[]

BIOGEOGRAPY BASED OPTIMIZATION

Pablo Huertas Arroyo

21 de junio de 2022

UNIVERSIDAD DE GRANADA

Correo: phuertas@correo.ugr.es
DNI:77033078Y
Grupo 3A, subgrupo 2

Índice

1. RESUMEN

1.1. Algoritmo BBO

El algoritmo BBO (Biogeography Based Optimization) es un algoritmo evolutivo que optimiza una funcion de forma estocastica e iterativa mejorando las soluciones candidatas con respecto a una medida de calidad llamado fitness.

BBO optimiza un problema manteniendo una poblacion de soluciones candidatas(elitismo), y creando nuevas soluciones con dos procesos llamados migracion y mutacion que se encargan de mejorar las soluciones nuevas generadas, Estas nuevas soluciones serán candidatas si son de buena calidad.

La Bio-Geografía es el estudio de la extinción y la distribución geográfica de las especies biológicas, cuyos modelos matemáticos describen la evolución de nuevas especies, la migración de especies entre islas vecinas y la extinción de las mismas. Para poder comprender el comportamiento de la metaheurística BBO es necesario conocer definiciones y aspectos fundamentales que serán primordiales en el proceso de desarrollo de este algoritmo. A continuación se presenta una revisión de ciertos factores tales como, los índices de Migración (λ, μ), el índice de adecuación de un hábitat HSI (Habitat Suitability Index), y las variables de idoneidad SIVs (Suitability Index Variables), los cuales son las bases de este método de optimización.

Un área geográfica que se adapta mejor como residencia para albergar especies biológicas se dice que es un hábitat que tiene un alto índice de adecuación (HSI), por lo tanto, un hábitat que tiene un alto HSI puede estar compuesto por una gran diversidad de recursos, los cuales pueden ser, cascadas, diversidad topográfica, diversidad vegetativa, áreas de tierra, ríos, lagos, etc. De esta manera, surge un nuevo término denominado variables de idoneidad (SIVs), que son variables independientes que representan las características de habitabilidad de una isla.

Aquellos hábitats que tienen un alto HSI son capaces de hospedar a un mayor número de especies que aquellos hábitats que tienen un bajo HSI; de la misma forma se tiene que los índices de migración (inmigración y emigración) varían con respecto al HSI. Así podemos determinar que aquellos hábitats con un alto HSI tienen un alto índice de emigración debido principalmente al número de especies que el hábitat puede tener, mientras que el índice de inmigración resulta bajo ya que el hábitat puede contener demasiadas especies como para albergar otras provenientes de islas vecinas. Esto puede ocasionar que las nuevas especies no sobrevivan debido a que existiría una gran competencia por los recursos.

Por otro lado, el índice de inmigración en hábitats que tienen un bajo HSI es alto debido fundamentalmente a que este tipo de hábitats tienen una población pequeña, por lo tanto, disponen de áreas libres que podrían albergar nuevas especies. Hay que resaltar que el alto índice de inmigración ocurre por lo antes expuesto, más no porque especies de islas vecinas quieran llegar a estos hábitats, ya que después de todo, las características que tienen estos hábitats son indeseables para nuevas especies.

λ representa el índice de inmigración, mientras que μ representa el índice de emigración, ambos índices se encuentran en función del número de especies de una isla, S_o representa el punto de equilibrio donde λ es igual a μ, y S_{max} representa el máximo número de especies que la isla puede soportar.

Si observamos la curva de inmigración representado por λ se puede notar que la tasa máxima de inmigración representada por I , ocurre cuando no existen especies en la isla, y a medida que el número de especies va incrementando, la tasa de inmigración va disminuyendo, debido a que la isla se va llenando, y cada vez menos especies sobrevivirán al proceso de inmigración; por otro lado, se verifica también que cuando una isla alberga el máximo número de especies que esta puede soportar (S_{max}) la tasa de inmigración es cero.

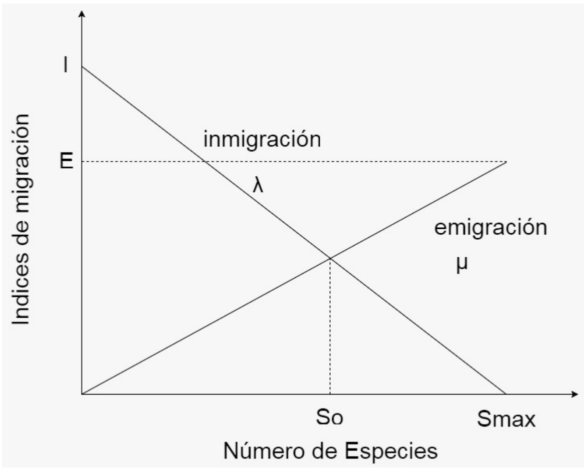


Figura 1: Curva de emigracion/inmigración

Ahora consideremos la curva de emigración μ, en donde si no existen especies en la isla la tasa de emigración es cero, y mientras el número de especies en la isla va incrementando la tasa de emigración también lo hace, lo cual quiere decir que a medida que aumenta el número de especies en la isla, más especies están disponibles para emigrar a islas vecinas. La tasa máxima de emigración E , ocurre cuando la isla alberga al máximo número de especies que esta puede soportar.

En resumen, se presentan una serie de diferentes factores que serán las bases de dicho metodo de optimizacion

- **Fitness**: Se utiliza para calcular la calidad de las soluciones. (En nuestro caso la dispersion)
- μ : Indice de emigracion de una poblacion en concreto. Esta variable indica cuanto probable va a ser que algun elemento de dicha solucion emigre a otra.
- λ : Indice de inmigracion de una poblacion en concreto. Esta variable indica cuanto probable va a ser que algun elemento de dicha solucion sea reemplazado

por un elemento de otra solucion.

