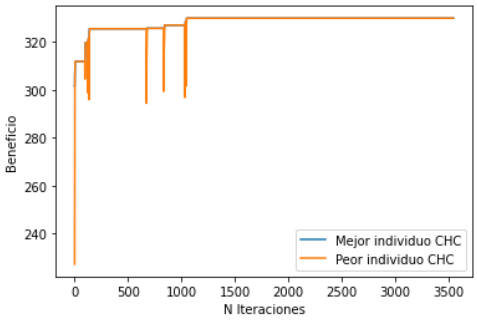
Como se puede observar en las gráficas, el algoritmo no tiene problema en encontrar individuos mejores al principio, aunque luego tiende a estancarse para encontrar mejores soluciones, al ser generacional, el peor individuo de cada población no obtiene muy buenos valores ya que en cada iteración se genera una población distinta, como no sabía por qué no convergía el peor individuo, he representado la evolución del valor medio de las poblaciones para comprobar que se estuvieran habiendo reemplazos entre ellas.

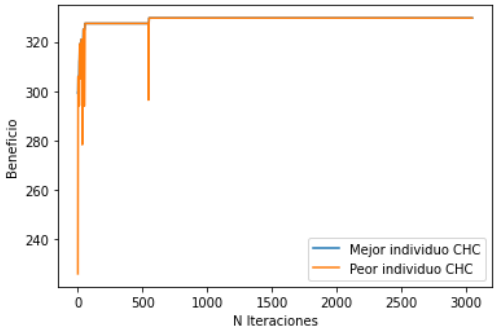
Como cada iteración la población resultante es igual o mejor que la anterior, podríamos decir que el algoritmo mejora la población pero tiende a dejar a individuos con valores bajos vivos entre cada una de las población.

GENETICO CHC

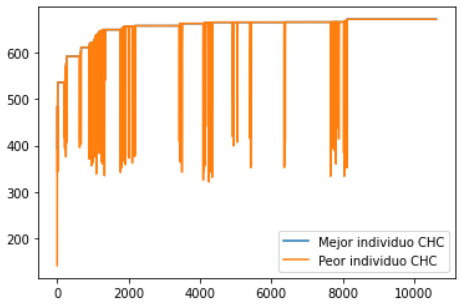
Problema con datos reales

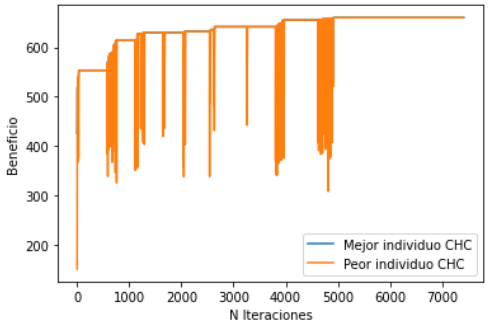


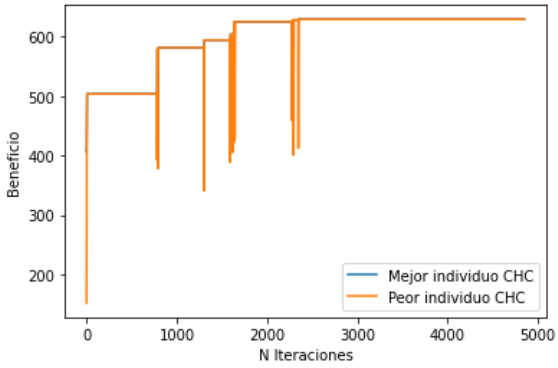




Problema con datos aleatorizado







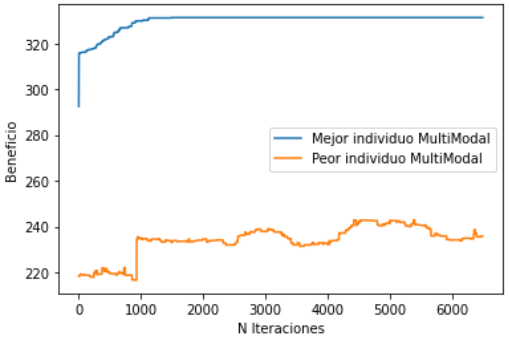
Como podemos observar las graficas resultantes para ambos problemas son similares, en ambos realiza una serie de re inicializaciones, a causa del algoritmo CHC en sí, que tiene un

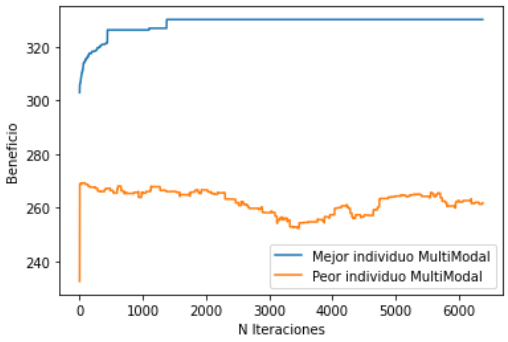
método de arranque que hace que se reinicie la población generada hasta el momento por completo, haciendo que el valor varie de forma notable, dando así lugar a los picos hacia abajo que sufre el peor de los individuos de cada población, ya que el mejor es guardado como primero en la población que surge al reiniciar, de ahí que sufra estos cambios.

