[]

METAHEURISTICAS PRACTICA 1

Pablo Huertas Arroyo

16 de mayo de 2022

UNIVERSIDAD DE GRANADA

Correo: phuertas@correo.ugr.es
DNI:77033078Y
Grupo 3A, subgrupo 2
Horario: Lunes de 17:30 a 19:30

Índice

1. INTRODUCCION AL PROBLEMA ABORDADO

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?**

Esto es lo que realizamos en esta prácticas, donde probaremos diferentes algoritmos para resolver el problema, y los compararemos entre ellos para poder extraer nuestras propias conclusiones.

2. ESQUEMA COMÚN

2.1. Representación de soluciones

Para esta práctica, he implementado 7 algoritmos distintos.

- Algoritmo 1: **Algoritmo Genético Generacional con cruce Uniforme**
- Algoritmo 2: **Algoritmo Genético Generacional con cruce de Posición**
- Algoritmo 3: **Algoritmo Genético Estacionario con cruce Uniforme**
- Algoritmo 4: **Algoritmo Genético Estacionario con cruce de Posición**
- Algoritmo 5: **Algoritmo Memético con 3 variantes diferentes**
  - Algoritmo 5.1: **AM con BL en todos los cromosomas**
  - Algoritmo 5.2: **AM con BL en los N*0'1 mejores cromosomas**
  - Algoritmo 5.3: **AM con BL en N*0'1 aleatorios cromosomas**

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 binario, donde la posición i-ésima es 1 si el elemento i-ésimo está seleccionado, y 0 en caso contrario.

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 soluciones. 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 de la población.

2.2. 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 1: Algoritmo de Evaluación de la Función Objetivo

```
Input: distancias(vector), seleccionados(vector), m(matriz distancias)
distancias ← 0

VectorDistancias ← GenerarVectorDistancias()
DispersionComparacion ← Calculardispersion(VectorDistancias)

Mejora ← TRUE
for i ∈ Size(seleccionados) do
    acomparar ← i
    for j ∈ Size(seleccionados) do
        if seleccionados[i] ≠ acomparar then
            distancias[i]+ = m[acomparar][seleccionados[i]]
```

2.3. 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 soluciones aleatorias para la población inicial es común y varios operadores más, por lo que voy a desglosar uno a uno para entrar más en profundidad.

2.3.1. Operador de generación de vecino

2.3.2. Generación de soluciones aleatorias

Para la generación de la primera población aleatoria, utilizo dos funciones distintas, una para generar soluciones aleatorias, y otra que genera una población completa de tamaño 50, donde 50 es el tamaño de la población que vamos a usar en todos los

algoritmos.
La segunda función es la que llama a la primera 50 veces.

Algorithm 2: Algoritmo de Generación de Soluciones Aleatorias

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

```
1  solucion ← ∅
2  seleccionados ← ∅
3  while Size(seleccionados) < m do
4  |  seleccionados ← Numero aleatorio que no esta en seleccionados
5  |  solucion ← seleccionados.back
```

Algorithm 3: Algoritmo de Generación de Poblaciones Aleatorias

Input: n(número de puntos) m(número de puntos a seleccionar), semilla(número que simboliza una semilla estática), matrizdatos(matriz distancias)
Output: p(poblacion generada)

```
1  p ← ∅
2  for i ∈ 50 do
3  |  solucion ← GeneraSolucionAleatoria(n,m)
4  |  seleccionados ← ∅
5  |  for j ∈ Size(solucion) do
6  |  |  if (solucion[j]! = 0) then
7  |  |  |  seleccionados ← j
8  |  Algorithm1 − − > EvaluaciondelaFuncionObjetivo
9  |  p ← solucion
```

2.3.3. Mecanismo de selección en los Algoritmos Genéticos

Para estos algoritmos, en el mecanismo de selección hemos usado el torneo binario, que consiste en elegir dos individuos aleatorios de la población y seleccionar el que mejor fitness, en este caso menor dispersión, de estos. En el esquema generacional, se aplican n torneos binarios, donde n es el tamaño de la población. En cambio en el esquema estacionario, se aplican solamente 2 torneos binarios, que darán lugar a los dos padres que posteriormente son cruzados.

Algorithm 4: Algoritmo de Selección

Input: p(poblacion)
Output: pnueva(poblacion generada)

```
1  pnueva ← ∅
2  if emphEsquemaGeneracional then
3  |  numero iteraciones ← n
4  else
5  |  numero iteraciones ← 2
6  for i ∈ numeroiteraciones do
7  |  posicion1 ← GenerarNumeroAleatorioEntre(0, n)
8  |  posicion2 ← GenerarNumeroAleatorioEntre(0, n)
9  |  Las posiciones no deben ser iguales
10 |  if dipersion(posicion1) < dipersion(posicion2) then
11 |  |  pnueva ← p(posicion1)
12 |  else
13 |  |  pnueva ← p(posicion2)
14 return pnueva
```

2.3.4. Operadores de cruce

Para estos algoritmos, he usado dos operadores de cruce distintos. En el primero a partir de dos padres se generan dos hijos, mientras que en el segundo se genera un solo hijo. Ambos tienen en común que las posiciones que contienen el mismo valor en ambos padres se mantienen en los dos hijos generados.