- **Isla** : En nuestro caso de trata de una poblacion
- **HSI** : Habitat Suitability Index. índice numérico que representa la capacidad de un hábitat de mantener un cierto nivel de población de diferentes especies. Se obtiene de la combinación de las diferentes variables que afectan a la calidad de vida de las especies que habitan ese hábitat.
- **SIV** : Suitability Index Variables. Variables independientes que representan las caracteristicas de habitabilidad de una isla. (En nuestro cada SIV sera cada una de las soluciones de la poblacion)

Existen tres mecanismos principales de este algoritmo, los cuales son:

- **Elitismo** : Antes de generar nuevas soluciones se guardan un conjunto de soluciones que sean las mejores. Estas soluciones seran introducidas al finalizar por las peores de las nuevas generadas.
- **Migracion** : Se realiza una migracion de una solucion a otra. Por probabilidad los elementos de las mejores soluciones se migran a otras soluciones.
- **Mutacion** : Se realiza una mutacion de una solucion. Por probabilidad se cambia un elemento de una solucion por otro. Se debe conservar la factibilidad de las soluciones.

1.2. Problema donde se prueba el BBO

El problema elegido a abordar en esta practica es el siguiente: Problema de la mínima dispersión diferencial**(MDD)**. Es un problema de optimización combinatoria consistente en seleccionar un subconjunto M de m elementos (|M|=m) de un conjunto inicial N de n elementos (con n>m) de forma que se minimice la dispersión entre los elementos escogidos.

Este problema tiene diferentes **aplicaciones en el campo de la optimización**, como pueden ser la elección de la localización de elementos públicos, selección de grupos homogéneos, identificación de redes densas, reparto equitativo, problemas de flujo, etc

$$\text{Minimize} \quad Max_{i\in M}\{\sum_{j\in M} d_{ij}\} - Min_{i\in M}\{\sum_{j\in M} d_{ij}\}$$
$$\text{Subject to} \quad M \subset N, |M| = m$$

donde:

- M es una solución al problema que consiste en un vector binario que indica los m elementos seleccionados
- d_{ij} es la distancia existente entre los elementos i y j.

Para resolver este problema se utilizarán 50 casos seleccionados con distancias reales con, n entre 25,50,75,100,125,150, y m enttre 2 y 45.

La Dispersión de una Solución es la diferencia de los valores extremos, es decir, la diferencia de la sumas de las distancias de dichos puntos al resto de los puntos. Por ejemplo, si tenemos 8 puntos para colocar farmacias, y solo podemos colocar 4, **¿cuál es la forma de colocarlas, de forma que se reduzca la dispersión?**

Los datos se encuentran en unos ficheros *.txt*, donde hay una primera línea que indica el numero de elementos n y el número de elementos a seleccionar m del problema.
Luego se encuentran ***n*(n-1)/2*** líneas con el formato i,j,d_{ij} que tienen el contenido de las **distancias entre los elementos**.
En mi caso, para los dos algoritmos he leido estos ficheros y he almacenado los datos en una matriz distancias completa, donde la diagonal es 0, y las triangulares superiores e inferiores son simétricas entre sí.

La posición (2,3) de la matriz distancias es la distancia entre los elementos 2 y 3, que a su vez es la misma que la posición (3,2).

La **representación de la solución** es un vector de enteros, donde la posición i-ésima es el numero del elemento que esta seleccionado. Mantengo en el conjunto de datos solución en la implementación el vector binario usado en la practica anterior, para la reutilizacion de código.

Para la **factorización de la función objetivo**, a la hora de generar una nueva solución no es necesario volver a calcular por completo el vector de distancias para obtener la nueva dispersión. Basta con restar la distancia a cada elemento de la solución al elemento que se ha quitado de la solución actual, y sumarle la distancia del nuevo elemento a todas las demás de la solución.
Entonces,teniendo el vector de distancias actualizado, para saber la dispersión de dicho conjunto de elementos restamos la mayor distancia de dicho vector con la menor

Debido a que se requiere aleatoriedad en ambos algoritmos, ya que son probabilísticos, he usado un vector de semillas, donde en cada iteración que realiza cada algoritmo se genera una nueva semilla, y se utiliza para generar nuevas soluciónes. El valor estático de la semilla sirve para que cada vez que se ejecute el algoritmo, se obtengan las mismas Soluciónes.

También se pedía calcular el tiempo de ejecución de cada algoritmo, por lo que he usado objetos de la clase <**chrono**> para tener una alta precisión en los tiempos, y los muestro en **segundos**.

Al finalizar cada algoritmo calculo el tiempo demorado por dicho algoritmo y la dispersión de la mejor solución encontrada.

1.3. Exploración vs Explotacion

En BBO no hay algo como una operación de cruce; las soluciones se ajustan finamente gradualmente mientras el proceso continúa por la operación de migración. Esto da una ventaja al BBO sobre otras técnicas evolutivas. BBO tiene una gran habilidad de explotación en un problema de optimización global.

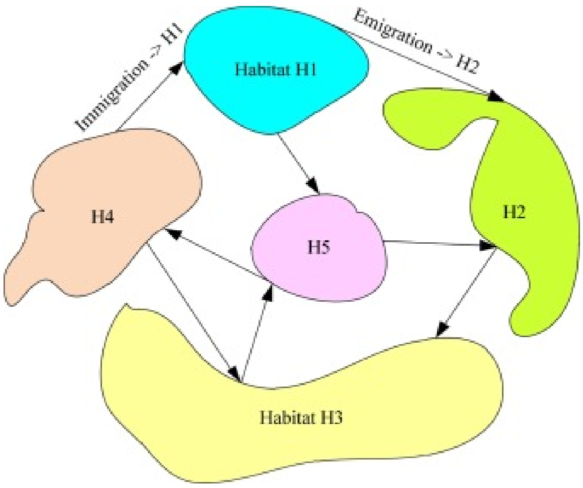


Figura 2: Islas

Para la explotación se van a utilizar las migraciones, que van a consistir en migrar elementos de las mejores soluciones a las peores. Esto se hace para que las soluciones buenas mejoren a las peores. Para la exploración se utilizan las mutaciones, que son las que se aplican a todas las soluciones en igual proporción. Habrá una variable que define con que probabilidad se produce una mutacion. Las mutaciones producen un cambio en la solucion, y se debe buscar un elemento nuevo que va a ser el que entre en la solucion por el anterior.