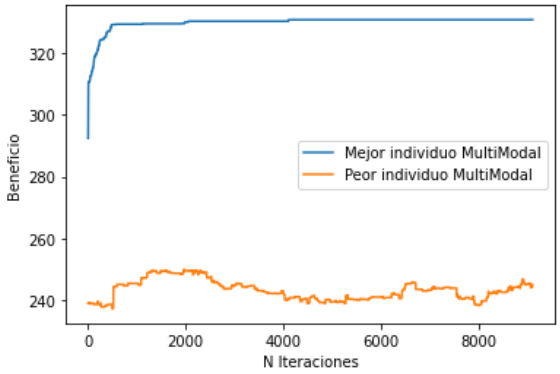
Cabe destacar que el valor de la distancia mínima para que se produjera el cruce entre dos padres es diferente en cada uno de los problemas, para el problema con datos reales he usado una distancia de 25 mientras que para los datos aleatorizados de 15.

GENETICO MULTIMODAL

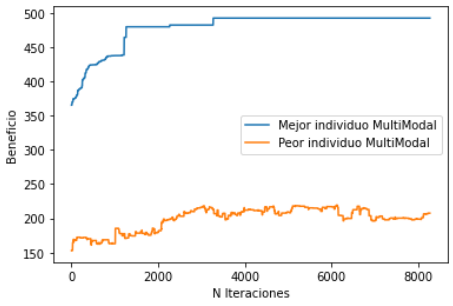
Problema con datos reales



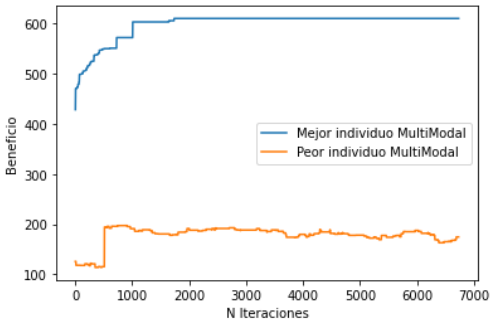




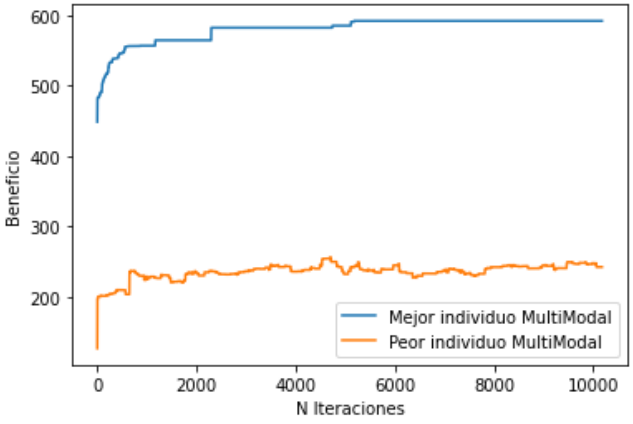
Problema con datos aleatorizado



semilla10



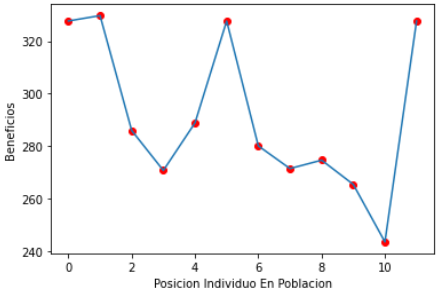
semilla 50



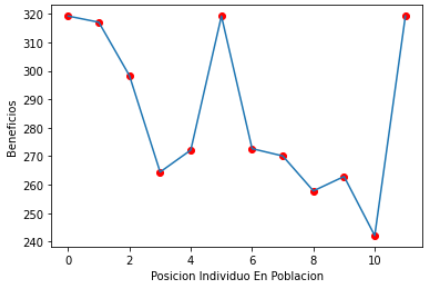
semilla 100

Observando las graficas del multimodal, podemos decir que son similares a las del generacional básico, excepto en la convergencia del peor individuo.

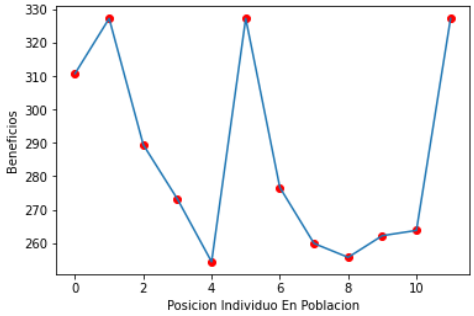
En cuanto a los valores de los peores individuos, ambos algoritmos pueden converger a valores similares si se utilizan tasas de mutación y operadores de selección adecuados. En general, se espera que los valores de los peores individuos aumenten a medida que avanza la búsqueda en ambos algoritmos, aunque estos valores pueden fluctuar debido a la aleatoriedad intrínseca del proceso.