- Cruce de Posición
  Las posiciones (genes) restantes de ambos padres se desordenan aleatoriamente de forma independiente, y se completan los huecos en ambos hijos con dichos valores desordenados.
  Por lo tanto, esto da lugar a dos soluciones válidas (con m 1's) que en este caso son los hijos

Algorithm 5: Operador de Cruce Basado en Posición

```
Input: p1(padre 1), p2(padre2)
Output: h1(hijo1), h2(hijo2)

restantesp1 ← ∅
restantesp2 ← ∅
posiciones no asignadas ← ∅

for i ∈ n do
    if Disintos(p1[i], p2[i]) then
        Si el gen en la posicion i es distinto en ambos padres, no lo heredan
        restantesp1 ← p1[i]
        restantesp2 ← p2[i]
        posiciones no asignadas ← i
    else
        Si el gen en la posicion i es igual en ambos padres, lo heredan
        h1[i] ← p1[i]
        h2[i] ← p2[i]
restantesp1 ← Desordenar aleatoriamente
restantesp2 ← Desordenar aleatoriamente

for i ∈ Size(restantesp1) do
    h1[posiciones no asignadas[i]] ← restantesp1[i]
    h2[posiciones no asignadas[i]] ← restantesp2[i]

returnh1, h2
```

- Cruce Uniforme
  Las selecciones restantes se seleccionan aleatoriamente de un padre o de otro, lo que puede dar lugar a un hijo cuya solución no sea válida.
  Por lo tanto, necesitamos un operador de reparación, que lo que hace es dada una solución con x 1's, la convierte en una solución con m 1's, para que sea válida.

Algorithm 6: Operador de Cruce Basado en Posición

```
Input: p1(padre 1), p2(padre2)
Output: h(hijo)

posiciones no asignadas ← ∅
for i ∈ n do
    if Disintos(p1[i], p2[i]) then
        posiciones no asignadas ← i
        if GenerarNumeroAleatorioEntre(0, 1) == 0 then
            h[i] ← p1[i]
        else
            h[i] ← p2[i]
    else
        Si el gen en la posicion i es igual en ambos padres, lo heredan
        h[i] ← p1[i]
numero de escogidos ← ContarSeleccionadosEn(h)

if numero de escogidos! = m then
    h ← Reparar(h)
return h
```

Algorithm 7: Operador de Reparación

```
Input: h(hijo), v(pos a rellenar)
Output: h(ijo)

cantidad a reparar ← m − ContarSeleccionadosEn(h)
distancias ← GenerarDistancias(h)
mediadistancias ← Media(distancias)

if cantidad a reparar < 0 then
    Sobran elementos seleccionados
    while cantidad a reparar < 0 do
        Busco el elemento cuyas distancias se alejen mas de la media
        posicion ← 0
        distanciamayor ← 0
        for i ∈ Size(distancias) do
            if ABS(distancias[i] − mediadistancias) > distanciamayor then
                distanciamayor ← ABS(distancias[i] − mediadistancias)
                posicion ← i
        Elimino dicho elemento seleccionado de la solucion del hijo
        h[posicion] ← v
        cantidad a reparar ← cantidad a reparar + 1
else
    Faltan elementos seleccionados
    while cantidad a reparar > 0 do
        Busco el elemento que minimice la media de distancias
        posicion ← 0
        mediamin ← 0
        for i ∈ Size(h) do
            Pruebo uno a uno añadiendo los elementos no seleccionados a la solucion
            if mediaconnuevoelemento < mediamin then
                mediamin ← media
                posicion ← i
    Añado dicho elemento no seleccionado de la solucion del hijo
    h[posicion] ← v
    cantidad a reparar ← cantidad a reparar − 1
return h
```

2.3.5. Operador de mutación

La mutacion consiste en modificar con cierta probabilidad uno o varios genes de la poblacion 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 solucion de dicho cromosoma. Por ejemplo, en una solucion 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 8: Operador de Mutación

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

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

3. ESTRUCTURA DE MÉTODO DE BÚSQUEDA

3.1. Pseudocódigo del método de búsqueda

La estructura general del método de búsqueda es la siguiente:

- Primero de aplica el operador de selección, de donde la poblacion actual se selecciona los que van a formar la poblacion siguiente.
- Segundo, se aplica el operador de cruce, donde con cierto porcentaje dependiendo del esquema de cruce escogido, se cruzan padres con los operadores vistos anteriormente para dar lugar a los nuevos hijos que van a formar la poblacion siguiente.
- Tercero, se aplica el operador de mutación, donde con cierto porcentaje pueden mutar x genes, pero siempre las soluciones son factibles.
  Es decir, si tiene que haber m elementos seleccionados y se elige mutar el segundo elemento del cromosoma y, se ha de encontrar otro gen que tenga el valor contrario al mutado, para que cambie tambien su valor y la solucion siga siendo factible
- Como cuarto y ultimo, se aplica el operador de reemplazamiento, donde dependiendo del esquema que se haya usado, se realiza de una forma u otra. Es la última fase que va a dar lugar a la población completa que va a sustituir a la que había en el momento del inicio de la seleccion.
  En el esquema generacional, solo se comprueba si en la poblacion nueva se ha perdido la mejor solucion que habia en la poblacion actual, y si es así, se sustituye por la peor solucion en la poblacion nueva. Esto se hace para conservar las mejores soluciones que van apareciendo en la ejecucion del algoritmo. Este proceso se llama elitismo.
  En el esquema estacionario, la nueva poblacion solo tiene 2 cromosomas, y estas dos soluciones compiten por entrar en la poblacion contra las dos peores de estas.

Algorithm 9: Estrctura del Método de búsqueda

Input: p(poblacion)
Output: pnueva(poblacion generada)

```
1  num evaluaciones ← 0
2  pnueva ← p
3  while num evaluaciones < 100000 do
4      poblacionauxiliar ← p
5      poblacionauxiliar ← OperadorSeleccion(poblacionauxiliar)
6      poblacionauxiliar ← OperadorCruce(poblacionauxiliar)
7      poblacionauxiliar ← OperadorMutacion(poblacionauxiliar)
8      pnueva ← OperadorReemplazo(poblacionauxiliar)
9      num evaluaciones ← num evaluaciones + veces que han sido calculadas las distancias de las soluciones
10 return pnueva
```

3.2. Algoritmos Genéticos y reemplazamiento

3.2.1. Esquema de evolución

En esta práctica vemos dos esquemas distintos de evolución:

- Esquema generacional con elitismo.
  Las poblaciones en cada iteracion nunca pierden la mejor solucion de esta Por

lo tanto, en el proceso de reemplazamiento se comprueba si se ha perdido la mejor solucion de la poblacion t, y si es así, se sustituye por la peor solucion de la poblacion t+1, que es la que va a pasar a ser la actual en la proxima iteracion.
Con este esquema podemos ver que las poblaciones cambian de forma radical muy rapidamente.