1.4. Equilibrio

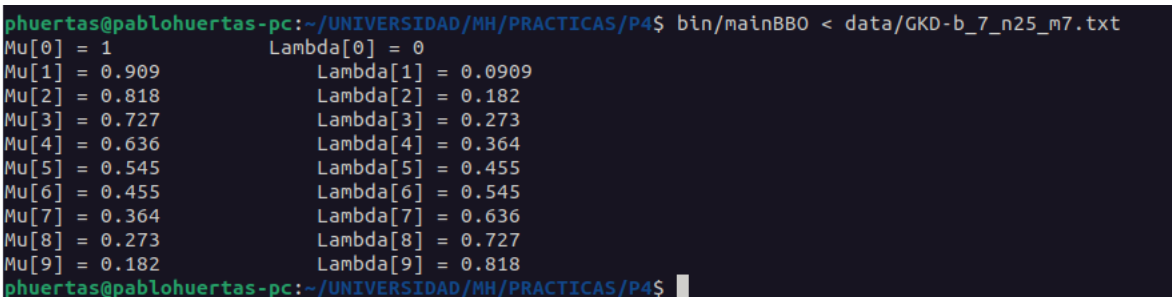
Este algoritmo presenta muy buen equilibrio entre la exploración y la explotación. Esto se debe a que casi con igual proporcion se van a realizar las migraciones y las mutaciones. Sin embargo, en el problema donde ha sido aplicado el algoritmo la migracion no funciona tan bien como en otros casos, y esto se debe a que para ser factible la solucion como hemos comentado antes debe tener un numero exacto de elementos seleccionados. Por lo tanto, cuando se realiza la migracion en una solucion, se debe eliminar un elemento de los seleccionados de dicha solucion y esto se hace de forma aleatoria. Se pueden perder elementos de las soluciones que sean prometedores por la aleatoriedad del algoritmo a la hora del reemplazamiento del elemento de la solucion.

En este problema en especifico encontramos una mejor explotacion que exploracion, ya que la exploracion realmente no cambia mucho la solucion, al tener que encontrar otro elemento que entre en esta para guardar la factibilidad. En cambio en la explotacion a pesar de que podamos perder un elemento bueno de la solucion, entrará un elemento que probabilisticamente debe ser prometedor tambien.

2. ADAPTACION DEL BBO AL PROBLEMA DE LA MINIMA DISPERSION DIFERENCIAL

Para adaptar este algoritmo al problema de la minima dispersion diferencial he tenido que hacer ciertos cambios.

Primero el tamanio de la poblacion(numero de islas), viene dado por el tamaño de la poblacion aleatoria generada.
Las variables de μ y λ vienes dadas en un vector del mismo tamaño de la poblacion.
Por ejemplo para una poblacion de tamaño 10 estos serian los valores de μ y λ:



```
phuertas@pablohuertas-pc:~/UNIVERSIDAD/MH/PRACTICAS/P4$ bin/mainBBO < data/GKD-b_7_n25_m7.txt
Mu[0] = 1             Lambda[0] = 0
Mu[1] = 0.909             Lambda[1] = 0.0909
Mu[2] = 0.818             Lambda[2] = 0.182
Mu[3] = 0.727             Lambda[3] = 0.273
Mu[4] = 0.636             Lambda[4] = 0.364
Mu[5] = 0.545             Lambda[5] = 0.455
Mu[6] = 0.455             Lambda[6] = 0.545
Mu[7] = 0.364             Lambda[7] = 0.636
Mu[8] = 0.273             Lambda[8] = 0.727
Mu[9] = 0.182             Lambda[9] = 0.818
phuertas@pablohuertas-pc:~/UNIVERSIDAD/MH/PRACTICAS/P4$
```

Figura 3: Valores de μ y λ para un tamaño de poblacion de 10

Ahora toca ordenar la poblacion por fitness, y esto se realiza ordenando de menor a mayor dispersion.
Las soluciones con menor dispersion estarán mas arriba en la poblacion. Asi se cumple la relacion 1 a 1 entre el vector de μ y λ y la poblacion.
Es por esto, que la primera solucion será la mejor, y coincide con el máximo valor de μ que es 1.
Es interesante que esta solucion al ser la mejor comparta sus caracteristicas a las peores, que tendran un valor de λ elevado.

Se genera un conjunto de elites, de tamaño elegido arbitrariamente, por ejemplo 0.1*tamaño de la poblacion si la poblacion es minimo de 10 soluciones, o mayor sera este valor si la poblacion es mas pequeña para asegurarnos que hay al menos algun elite que se guarda.

Ademas vamos a ir almacenando siempre la mejor solucion encontrada. Cada vez que se ordena la poblacion por fitness, se comprueba si el primer elemento tiene menor dispersion que la mejor encontrada hasta el momento y si es asi se sustituye. Esta solucion sera la solucion final.

Luego se entraria en el bucle donde se generan cada una de las generaciones del algoritmo, con dos bucles internos correspondientes anidados, uno de 0 hasta el tamaño de la poblacion y otro de 0 hasta el numero de elementos que han de ser seleccionados en el caso del problema en concreto.

En el bucle más interno es donde se realiza el proceso de migracion, mientras que en el bucle intermedio es donde se va a realizar la mutacion con una probabilidad tambien elegida arbitrariamente.

El proceso de mutacion se podria meter dentro del bucle más interno, pero habria que disminuir la probabilidad de mutacion ya que se ejecutarían muchas más mutaciones de las que deseamos.
De todas formas esto es un parametro muy arbitrario, ya que el valor que asignemos a la probabilidad de mutacion va a depender mucho de si queremos una mayor explotacion o menor.

Cada vez que se migra o muta un elemento, se calcula la funcion objetivo de la solucion modificada.