Semilla 10

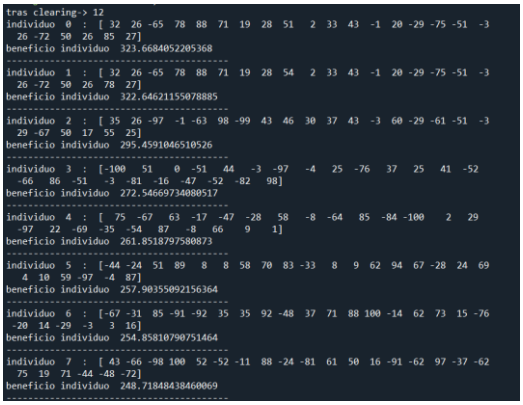


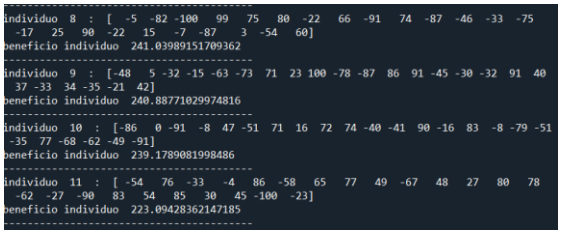
Semilla 50



Semilla 100

Salida por consola de los resultados:





Estas graficas muestran a la población resultante al acabar de ejecutar el algoritmo multimodal, cada uno de los puntos representa el beneficio obtenido de todos los individuos que sobreviven a la distancia de clearing

La razón principal es que el proceso de clearing tiene como objetivo eliminar individuos que estén muy cerca en el espacio de búsqueda para aumentar la diversidad en la población. Aunque, esto no garantiza que todos los individuos cercanos tengan el mismo beneficio.

Además, los beneficios pueden verse afectados por otros factores como la forma en que se codifican los individuos, la función de fitness utilizada, la forma en que se implementan los

operadores genéticos (cruce y mutación), entre otros. Por lo tanto, es común que algunos individuos cercanos tengan diferentes beneficios incluso después del proceso de clearing.

En todas las ejecuciones con cada una de las semillas ha dado 12 individuos resultantes tras realizar el clearing a la población. Por lo que el clearing elimina un 70% de la población con los valores utilizados, pasa lo mismo con los valores que en las distancias de cruce del CHC, para los datos reales clearing_dist = 0,15 y para datos aleatorios clearing_dist = 0,25, solo que el clearing en los datos aleatorios deja entre 8 y 17 individuos vivos según la iteración, por lo que he puesto solo las graficas de los datos reales ya que representan mejor el impacto de la función de clearing sobre la población resultante.

También he añadido dos imágenes, que corresponden a la misma iteración del algoritmo multimodal, y muestra los individuos que sobreviven al clearing junto con sus beneficios. Es lo mismo que las imágenes, pero para que se vea bien la población que sobrevive y las diferencias que existen entre ellos.

7-Bibliografía científica

Daniel, O. A. & Guillermo, F. M. & Gustavo, M. y Juan, A. S. (2009). Optimización de Redes Eléctricas Mediante la Aplicación de Algoritmos Genéticos. Información Tecnológica, 20(4), 137-148.

En este trabajo, se estudia la optimización de la distribución de redes eléctricas, aplicando algoritmos genéticos para minimizar las perdidas técnicas resultantes según el efecto Joule. *(Fenómeno por el que los electrones en movimiento de una corriente eléctrica impactan contra el material a través del cual están siendo conducidos. La energía cinética que tienen los electrones se convierte entonces en energía térmica, calentando el material por el que circulan.)*

Rincón O., J. C., (2006). APLICACIÓN DE ALGORITMOS GENÉTICOS EN LA OPTIMIZACIÓN DEL SISTEMA DE ABASTECIMIENTO DE AGUA DE BARQUISIMETO-CABUDARE. Avances en Recursos Hidráulicos, (14), 25-38.

En este trabajo, se propone el uso de los algoritmos genéticos, para llevar a cabo la optimización de los costos y el tiempo requeridos para llevar a cabo un proyecto de expansión de sistemas de abastecimiento de agua. Aplicado específicamente para el sistema que abastece a las ciudades venezolanas Barquisimeto y Cabudare.

Ledo, L. V., & Roque, Y. A. (2013). Algoritmos Genéticos aplicados a la optimización de antenas. *Telemática*, *12*(3), 21-31.

En este trabajo, se aplican algoritmos genéticos para la optimización en el proceso y diseño en antenas, para tres tipos distintos, que son la bocina piramidal, el reflector parabólico y el parche circular

Catania, C., & Garino, C. G. (2008). Reconocimiento de patrones en el tráfico de red basado en algoritmos genéticos. *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, *12*(37), 65-75.

En este trabajo, se aplican los algoritmos genéticos, con la finalidad de categorizar documentos de forma automática, durante el desarrollo se introducen cinco operadores distintos que son diseñados para este problema en especifico y salen de las implementaciones básicas de los algoritmos genéticos.