- Esquema estacionario.
  Las poblaciones en cada iteración pueden como mucho perder las dos peores soluciones de esta, ya que entran en una especie de torneo donde compiten con las dos soluciones conseguidas en dicha iteracion. Las dos mejores serán las que se mantienen. En este esquema, las poblaciones no cambian con tanta rapidez como el esquema generacional.

3.3. Algoritmos Meméticos

Los algoritmos meméticos son técnicas de optimización que combinan conceptos tomados de otras metaheuristicas, como la búsqueda basadda en poblaciones y la busqueda de mejora local. Estos algoritmos implementados, tienen la misma estructura que los algoritmos genéticos generacionales, pero con la diferencia de que:

- En la primera variante del algoritmo, *AM-(10,1.0)* Cada 10 generaciones, se aplica la BL sobre todos los cromosomas de la población. Es decir, cuando el numero de iteraciones es multiplo de 10, se aplica la BL sobre todos los cromosomas de la población, lo que aumenta considerablemente la capacidad del algoritmo en buscar mejores soluciones.
- En la segunda variante del algoritmo, *AM-(10,0.1)* Cada 10 generaciones, se aplica la BL sobre un subconjunto de cromosomas de la población seleccionado aleatoriamente con probabilidad igual a 0.1 para cada cromosoma. Es decir, cuando el numero de iteraciones es multiplo de 10, se aplica sobre n*0.1 aleatorios cromosomas de la poblacion la busqueda local.
- En la tercera variante del algoritmo, *AM-(10,0.1mej)* Cada 10 generaciones, se aplica la BL sobre los 10 % mejores cromosomas de la poblacion.

3.3.1. Búsqueda Local(BL)

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 10: Algoritmo de búsqueda local

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

CopiaSolucion ← ∅
CopiaDistancia ← ∅
Vecindario ← ∅

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

VectorDistancias ← GenerarVectorDistancias()
DispersionComparacion ← Calculardispersion(VectorDistancias)

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

    Actualizamos las variables antes de recorrer el vecindario
    Copiasolucion ← solucion
    Mejora ← FALSE
    dispersioncomparacion ← Dispersion

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

4. PROCEDIMIENTO CONSIDERADO PARA DESARROLLAR LA PRÁCTICA

Para esta práctica se ha usado un entorno de programación de C++ común, con las carpetas *bin, obj, include, src, lib, data* y el archivo *Makefile* que se encuentra en la carpeta raíz de la que cuelgan dichas carpetas. He modularizado la mayor parte de funciones usadas en dos ficheros, el de declaración llamado funciones.h que contiene las declaraciones de las funciones, y el funciones.cpp que contiene las definiciones de las funciones.
He usado tambien el fichero random.h para la generacion de numeros aleatorios con las semillas del vector.
Los algoritmos geneticos generacionales, estacionarios y los algoritmos memeticos los tengo modularizados tambien en sus correspondientes ficheros, y un main general que ejecuta todos los algoritmos además de un main independiente por cada uno de ellos.

Figura 1: Diagrama en forma de árbol de los ficheros utilizados

El fichero makefile contiene las ordenes necesarias para compilar el proyecto, que básicamente lo que hace es compilar el fichero *main.cpp* y generar el ejecutable enlazandolo con los archivos objetos funciones.o y random.o, que tambien son compilados. He seguido la idea del pseudocódigo que se proporciona en Prado, tanto del seminario como del guion de la practica. La directiva de compilacion usada para optimizar el programa es la de *g++ -O2*
Para ejecutar entonces el programa basta con situarse en la carpeta donde se encuentran estos directorios y llamar a make. Luego con realizar una redimension de entrada básica al ejecutable funciona correctamente
Por ejemplo: bin/main >data/ficheroentrada.txt
Basicamente, este es el procedimiento considerado a la hora de desarrollar esta prác-

tica, apuntes de la asignatura, algunas dudas resueltas en clase y algo de búsqueda de información sobre la STL.

5. EXPERIMENTOS 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	52,22	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	74,12	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	97,1061	81,35	0,0022106500
GKD-b_36_n125_m37	471,2490	67,02	0,0230018000
GKD-b_37_n125_m37	502,6070	69,43	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	437,38	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	0,0042312100
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	483,6690	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	0,0000312290
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	0,0032435300
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	0,0070431400
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	0,0361438000
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	0,0111683000
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	0,0946291000
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	563,6460	55,83	0,0816988000

(b) Tabla de resultados de BL

Figura 2: 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 3: 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.

¿Por qué Greedy tiene tiempos menores?
El algoritmo greedy es más eficiente respecto a lo que tiempo se refiere, ya que :

Generacion de primera solucion El algoritmo greedy solo tiene que generar dos elementos aleatorios a introducir en la primera solucion, mientras que el algoritmo de busqueda local tiene que generar aleatoriamente una solucion completa.

Generacion de vecindario El algoritmo de Busqueda Local tiene que generar el vecindario completo, lo que requiere un coste de O(X*Y), siendo X el numero de elementos de la solucion, e Y el número de elementos restantes. Ya que el conjunto de solucion junto a los restantes son los N elementos, este paso tiene un coste de O(N), lo que supone una diferencia de tiempo con respecto a Greedy

Evaluacion de funcion objetivo En este caso, los dos algoritmos se conforman de forma muy parecida, ya que se realiza una factorización en el cálculo del Vector de Distancias en ambos, por lo que no se pueden extraer conclusiones de aquí.