Algorithm 1: Algoritmo de Optimizacion Basada de Biogeografia

Input: $n, m, poblacion, prob_{mutacion}, n_{generaciones}, n_{elites}$
Output: *solucion*

$poblacion \leftarrow$ *(ORDENAR POR FITNESS)*
$mu \leftarrow \{\mathbf{1}, \ldots, \mathbf{0}\}$
$lambda \leftarrow \{\mathbf{0}, \ldots, \mathbf{1}\}$
$mejor_{solucion} \leftarrow \mathbf{poblacion[0]}$
$VectorDistancias \leftarrow$ *GenerarVectorDistancias()*
$DispersionComparacion \leftarrow$ *Calculardispersion(VectorDistancias)*
$Mejora \leftarrow \mathbf{TRUE}$
for $n_{iteraciones} \in n_{generaciones}$ **do**
    $elites \leftarrow \{\mathbf{poblacion[0]}, \ldots, \mathbf{poblacion[n_{elites}]}\}$
    for $i \in Size(poblacion)$ **do**
        for $j \in m$ **do**
            if $Random.Get(0,1) \leq \mathbf{lambda[i]}$ **then**
                $\mathbf{poblacion[i]} \leftarrow$ *MIGRACION*
                $\mathbf{poblacion[i]} \leftarrow$ *RECALCULAR FUNCION OBJETIVO*
        if $Random.Get(0,1) \leq \mathbf{prob_{mutacion}}$ **then**
            $\mathbf{poblacion[i]} \leftarrow$ *MUTACION*
            $\mathbf{poblacion[i]} \leftarrow$ *RECALCULAR FUNCION OBJETIVO*
    $poblacion \leftarrow$ *(ORDENAR POR FITNESS)*
    $poblacion \leftarrow$ *(SUSTITUIR PEORES POR ELITES)*
    if $mejor_{solucion}.dispersion < \mathbf{poblacion[0].dispersion}$ **then**
        $mejor_{solucion} \leftarrow \mathbf{poblacion[0]}$
$return\{mejor_{solucion}\}$

2.1. Descripción de la función objetivo

La función objetivo de este problema es la de encontrar la dispersión a partir de un vector de booleanos donde la posición i-ésima es 1 si el elemento i-ésimo está seleccionado, y 0 en caso contrario.

Para evaluar la función objetivo, se convierte internamente el vector de booleanos en una selección de elementos de números enteros.
Para ello, se recorre el vector de booleanos, y si la posición i-ésima es 1, se añade al final del vector de seleccionados el elemento i-ésimo.
Tenemos la matriz de distancias comentada anteriormente, y la selección de elementos, por lo que para evaluar la función objetivo, para cada elemento del vector de seleccionado, en la posición i-ésima del vector distancias, añadimos la distancia del elemento i-ésimo a todos los demás elementos del vector de seleccionados.
Las posiciones se corresponden 1 a 1 en los vectores de seleccionados y distancias.

Algorithm 2: Algoritmo de Evaluación de la Función Objetivo

Input: distancias(vector), seleccionados(vector), m(matriz distancias)

$distancias \leftarrow 0$

$VectorDistancias \leftarrow GenerarVectorDistancias()$
$DispersionComparacion \leftarrow Calculardispersion(VectorDistancias)$

$Mejora \leftarrow$ **TRUE**
for $i \in Size(seleccionados)$ **do**
  $acomparar \leftarrow i$
  for $j \in Size(seleccionados)$ **do**
    if $seleccionados[i] \neq acomparar$ **then**
      $distancias[i] + = m[acomparar][seleccionados[i]]$

2.2. Descripción de los operadores comunes

Hay ciertos operadores y funciones que son comunes para los algoritmos desarrollados en esta práctica, ya que por ejemplo la generación de soluciónes aleatorias es común y varios operadores más, por lo que voy a desglosar uno a uno para entrar más en profundidad.

2.2.1. Generación de soluciones aleatorias

Para la generación de la primera solución aleatoria, utilizo una funcion para generar soluciónes aleatorias, donde el numero de posibles elementos a escoger es emphn y el numero que finalmente son seleccionados son emphm.

Algorithm 3: Algoritmo de Generación de soluciónes Aleatorias

Input: n(número de puntos) m(número de puntos a seleccionar), semilla(número que simboliza una semilla estática)
Output: solución(vector de booleanos)

$solucin \leftarrow \emptyset$
$seleccionados \leftarrow \emptyset$

while $Size(seleccionados) < m$ **do**
  $seleccionados \leftarrow$ *Numero aleatorio que no esta en seleccionados*
  $solucin \leftarrow seleccionados.back$

3. PROPUESTA DE MEJORA DE LA METAHEURISTICA

Como propuesta de mejora del algoritmo BBO, he decidido usar un operador de reparacion de las soluciones, ya que como hablamos antes hay soluciones que al migrar o al mutar pueden perder la factibilidad. Este operador de reparacion tiene como objetivo no volver a hacer factible la solucion de forma aleatoria tal y como estabamos haciendo, sino que tener en cuenta al eliminar el o los elementos que sobran o introducir los que faltan teniendo en cuenta cuanto se aleja su fitness(dispersion) a la media de las dispersiones de la solucion actual.
Es decir, el eliminar elementos de la solucion se eliminaran los que tengan un fitness mas alejado de la media de fitness de dicha solucion, e igual al introducir un elemento, que se tendra en cuenta con que valor queda la media de fitness con ese elemento en la solucion. Por lo tanto para introducir un elemento se introducira el que menos aumente la dispersion (o la reduzca), mientras que al eliminar un elemento bastará con eliminar el que tenga el fitness mas alejado de la media de fitness de la solucion(ya sea por encima o por debajo).

3.1. Hibridacion con Enfriamiento Simulado

Una tecnica de mejora del algoritmo es la Hibridacion con Enfriamiento Simulado.
Este algoritmo se lo aplico a las soluciones que pueden estar mas estancadas en un optimo local, ya que sabemos que el algoritmo de Enfriamiento simulado tiene buenas tecnicas de escape de estos optimos. Por lo tanto lo aplico por defecto a todas las soluciones cada 5 generaciones para que haya una exploracion mas pronunciada.

A continuacion se muestra una descripcion del algoritmo de Enfriamiento Simulado

3.1.1. Algoritmo de Enfriamiento Simulado

El algoritmo de Enfriamiento Simulado *(Simulated Annealing)* es un algoritmo de búsqueda metaheuristica para problemas de optimización global.
El objetivo de este algoritmo es encontrar una solución optima o casi optima de un problema en un espacio de búsqueda grande.
Tiene un criterio probabilistco de aceptacion de soluciónes basado en Termodinamica.
La forma que tiene de escapar de óptimos locales, es la posibilidad de aceptar soluciónes peores con una cierta probabilidad, la cual va disminuyendo conforme se va avanzando en el algoritmo hacia una buena solución.
Tiene una filosofía de diversificar al principio e intensificar al final, es decir, al principio del algoritmo se evaluan multiples soluciónes distintas y se selecciona la mejor, y al final se intensifica la búsqueda explotandola.