Actualización de la solucion constante El algoritmo Greedy actualiza sí o sí la solución al final de cada iteracion, ya que aunque ninguno mejore la dispersion, se escoge el que menos la empeore. Si el algorimo de Busqueda Local no encuentra ningun vecino que mejore la dispersion, termina su ejecución.

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

¿Por qué BL tiene menor media de Desviación?

El algoritmo de Busqueda Local tiene una menor media de desviacion que el algoritmo Greedy, es decir, que las dispersiones obtenidas de media con el algoritmo de Busqueda Local son menores(y por consiguiente, mejores) que las obtenidas por el algoritmo Greedy. La desviacion se calcula como la media de las desviaciones, en porcentaje, del valor obtenido por cada metodo en cada instancia respecto al mejor valor conocido para ese caso.

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

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 soluciones.
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 solucion el que menos la empeore.
Esto evita que el algoritmo de Busqueda Local vaya hacia soluciones peores(mínimos locales), y siempre se asegure que cuando actualiza la solucion es para una mejor dispersion.

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

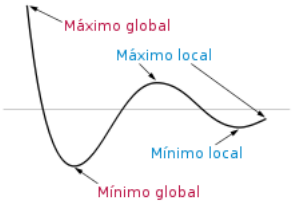

Figura 4: Gráfica que muestra el comportamiento de una búsqueda de una solucion

5.1. 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
AM-(10,1.0)	-0,0978477902	35,944350200000
AM-(10,0.1)	14,6659436747	6,939745800000
AM-(10,0.1mej)	32,5667149186	5,862021000000

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

Observando la tabla, podemos ver que el algoritmo que mejores tiempo consigue es el memetico donde cada 10 iteraciones se realiza una busqueda local completa por cada cromosoma de la poblacion actual, es por eso que tiene los tiempos mas altos. Esto tiene sentido ya que cada 10 iteraciones todas los cromosomas de la poblacion mejoran con dicho algoritmo, por lo que no va a quedarse estancado en optimos locales, escapando muy rapido de ellos.

Vemos una mejora evidente en el algoritmo genetico de esquema generacional frente al del esquema estacionario, esto se debe a que el esquema estacionario no mejora con tanta rapidez como lo hace el generacional, ya que como mucho en cada iteracion podra mejorar 2 soluciones, mientras que el generacional puede mejorar hasta n-1 soluciones, siendo n el numero de cromosomas de la poblacion, ya que es un algoritmo elitista, que nunca pierde la mejor solucion de la poblacion actual antes de ser reemplazada por la siguiente.

Las diferencias entre el AGE-Uniforme y el AGE-Posición no son significativas por el fitness obtenido, pero si por el tiempo de ejecucion, ya que el operador de cruce

basado en posicion requiere mucho menos tiempo que el uniforme. Esto se debe a que el uniforme en muchas ocasiones, llama al operador de reparación, que tiene un coste bastante alto, y además solo se obtiene un hijo con los dos padres, mientras que en el basado en posicion, se obtienen dos hijos de dos padres lo que acelera bastante el proceso. Estas diferencias son aplicables igualmente en el esquema generacional, aunque la probabilidad de cruce sea 0.7 en vez de 1

Respecto a los algoritmos memeticos, el algoritmo ganador respecto a fitness como he comentado anteriormente es el de la primera variante, ya que al aplicar una busqueda local sobre todos los cromosomas, nunca se queda estancado en la mejora de soluciones. El de la segunda variante, que aplica la busqueda local sobre un 10 % aleatorio de los cromosomas de la poblaicon, es el segundo mejor que hemos conseguido respecto a fitness, pero es un poco peor respecto a tiempo que la ultima variante. Este buen fitness conseguido se debe a la aleatoriedad de los cromosomas seleccionados para la busqueda local, ya que los cromosomas que se seleccionan aleatoriamente pueden estar en maximos locales, por lo que gracias a la busqueda local, se pueden mejorar dichas soluciones.

El ultimo como vemos obtiene el peor resultado respecto a fitness de los 3 algoritmos memeticos, y esto se debe a que al aplicarse la busqueda local sobre el 10 % de mejores soluciones, muchas veces la búsqueda local se queda estancada en dichas soluciones porque en el entorno no se encuentra mejora, entonces al aplicarse esa busqueda sobre las mejores no se mejora tanto como la anterior variante, que aleatoriamente es probable que cada x iteraciones seleccione cromosomas que se encuentran estancados, mientras que esta variante no, siempre va a coger a los mejores y por el elitismo del esquema generacional, es probable que se aplica muchas veces a los mismos cromosomas.

Algoritmo Generacional Cruce Posicion			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo(s)**
GKD-b_1_n25_m2	0	0,00	1,29204
GKD-b_2_n25_m2	0	0,00	1,22937
GKD-b_3_n25_m2	0	0,00	1,29984
GKD-b_4_n25_m2	0	0,00	1,58829
GKD-b_5_n25_m2	0	0,00	1,42524
GKD-b_6_n25_m7	27,2722	53,37	1,31301
GKD-b_7_n25_m7	22,6883	37,86	1,27461
GKD-b_8_n25_m7	24,9729	32,88	1,5447
GKD-b_9_n25_m7	19,9912	14,62	1,2667
GKD-b_10_n25_m7	30,2731	23,15	1,45371
GKD-b_11_n50_m5	7,44909	74,14	2,59923
GKD-b_12_n50_m5	5,65465	62,49	2,08003
GKD-b_13_n50_m5	9,15487	74,20	2,61647
GKD-b_14_n50_m5	13,7464	87,90	1,86196
GKD-b_15_n50_m5	9,843	71,01	2,2693
GKD-b_16_n50_m15	72,3568	40,92	3,1282
GKD-b_17_n50_m15	48,1076	0,00	2,22071
GKD-b_18_n50_m15	64,1659	32,68	2,32059
GKD-b_19_n50_m15	72,0802	35,61	2,40613
GKD-b_20_n50_m15	75,2435	36,59	3,12248
GKD-b_21_n100_m10	30,5304	54,69	2,68214
GKD-b_22_n100_m10	23,8794	42,78	3,63799
GKD-b_23_n100_m10	32,7316	53,12	2,58214
GKD-b_24_n100_m10	30,1696	71,36	2,61942
GKD-b_25_n100_m10	27,0688	36,46	2,47838
GKD-b_26_n100_m30	357,588	52,81	4,25805
GKD-b_27_n100_m30	296,27	57,10	3,52556
GKD-b_28_n100_m30	307,177	65,37	3,46504
GKD-b_29_n100_m30	210,471	34,69	4,97631
GKD-b_30_n100_m30	303,905	58,05	3,51115
GKD-b_31_n125_m12	28,7025	59,08	3,4585
GKD-b_32_n125_m12	49,1306	61,76	3,41972
GKD-b_33_n125_m12	38,5249	51,90	3,89056
GKD-b_34_n125_m12	43,5066	55,21	4,53728
GKD-b_35_n125_m12	46,4377	61,00	4,42925
GKD-b_36_n125_m37	308,568	49,63	6,76584
GKD-b_37_n125_m37	292,751	32,06	4,97138
GKD-b_38_n125_m37	489,401	61,59	4,65267
GKD-b_39_n125_m37	388,36	56,59	5,76321
GKD-b_40_n125_m37	352,16	49,40	4,74698
GKD-b_41_n150_m15	45,1672	48,31	3,63887
GKD-b_42_n150_m15	58,1567	53,94	3,35883
GKD-b_43_n150_m15	38,7663	30,99	3,28631
GKD-b_44_n150_m15	70,8633	63,40	4,24379
GKD-b_45_n150_m15	57,136	51,39	3,18887
GKD-b_46_n150_m45	599,277	62,00	5,89684
GKD-b_47_n150_m45	494,274	53,75	7,09553
GKD-b_48_n150_m45	380,679	40,44	5,82675
GKD-b_49_n150_m45	649,107	65,12	5,68422
GKD-b_50_n150_m45	671,298	62,93	4,7414
Media Desv:	**45,4862646141**		
Media Tiempo:	**3,312911800000**		

Figura 5: Desviaciones y tiempos de Algoritmo Genetico Generacional con Cruce Basado en Posicion

Algoritmo Generacional Cruce Uniforme			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo(s)**
GKD-b_1_n25_m2	0	0,00	1,13781
GKD-b_2_n25_m2	0	0,00	1,17559
GKD-b_3_n25_m2	0	0,00	1,24084
GKD-b_4_n25_m2	0	0,00	1,21566
GKD-b_5_n25_m2	0	0,00	1,17558
GKD-b_6_n25_m7	23,9673	46,94	1,15027
GKD-b_7_n25_m7	53,9403	73,86	5,42681
GKD-b_8_n25_m7	26,1598	35,93	1,05756
GKD-b_9_n25_m7	38,2793	55,41	1,15215
GKD-b_10_n25_m7	46,454	49,92	5,78582
GKD-b_11_n50_m5	10,2306	81,17	3,89298
GKD-b_12_n50_m5	1,41402	-11,97	5,9073
GKD-b_13_n50_m5	0,19514	-39,48	5,48951
GKD-b_14_n50_m5	12,4015	86,59	2,21683
GKD-b_15_n50_m5	8,94943	68,12	1,60966
GKD-b_16_n50_m15	96,9125	55,89	3,01695
GKD-b_17_n50_m15	19,3126	-84,26	34,1734
GKD-b_18_n50_m15	125,096	65,47	2,34721
GKD-b_19_n50_m15	91,8608	49,48	2,74763
GKD-b_20_n50_m15	97,2504	50,94	2,40503
GKD-b_21_n100_m10	21,1639	34,64	2,43629
GKD-b_22_n100_m10	23,1014	36,65	25,7476
GKD-b_23_n100_m10	35,1113	56,30	3,10414
GKD-b_24_n100_m10	13,0992	15,72	28,3542
GKD-b_25_n100_m10	141,301	87,83	35,6458
GKD-b_26_n100_m30	285,536	40,91	4,46756
GKD-b_27_n100_m30	213,914	40,58	4,48864
GKD-b_28_n100_m30	244,587	56,51	4,53394
GKD-b_29_n100_m30	214,755	36,00	4,34235
GKD-b_30_n100_m30	188,153	32,25	4,36377
GKD-b_31_n125_m12	55,3014	78,76	4,34117
GKD-b_32_n125_m12	40,1096	53,16	3,25636
GKD-b_33_n125_m12	270.594	99,99	57,7285
GKD-b_34_n125_m12	35,8875	45,70	3,99922
GKD-b_35_n125_m12	33,8321	46,46	3,42254
GKD-b_36_n125_m37	253,127	38,59	6,62717
GKD-b_37_n125_m37	373,656	46,77	6,25037
GKD-b_38_n125_m37	315,38	40,40	8,7993
GKD-b_39_n125_m37	302,782	44,32	5,87947
GKD-b_40_n125_m37	314,139	43,28	7,07518
GKD-b_41_n150_m15	46,769	50,08	3,47348
GKD-b_42_n150_m15	49,7234	46,12	3,63142
GKD-b_43_n150_m15	62,861	57,44	3,57339
GKD-b_44_n150_m15	64,0132	59,48	3,2906
GKD-b_45_n150_m15	50,391	44,88	4,39318
GKD-b_46_n150_m45	313,033	27,24	8,73294
GKD-b_47_n150_m45	292,366	21,81	9,96125
GKD-b_48_n150_m45	386,291	41,30	8,72237
GKD-b_49_n150_m45	475,762	52,41	8,56159
GKD-b_50_n150_m45	420,746	40,85	9,63653
Media Desv:	**40,0088991109**		
Media Tiempo:	**7,463298200000**		