El máximo de éxitos que se podrán generar como máximo en cada iteracion del algoritmo serán n.
El máximo de vecinos que se podrán generar como máximo en cada iteracion del algoritmo serán 10*n.
La constante μ tendra valor 0.3 en toda la ejecucion

La temperatura inicial se calculará como...

$$To = \frac{\mu * C(So)}{-\ln(\varphi)} \tag{1}$$

siendo $\varphi = \mu$

Beta se calculará de la forma...

$$\beta = \frac{t_i - t_f}{t_i * t_f * n} \tag{2}$$

siendo t_i la temperatura inicial, t_f la temperatura final y n el tamaño de la solución.

Al final de cada iteracion se calcula la temperatura que se va a tomar como nueva, y esta se calcula de la forma...

$$t_{k1} = \frac{t_k}{1 + (\beta * t_k)} \tag{3}$$

siendo t_k la temperatura actual y k el numero de iteracion.

El número máximo de evaluaciones de la funcion objetivo del algoritmo completo serán **100000**

Algorithm 4: Algoritmo de Enfriamiento Simulado

Input: TemperaturaInicial(temperatura inicial), TemperaturaFinal(temperatura final)
Output: TemperaturaActual(temperatura actual)

$TemperaturaInicial \leftarrow CalcularTemperaturaInicial(\mu, \varphi, C(S_o))$
$TemperaturaActual \leftarrow TemperaturaInicial$
$TemperaturaFinal \leftarrow 10^{-3}$
$nEnfriamientos \leftarrow 1000/n$
$maxVecinos \leftarrow 10 * n$
$maxExitos \leftarrow n$

$iteraciones \leftarrow 0$
$evaluaciones \leftarrow 0$
$s_o \leftarrow CalcularsoluciónAleatoria$
$s_{mejor} \leftarrow S_o$

while $(TemperaturaActual > TemperaturaFinal)\&\&(iteraciones < nEnfriamientos)\&\&(nEvaluaciones < 100000)$ **do**
    $exitos \leftarrow 0$
    for $i \in 1 \dots maxVecinos$ && $exitos < maxExitos$ **do**
        $S_{Vecino} \leftarrow GenerarsolucinVecinaAleatoria()$
        $evaluaciones \leftarrow evaluaciones + 1$

        $\Delta_{dispersion} \leftarrow dispersion(S_{vecino}) - dispersion(S_{actual})$
        Calculamos la probabilidad de que se acepte la nueva si es peor que la actual
        $probabilidad \leftarrow e^{-\Delta_{dispersion}/TemperaturaActual}$

        if $(\Delta_{dispersion} < 0)$ *or* $(GetRandomNumber(0,1) < probabilidad)$ **then**
            $S_{actual} \leftarrow S_{vecino}$
            $exitos \leftarrow exitos + 1$

        if $dispersion(S_{actual}) - dispersion(S_{mejor})$ **then**
            $S_{mejor} \leftarrow S_{actual}$

    $beta \leftarrow CalcularBeta$
    $TemperaturaActual \leftarrow CalcularTemperaturaActual(t_k, \beta, t_i, n)$
    $iteraciones \leftarrow iteraciones + 1$

return S_{mejor}

3.1.2. Operador de mutacion

La mutacion consiste en modificar con cierta probabilidad uno o varios genes de la poblacion (en este caso de una solución) aleatoriamente. La probabilidad de mutacion es dada por la constante *probabilidad 0.1.* Cuando muta un gen de un cromosoma, tenemos que encontrar otro gen del mismo cromosoma con el valor contrario, para mantener la factibilidad de la solución de dicho cromosoma. Por ejemplo, en una solución con 10 elementos donde se seleccionan 3, si se va a mutar el segundo seleccionado, tenemos que buscar uno de los 7 elementos que no esten seleccionados de manera aleatoria y cambiar el valor de cada gen. El rango de elementos que pueden ser mutados, van desde 0 hasta el producto del numero de cromosomas por el numero de genes por cromosoma.
Si la poblacion tiene 10 cromosomas, y cada cromosoma 5 genes, si se genera para mutar el elemento 15, será el sexto gen del segundo cromosoma.

Algorithm 5: Operador de Mutación

Input: p(poblacion), prob(probabilidad)
Output: pnueva(poblacion generada)

```
rango mutacion ← p.NumeroDeCromosomas() · p.NumeroDeGenesPorCromosoma()
for i ∈ Size(p) do
    if GenerarNumeroAleatorioEntre(0, 1) < prob then
        Genero aleatoriamente un elemento en el rango de mutacion
        posicion ← GenerarNumeroAleatorioEntre(0, rango)
        Para el elemento de la posicion generada, busco otro gen del mismo cromosoma con el valor contrario
        posicion2 ← Gen del mismo cromosoma aleatorio con valor contrario
        Swap(posicion, posicion2)
return pnueva
```

En la ejecucion del algoritmo se realizan 10 iteraciones y cada BL como máximo hara **10000** evaluaciones o no mejore la solución en todo el entorno. El valor usado para el numero de genes a mutar de la solución es $t = 0.3$

3.2. Hibridacion con Busqueda Local

Anteriormente hemos visto una posible hibridacion con el algoritmo de Enfriamiento Simulado.
Ahora vamos a ver una hibridacion con la busqueda local. Sabemos que el algoritmo de Busqueda Local tiene una explotacion muy pronunciada, por lo que si lo combinamos con un algoritmo con buena exploracion como es el BBO en nuestro caso (que hemos visto que para este problema la explotacion no es su punto fuerte por la propia estructura de la funcion objetivo) podemos obtener muy buenos resultados y vamos a experimentar sobre ello.

A continuacion se muestra una descripcion del algoritmo de busqueda local.

3.2.1. Busqueda Local

Este algoritmo es un tipo de algoritmos de busqueda por trayectorias simples. En este algoritmo, se empieza con una solución inicial completa y aleatoria, es decir, una Solución con M elementos que no se repiten entre sí. El orden de estos elementos no es relevante.