Figura 6: Desviaciones y tiempos de Algoritmo Genetico Generacional con Cruce Uniforme

Algoritmo Generacional Estacionario Posicion			
Caso	Coste medio obtenido	Desv	Tiempo(s)
GKD-b_1_n25_m	0	0,00	2,19455
GKD-b_2_n25_m	0	0,00	2,46809
GKD-b_3_n25_m	0	0,00	2,18506
GKD-b_4_n25_m	0	0,00	2,51318
GKD-b_5_n25_m	0	0,00	2,28503
GKD-b_6_n25_m	20,3955	37,64	2,88895
GKD-b_7_n25_m	28,9062	51,23	2,65469
GKD-b_8_n25_m	31,5014	46,79	2,62135
GKD-b_9_n25_m	29,2807	41,70	2,65865
GKD-b_10_n25_	26,8843	13,46	2,69581
GKD-b_11_n50_	11,8397	83,73	3,73719
GKD-b_12_n50_	8,44707	74,89	3,82263
GKD-b_13_n50_	10,7425	78,01	4,51812
GKD-b_14_n50_	10,5282	84,20	3,8063
GKD-b_15_n50_	2,94375	2,13	4,25966
GKD-b_16_n50_	129,594	67,02	4,77252
GKD-b_17_n50_	64,6073	25,54	4,33776
GKD-b_18_n50_	102,807	57,98	4,36899
GKD-b_19_n50_	92,0281	49,57	4,26829
GKD-b_20_n50_	84,7451	43,70	4,53975
GKD-b_21_n100	36,1919	61,78	4,86002
GKD-b_22_n100	50,7113	73,05	6,06697
GKD-b_23_n100	40,0372	61,67	4,43894
GKD-b_24_n100	43,915	80,32	4,52709
GKD-b_25_n100	43,9993	60,91	4,3495
GKD-b_26_n100	398,337	57,64	5,69297
GKD-b_27_n100	392,754	67,64	5,36053
GKD-b_28_n100	304,877	65,11	5,35588
GKD-b_29_n100	413,374	66,75	5,58292
GKD-b_30_n100	364,212	65,00	6,3832
GKD-b_31_n125	45,9693	74,45	5,88754
GKD-b_32_n125	67,3469	72,10	6,01124
GKD-b_33_n125	46,6047	60,24	6,56328
GKD-b_34_n125	61,459	68,29	5,84596
GKD-b_35_n125	70,599	74,34	6,90021
GKD-b_36_n125	371,99	58,22	7,28885
GKD-b_37_n125	634,805	68,67	8,41454
GKD-b_38_n125	574,123	67,26	7,38416
GKD-b_39_n125	514,417	67,23	8,00534
GKD-b_40_n125	473,679	62,38	8,31196
GKD-b_41_n150	51,515	54,68	5,30314
GKD-b_42_n150	95,3161	71,89	5,11484
GKD-b_43_n150	100,201	73,30	5,79904
GKD-b_44_n150	95,4316	72,82	5,96662
GKD-b_45_n150	82,0678	66,16	5,38967
GKD-b_46_n150	577,659	60,57	7,52263
GKD-b_47_n150	490,658	53,41	6,86753
GKD-b_48_n150	518,612	56,28	7,02272
GKD-b_49_n150	629,565	64,04	6,73044
GKD-b_50_n150	679,926	63,40	6,59611
Media Desv:	**54,5438597140**		
Media Tiempo:	**5,062808200000**		

Figura 7: Desviaciones y tiempos de Algoritmo Genetico Estacionario con Cruce Basado en Posicion

Algoritmo Generacional Estacionario Uniforme			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo(s)**
GKD-b_1_n25_m	0	0,00	4,12507
GKD-b_2_n25_m	0	0,00	4,40917
GKD-b_3_n25_m	0	0,00	4,13849
GKD-b_4_n25_m	0	0,00	4,23267
GKD-b_5_n25_m	0	0,00	4,35989
GKD-b_6_n25_m	32,4313	60,78	4,95066
GKD-b_7_n25_m	26,7614	47,32	5,25469
GKD-b_8_n25_m	22,2701	24,74	4,90711
GKD-b_9_n25_m	29,2807	41,70	5,24524
GKD-b_10_n25_	30,2731	23,15	5,15884
GKD-b_11_n50_	7,80036	75,31	7,68975
GKD-b_12_n50_	5,12149	37,76	7,94563
GKD-b_13_n50_	11,2391	78,98	7,33929
GKD-b_14_n50_	10,5282	84,20	7,26938
GKD-b_15_n50_	16,1091	82,29	7,80707
GKD-b_16_n50_	76,9332	44,44	8,55234
GKD-b_17_n50_	133,835	64,05	8,13868
GKD-b_18_n50_	121,818	64,54	8,66692
GKD-b_19_n50_	141,531	67,21	8,04705
GKD-b_20_n50_	109,921	56,59	9,01952
GKD-b_21_n100	49,8307	72,24	9,45472
GKD-b_22_n100	38,6766	64,67	10,0716
GKD-b_23_n100	43,9008	65,05	8,61634
GKD-b_24_n100	50,8456	83,01	9,67722
GKD-b_25_n100	37,2907	53,87	9,34306
GKD-b_26_n100	464,791	63,70	11,1289
GKD-b_27_n100	317,522	59,97	10,4637
GKD-b_28_n100	376,596	71,75	10,8001
GKD-b_29_n100	318,626	56,86	10,5541
GKD-b_30_n100	347,675	63,33	10,3946
GKD-b_31_n125	64,1745	81,70	11,4083
GKD-b_32_n125	56,9464	67,01	11,4802
GKD-b_33_n125	49,6298	62,66	11,6137
GKD-b_34_n125	67,1758	70,99	11,2684
GKD-b_35_n125	63,3535	71,41	11,3743
GKD-b_36_n125	397,282	60,88	14,557
GKD-b_37_n125	615,27	67,67	15,0059
GKD-b_38_n125	444,676	57,73	14,1932
GKD-b_39_n125	422,706	60,12	14,8948
GKD-b_40_n125	390,617	54,38	14,7682
GKD-b_41_n150	62,1721	62,45	9,9877
GKD-b_42_n150	123,194	78,25	10,3234
GKD-b_43_n150	102,814	73,98	10,4682
GKD-b_44_n150	63,6524	59,25	10,1271
GKD-b_45_n150	86,9856	68,07	9,35756
GKD-b_46_n150	725,818	68,62	13,1913
GKD-b_47_n150	458,936	50,19	13,4575
GKD-b_48_n150	567,176	60,02	14,4823
GKD-b_49_n150	554,187	59,15	13,8588
GKD-b_50_n150	574,385	56,67	12,6269
Media Desv:	**55,9744088626**		
Media Tiempo:	**9,524131200000**		

Figura 8: Desviaciones y tiempos de Algoritmo Genetico Estacionario con Cruce Uniforme

Algoritmo Generacional Memetico 1			
Caso	Coste medio obtenido	Desv	Tiempo(s)
GKD-b_1_n25_m	0	0,00	1,32844
GKD-b_2_n25_m	0	0,00	1,3617
GKD-b_3_n25_m	0	0,00	1,30336
GKD-b_4_n25_m	0	0,00	1,29996
GKD-b_5_n25_m	0	0,00	1,30408
GKD-b_6_n25_m	12,718	0,00	2,31741
GKD-b_7_n25_m	14,0988	0,00	2,19363
GKD-b_8_n25_m	16,7612	0,00	2,28797
GKD-b_9_n25_m	17,0692	0,00	2,17895
GKD-b_10_n25_	23,2652	0,00	2,22972
GKD-b_11_n50_	1,9261	0,00	3,47111
GKD-b_12_n50_	2,0513	-2,11	3,30546
GKD-b_13_n50_	2,36231	0,00	3,52936
GKD-b_14_n50_	3,79036	56,12	3,31325
GKD-b_15_n50_	2,85313	0,00	3,50102
GKD-b_16_n50_	42,7458	0,00	9,0142
GKD-b_17_n50_	48,1076	0,00	9,08336
GKD-b_18_n50_	43,1961	0,00	8,53952
GKD-b_19_n50_	46,4125	0,00	8,94903
GKD-b_20_n50_	47,7151	0,00	9,68906
GKD-b_21_n100	13,2385	-4,48	11,8858
GKD-b_22_n100	17,3662	21,32	12,2146
GKD-b_23_n100	13,623	-12,64	11,9184
GKD-b_24_n100	16,5994	47,95	11,6897
GKD-b_25_n100	12,8401	-33,76	12,9156
GKD-b_26_n100	159,192	-5,99	48,8407
GKD-b_27_n100	124,171	-2,36	50,019
GKD-b_28_n100	139,692	23,85	47,6641
GKD-b_29_n100	140,612	2,25	49,5729
GKD-b_30_n100	131,215	2,85	46,4946
GKD-b_31_n125	15,0836	16,02	20,8442
GKD-b_32_n125	19,4661	3,48	18,1636
GKD-b_33_n125	14,3142	-21,96	19,2059
GKD-b_34_n125	21,3846	8,87	18,3228
GKD-b_35_n125	14,6778	-16,60	20,6903
GKD-b_36_n125	148,066	-4,98	82,1922
GKD-b_37_n125	195,968	-1,49	82,7595
GKD-b_38_n125	186,573	-0,75	86,3751
GKD-b_39_n125	171,359	1,62	80,1109
GKD-b_40_n125	176,972	-0,69	86,1119
GKD-b_41_n150	24,7014	4,72	28,6842
GKD-b_42_n150	24,4634	-7,73	30,1066
GKD-b_43_n150	20,8233	-19,11	31,0408
GKD-b_44_n150	20,8166	-16,01	31,9829
GKD-b_45_n150	24,7032	-9,66	31,7786
GKD-b_46_n150	208,037	-9,48	145,891
GKD-b_47_n150	223,16	-2,44	151,028
GKD-b_48_n150	180,001	-25,97	147
GKD-b_49_n150	242,116	6,49	150,445
GKD-b_50_n150	243,47	-2,21	151,068
Media Desv:	**-0,0978477902**		
Media Tiempo:	**35,944350200000**		