La idea es tras haber generado una completa Solución aleatoria válida, generar el **vecindario completo** de la Solución actual, **desordenarlo aleatoriamente**, y recorrerlo comparando en cada iteracion si se mejora la Dispersión.
Si se mejora la Dispersión, se **selecciona dicha Solución como Solución actual** y se vuelve a generar el vecindario. Este proceso se hace hasta que no se mejore la Dispersión con todo el vecindario generado o hasta que se hayan hecho **100000 evaluaciones de la funcion objetivo**. Es decir, comprobar 100000 veces si se mejora la Dispersión.
Como vemos este algoritmo se parece a Greedy en que ambos cuando encuentran una Solución mejor que la anterior la seleccionan, y no se espera en este caso a recorrer todo el vecindario para encontrar una mejor Solución. Es por eso que este algoritmo se llama Busqueda Local de **Primero el mejor**

La generación de la primera Solución aleatoria se hace con un bucle que va generando numeros aleatorios entre 0 y n-1, de forma que si no se ha añadido aún a la Solución, lo añade. Este proceso se repite hasta que el numero de elementos de la Solución sea igual a M
Para la generación de vecinos, uso un vector de tuplas, que contienen el elemento que se va a intercambiar y el elemento que se va a intercambiar y va a entrar a la Solución provisional.
Por ejemplo, si tengo M=6 y N=3, Solución provisional=(1,3,5), y genero el vecindario de esta Solución, este será el vector de tuplas
(1,0), (1,2) ,(1,4), (3,0), (3,2), (3,4), (5,0), (5,2), (5,4).
Entonces, desordena este vector aleatoriamente y se va intercambiando la posicion primera de la tupla que se encuentra en la Solución por la segunda posicion de la tupla que no se encuentra en la Solución
La factorización es la misma que en el algoritmo greedy, cuando se intercambia un elemento de la Solución por otro, en el vector distancias a cada elemento se le resta la distancia con el elemento que se elimina, y se le suma la distancia con el elemento

que se añade, ademas de añadir en la posicion del elemento añadido la distancia con todos los demas de la Solución.

PSEUDOCÓDIGO DEL ALGORITMO DE BUSQUEDA LOCAL

Algorithm 6: Algoritmo de búsqueda local

```
v ← 0,w ← 0
S ← D
T ← ∅
solucin ← ∅
Elementosrestantes ← V
DispersionComparacion ← ∅
Distancias ← ∅
Dispersion ← ∅

Copiasolucin ← ∅
CopiaDistancia ← ∅
Vecindario ← ∅

while solucin < M do
    Vamos generando elementos aleatorios y los introducimos a la solución
    Elementoaintroducir ← GenerarElementoAleatorio(Elementos restantes)
    Elementosrestantes ← Elementosrestantes − Elementoaintroducir
    solucin ← solucin ∪ Elementoaintroducir
Ya tenemos una solución completa y válida de tamaño M
El conjunto de elementos restantes solo contiene
los elementos que no están en la solución

VectorDistancias ← GenerarVectorDistancias()
DispersionComparacion ← Calculardispersion(VectorDistancias)

Mejora ← TRUE
while Mejora == TRUE && iteraciones ≤ 100000 do
    Generamos un vecindario completo de la solución actual
     y lo mezclamos aleatoriamente
    Vecindario ← GenerarVecindario(solución)
    Vecindario ← Desordenar(Vecindario)

    Actualizamos las variables antes de recorrer el vecindario
    Copiasolucin ← solucin
    Mejora ← FALSE
    dispersioncomparacion ← Dispersion

    for i ∈ Size(Vecindario)&&mejora == FALSE do
        Recorremos el vecindario
        Copiasolucin ← SustituirPunto(vecindario[i])
        CopiaDistancias ← GenerarVectorDistancias(Copiasolución)
        dispersioncomparacion ← Calculardispersion(CopiaDistancias)
    if dispersion comparacion <dispersion then
        Si la dispersion es mejor, actualizamos la solución
        dispersion ← dispersion comparacion
        solucin ← Copiasolución
        Mejora ← TRUE
        VectorDistancias ← CopiaDistancias
        Restantes ← CalcularRestantes(solución)
    else
        Si la dispersion no es mejor, no actualizamos la solución,
        y volvemos al estado anterior
        Copiasolucin ← solucin
        CopiDistancias ← VectorDistancias
    Iteraciones ← Iteraciones + 1
Devolvemos la solución
Return solución
```

4. ESTUDIO EXPERIMENTAL Y ANÁLISIS DE RESULTADOS

En ambos algoritmos hemos usado el mismo vector de semillas, que en cada iteración que ejecuta el programa el algoritmo, se coge la posicion i-esima del vector de semillas.
El vector semillas es (1,2,3,4,5) Por lo tanto en la primera iteracion se define la semilla como Random::Seed(1), y asi sucesivamente.
Para comparar los resultados entre los dos algoritmos implementados en esta práctica, he hecho una tabla donde se muestran, para cada algoritmo, el tiempo medio y la dispersion media conseguida entre las 5 iteraciones conseguido con cada uno de los ficheros de datos.

Algoritmo Greedy			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo(s)**
GKD-b_1_n25_m2	0,0000	0,00	0,0000000256
GKD-b_2_n25_m2	0,0000	0,00	0,0000000272
GKD-b_3_n25_m2	0,0000	0,00	0,0000000288
GKD-b_4_n25_m2	0,0000	0,00	0,0000000326
GKD-b_5_n25_m2	0,0000	0,00	0,0000000230
GKD-b_6_n25_m7	59,0224	78,45	0,0001462840
GKD-b_7_n25_m7	73,6978	80,87	0,0001497410
GKD-b_8_n25_m7	90,3868	81,46	0,0001523710
GKD-b_9_n25_m7	91,6874	81,38	0,0001492890
GKD-b_10_n25_m7	85,5476	72,80	0,0001473730
GKD-b_11_n50_m5	41,3777	95,35	0,0001821140
GKD-b_12_n50_m5	41,7288	94,92	0,0001892350
GKD-b_13_n50_m5	30,3683	92,22	0,0001811580
GKD-b_14_n50_m5	35,8852	95,37	0,0001831770
GKD-b_15_n50_m5	55,7863	94,89	0,0002085510
GKD-b_16_n50_m15	249,9140	82,90	0,0012096900
GKD-b_17_n50_m15	206,6920	76,72	0,0012426700
GKD-b_18_n50_m15	157,5670	72,59	0,0012195900
GKD-b_19_n50_m15	180,7680	74,32	0,0013962000
GKD-b_20_n50_m15	176,8530	73,02	0,0012447400
GKD-b_21_n100_m10	87,8713	84,26	0,0012373300
GKD-b_22_n100_m10	88,5641	84,57	0,0012903300
GKD-b_23_n100_m10	99,2332	84,54	0,0012189000
GKD-b_24_n100_m10	71,3271	87,89	0,0012417800
GKD-b_25_n100_m10	82,8395	79,24	0,0012700300
GKD-b_26_n100_m30	506,5690	66,69	0,0111152000
GKD-b_27_n100_m30	471,1350	73,02	0,0109178000
GKD-b_28_n100_m30	481,1330	77,89	0,0109190000
GKD-b_29_n100_m30	341,3560	59,73	0,0109695000
GKD-b_30_n100_m30	443,3280	71,24	0,0109686000
GKD-b_31_n125_m12	117,0180	89,96	0,0021813600
GKD-b_32_n125_m12	73,7461	74,52	0,0022404800
GKD-b_33_n125_m12	138,7920	86,65	0,0022305500
GKD-b_34_n125_m12	99,4028	80,39	0,0021615800
GKD-b_35_n125_m12	97,1065	81,35	0,0022106500
GKD-b_36_n125_m37	471,2490	67,02	0,0230018000
GKD-b_37_n125_m37	502,6070	60,43	0,0227816000
GKD-b_38_n125_m37	536,0130	64,93	0,0229423000
GKD-b_39_n125_m37	510,3990	66,97	0,0224148000
GKD-b_40_n125_m37	427,3790	427,38	0,0236678000
GKD-b_41_n150_m15	120,1290	80,57	0,0040878100
GKD-b_42_n150_m15	154,6890	82,68	0,0041689400
GKD-b_43_n150_m15	167,8920	84,06	0,0041644700
GKD-b_44_n150_m15	145,4070	82,16	0,0044095400
GKD-b_45_n150_m15	124,5110	77,69	0,0042332100
GKD-b_46_n150_m45	637,7070	64,29	0,0437122000
GKD-b_47_n150_m45	524,0290	56,38	0,0440777000
GKD-b_48_n150_m45	483,4690	53,10	0,0463861000
GKD-b_49_n150_m45	653,2360	65,34	0,0437011000
GKD-b_50_n150_m45	727,2570	65,78	0,0441537000

(a) Tabla de resultados de Greedy

Algoritmo BL			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo(s)**
GKD-b_1_n25_m2	0,0000	0,00	0,0000248110
GKD-b_2_n25_m2	0,0000	0,00	0,0000252718
GKD-b_3_n25_m2	0,0000	0,00	0,0000245076
GKD-b_4_n25_m2	0,0000	0,00	0,0000246664
GKD-b_5_n25_m2	0,0000	0,00	0,0000332290
GKD-b_6_n25_m7	39,2743	67,62	0,0002810230
GKD-b_7_n25_m7	31,4916	55,23	0,0002799990
GKD-b_8_n25_m7	31,9922	47,61	0,0003393000
GKD-b_9_n25_m7	28,6988	40,52	0,0004346210
GKD-b_10_n25_m7	37,8571	38,54	0,0003469450
GKD-b_11_n50_m5	15,8881	87,88	0,0003353120
GKD-b_12_n50_m5	12,7408	83,35	0,0005160320
GKD-b_13_n50_m5	15,3791	84,64	0,0003750850
GKD-b_14_n50_m5	18,6001	91,06	0,0003390440
GKD-b_15_n50_m5	14,0532	79,70	0,0003117420
GKD-b_16_n50_m15	119,7650	64,31	0,0029540300
GKD-b_17_n50_m15	90,2033	46,67	0,0034896000
GKD-b_18_n50_m15	91,2825	52,68	0,0027151700
GKD-b_19_n50_m15	123,8630	62,53	0,0031808200
GKD-b_20_n50_m15	95,9887	50,29	0,0022829500
GKD-b_21_n100_m10	34,4393	59,84	0,0032439300
GKD-b_22_n100_m10	37,0628	63,13	0,0028765400
GKD-b_23_n100_m10	42,2406	63,67	0,0027116500
GKD-b_24_n100_m10	30,6327	71,79	0,0032216900
GKD-b_25_n100_m10	36,3188	52,64	0,0029993100
GKD-b_26_n100_m30	335,4900	49,71	0,0272325000
GKD-b_27_n100_m30	358,5420	64,55	0,0237913000
GKD-b_28_n100_m30	357,7700	70,27	0,0227821000
GKD-b_29_n100_m30	265,3810	48,21	0,0190169000
GKD-b_30_n100_m30	359,5180	64,54	0,0175005000
GKD-b_31_n125_m12	39,5049	70,27	0,0084009500
GKD-b_32_n125_m12	43,2581	56,57	0,0051494000
GKD-b_33_n125_m12	59,6755	68,95	0,0056468300
GKD-b_34_n125_m12	42,2846	53,91	0,0054818700
GKD-b_35_n125_m12	48,9118	62,97	0,0070433400
GKD-b_36_n125_m37	333,7270	53,42	0,0437169000
GKD-b_37_n125_m37	427,0190	53,42	0,0546628000
GKD-b_38_n125_m37	544,1900	65,46	0,0363438000
GKD-b_39_n125_m37	361,1280	53,32	0,0474091000
GKD-b_40_n125_m37	419,5800	57,53	0,0394901000
GKD-b_41_n150_m15	76,9359	69,66	0,0087534800
GKD-b_42_n150_m15	76,6500	65,05	0,0104099000
GKD-b_43_n150_m15	82,1052	67,41	0,0133788000
GKD-b_44_n150_m15	69,5018	62,68	0,0115883000
GKD-b_45_n150_m15	69,9745	60,31	0,0110468000
GKD-b_46_n150_m45	543,2090	58,07	0,0687209000
GKD-b_47_n150_m45	431,2790	46,99	0,0946293000
GKD-b_48_n150_m45	501,3430	54,77	0,0764270000
GKD-b_49_n150_m45	537,4390	57,87	0,0835859000
GKD-b_50_n150_m45	563,4460	55,83	0,0816988000

(b) Tabla de resultados de BL

Figura 4: Tablas de resultados de Greedy y BL

Media Desv:	**76,5595103744**
Media Tiempo:	**0,008761569604**

(a) Desviacion y tiempo de Greedy

Media Desv:	**55,10878940233**
Media Tiempo:	**0,017145496976**

(b) Desviacion y tiempo de BL

Figura 5: Desviaciones y tiempos de Greedy y BL

Observando los datos de las tablas, podemos observar que el algoritmo greedy tiene un tiempo menor que busqueda local, mientras que tiene una mayor desviacion, lo que quiere decir que sus resultados de dispersiones son peores.