Figura 9: Desviaciones y tiempos de Algoritmo Memetico *AM-(10,1.0)*

Algoritmo Generacional Memetico 2			
Caso	Coste medio obtenido	Desv	Tiempo(s)
GKD-b_1_n25_m	0	0,00	1,14341
GKD-b_2_n25_m	0	0,00	1,15823
GKD-b_3_n25_m	0	0,00	1,14026
GKD-b_4_n25_m	0	0,00	1,17427
GKD-b_5_n25_m	0	0,00	1,14608
GKD-b_6_n25_m	13,4793	5,65	1,30756
GKD-b_7_n25_m	14,0988	0,00	1,28479
GKD-b_8_n25_m	16,7612	0,00	1,33771
GKD-b_9_n25_m	25,0145	31,76	1,29223
GKD-b_10_n25_	23,2652	0,00	1,27562
GKD-b_11_n50_	3,67407	47,58	2,05506
GKD-b_12_n50_	4,70656	54,93	1,95467
GKD-b_13_n50_	3,08164	23,34	1,9569
GKD-b_14_n50_	5,64178	70,52	1,86373
GKD-b_15_n50_	3,22012	11,40	1,9638
GKD-b_16_n50_	42,7458	0,00	3,02482
GKD-b_17_n50_	48,1076	0,00	2,77017
GKD-b_18_n50_	52,0761	17,05	2,86423
GKD-b_19_n50_	46,8501	0,93	2,84456
GKD-b_20_n50_	55,8	14,49	2,87682
GKD-b_21_n100	13,2385	-4,48	3,47362
GKD-b_22_n100	19,4482	29,74	3,69176
GKD-b_23_n100	17,9843	14,67	3,55749
GKD-b_24_n100	17,4087	50,37	3,58294
GKD-b_25_n100	24,6188	30,13	3,48452
GKD-b_26_n100	173,351	2,67	9,00499
GKD-b_27_n100	151,358	16,03	8,92843
GKD-b_28_n100	202,121	47,37	8,46329
GKD-b_29_n100	147,681	6,93	9,65493
GKD-b_30_n100	150,358	15,22	9,20863
GKD-b_31_n125	18,9635	38,06	5,54492
GKD-b_32_n125	20,7943	9,64	5,00968
GKD-b_33_n125	22,3013	16,90	5,09123
GKD-b_34_n125	26,5147	26,50	5,21903
GKD-b_35_n125	19,5094	7,16	5,23479
GKD-b_36_n125	200,266	22,39	15,0379
GKD-b_37_n125	219,35	9,33	14,175
GKD-b_38_n125	281,456	33,22	16,2351
GKD-b_39_n125	211,269	20,20	15,1548
GKD-b_40_n125	230,611	22,73	14,6141
GKD-b_41_n150	32,9218	29,09	6,63424
GKD-b_42_n150	30,0814	10,94	6,45382
GKD-b_43_n150	24,8378	-7,72	6,48469
GKD-b_44_n150	23,3418	-11,11	6,73589
GKD-b_45_n150	24,7283	-12,31	6,49018
GKD-b_46_n150	234,155	2,74	24,8468
GKD-b_47_n150	305,749	25,23	21,2904
GKD-b_48_n150	251,407	9,81	21,0616
GKD-b_49_n150	211,439	-7,08	23,8873
GKD-b_50_n150	252,116	1,29	22,3003
Media Desv:	**14,6659436747**		
Media Tiempo:	**6,939745800000**		

Figura 10: Desviaciones y tiempos de Algoritmo Memetico *AM-(10,0.1)*

Algoritmo Generacional Memetico 3			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo(s)**
GKD-b_1_n25_m	0	0,00	1,18727
GKD-b_2_n25_m	0	0,00	1,19125
GKD-b_3_n25_m	0	0,00	1,18243
GKD-b_4_n25_m	0	0,00	1,16339
GKD-b_5_n25_m	0	0,00	1,16443
GKD-b_6_n25_m	20,3955	37,64	1,27123
GKD-b_7_n25_m	19,6091	28,10	1,28701
GKD-b_8_n25_m	21,8265	23,21	1,35612
GKD-b_9_n25_m	29,2807	41,70	1,28386
GKD-b_10_n25_	26,238	11,33	1,30411
GKD-b_11_n50_	5,21583	63,07	2,12512
GKD-b_12_n50_	6,70799	68,38	1,92817
GKD-b_13_n50_	12,8654	81,64	1,92322
GKD-b_14_n50_	6,50866	74,45	1,95766
GKD-b_15_n50_	9,88369	71,13	1,92653
GKD-b_16_n50_	75,4985	43,38	2,92204
GKD-b_17_n50_	61,7735	22,12	2,72803
GKD-b_18_n50_	64,9047	33,45	2,69616
GKD-b_19_n50_	82,8634	43,99	2,716
GKD-b_20_n50_	82,0733	41,86	2,72719
GKD-b_21_n100	29,108	52,48	3,24772
GKD-b_22_n100	22,4415	39,11	3,31844
GKD-b_23_n100	28,1138	45,42	3,23292
GKD-b_24_n100	19,5417	55,78	3,26115
GKD-b_25_n100	25,4063	32,30	3,33184
GKD-b_26_n100	159,192	-5,99	7,55917
GKD-b_27_n100	221,25	42,55	7,42219
GKD-b_28_n100	202,934	47,58	7,45951
GKD-b_29_n100	159,705	13,93	7,96243
GKD-b_30_n100	186,968	31,82	7,31181
GKD-b_31_n125	23,6826	50,41	4,72974
GKD-b_32_n125	31,5137	40,38	4,68643
GKD-b_33_n125	31,4656	41,11	4,71503
GKD-b_34_n125	23,1404	15,78	4,5444
GKD-b_35_n125	21,5231	15,85	4,62413
GKD-b_36_n125	223,046	30,31	11,8952
GKD-b_37_n125	248,371	19,92	12,1549
GKD-b_38_n125	252,923	25,68	12,4745
GKD-b_39_n125	213,393	21,00	12,1285
GKD-b_40_n125	254,552	30,00	11,9807
GKD-b_41_n150	52,2652	55,33	5,57958
GKD-b_42_n150	39,4701	32,13	5,74599
GKD-b_43_n150	29,4663	9,20	5,73632
GKD-b_44_n150	44,7291	42,02	5,41781
GKD-b_45_n150	36,4511	23,81	5,84002
GKD-b_46_n150	256,872	11,34	18,9079
GKD-b_47_n150	326,516	29,99	18,1002
GKD-b_48_n150	406,299	44,19	16,9506
GKD-b_49_n150	315,659	28,27	18,2037
GKD-b_50_n150	315,764	21,19	18,537
Media Desv:	**32,5667149186**		
Media Tiempo:	**5,862021000000**		

Figura 11: Desviaciones y tiempos de Algoritmo Memetico *AM-(10,0.1mej)*