Aunque estas diferencias no sean muy significativas, a la hora de evaluar muchas ejecuciones de estos algoritmos, encontramos como se acentúa más la diferencia.

$$\textbf{Desviacion} = 100 * \sum_{i=1}^{n} \frac{ValorAlgoritmo_i - MejorValor_i}{ValorAlgoritmo_i} \tag{4}$$

Por lo tanto, tenemos unos datos de referencia, que contienen el mejor coste obtenido para cada instancia del problema. El algoritmo de Busqueda Local obtiene mejores dispersiones de media que Greedy, y esto es gracias a que este algoritmo tiene mas probabilidad de encontrar mejores soluciónes.
Al generar el vecindario completo se asegura que si no se encuentran mejores dispersiones, no las selecciona, al contrario que Greedy, que aunque ninguno mejore la dispersion añade a la solución el que menos la empeore.
Esto evita que el algoritmo de Busqueda Local vaya hacia soluciónes peores(mínimos locales), y siempre se asegure que cuando actualiza la solución es para una mejor dispersion.

En cambio, Greedy acepta soluciónes peores a la actual, y esto puede hacer que caiga en mínimos locales, y al siempre añadir elementos a la solución, no poder salir de ellos.

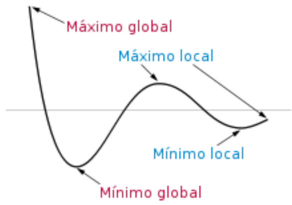


Figura 6: Gráfica que muestra el comportamiento de una búsqueda de una solución

4.1. Tabla resumen

Algoritmo	**Desviación media**	**Tiempo (en segundos)**
Greedy	76,5595103744	0,008761569604
BL	55,10878940233	0,017145496976
AGG-Uniforme	40,0088991109	7,463298200000
AGG-Posición	45,4862646141	3,312911800000
AGE-Uniforme	55,9744088626	9,524131200000
AGE-Posición	54,5438597140	5,062808200000
ES	39,6364925123	0,019224743220
BMB	35,4212565508	0,239388720600
ILS	38,7884380920	0,100818801540
ILS_ES	39,6364925123	0,019224843220
BBO	42,0759918345	0,455122914000
BBO-BL	20,5689749201	0,324748746000
BBO-BL+	10,3153847989	8,397473508000

Tabla 1: Tabla de medias de desviaciones y tiempos de los algoritmos

4.2. Analisis resultados BBO

A continuacion voy a analizar los resultados de BBO, con todas las variantes realizadas en el

4.2.1. BBO sin hibridacion

En la primera prueba del algoritmo hemos visto que nos da dispersion 42, con un tiempo bastante bueno y con los siguientes parametros:

- Tamaño de poblacion : 100
- Numero de iteraciones : 200
- Probabildad de mutacion : 0.1
- Numero de elites : 20

Observamos que tenemos parametros bastante generosos, lo que va a permitir bastante la exploracion, ya que como hemos comentado anteriormente el BBO en nuestro problema de optimizacion, no tiene una gran calidad de explotacion debido a la propia naturaleza de la funcion objetivo, por lo que hemos decidido probar a explorar de una forma mas exhaustiva, es decir, con una poblacion de tamaño mayor y mayor probabilidad de mutacion.

4.2.2. BBO con hibridacion BL

En esta hibridacion con la Busqueda Local hemos usado los siguientes parametros:

- Tamaño de poblacion : 50
- Numero de iteraciones : 100
- Probabildad de mutacion : 0.01
- Numero de elites : 5

Observamos una muy buena desviacion obtenida de media y un mejor tiempo que en el caso anterior, este se debe a que hemos disminuido bastante el numero de iteraciones(numero de generaciones maximo que se van a generar) y el tamaño de la poblacion.
Aprovechamos mejor la alta exploracion que presenta el BBO, para posteriormente aplicar una BL sobre la poblacion final.
Solo aplicamos dos BL, antes de empezar el BBO y al finalizar.

4.2.3. BBO con hibridacion BL++

En esta hibridacion con la Busqueda Local hemos usado los siguientes parametros:

- Tamaño de poblacion : 100

- Numero de iteraciones : 50
- Probabildad de mutacion : 0.1
- Numero de elites : 5

Observamos la mejor dispersion media hasta el momento y esto se debe a lo que comentamos anteriormente, de que aprovechamos el BBO para una exploracion exhaustiva y aqui aplicamos BBO a todos los elementos de la poblacion en cada iteracion. Es por esto que obtenemos los mejores resultados por el momento.

4.2.4. BBO con hibridacion ES

En esta hibridacion con el Enfriamiento Simulado hemos usado los siguientes parametros:

- Tamaño de poblacion : 100
- Numero de iteraciones : 50
- Probabildad de mutacion : 0.1
- Numero de elites : 5

Observamos que la hibridacion con Enfriamiento Simulado no ofrece malos resultados, pero bastante peores que las hibridaciones con BL. Esto se puede deber a que el algoritmo de Enfriamiento Simulado con 10000 evaluaciones de la funcion objetivo(que es el ES usado) no llega a explotar bien las soluciones, entones estamos hibridando un algoritmo que explora mucho como es el BBO, y otro que en nuestro caso tambien esta explorando mucho, por lo que no se puede conseguir una buena explotacion final y es por esto que los resultados no son tan buenos.