









PROBLEMA DE LA MÁXIMA DIVERSIDAD

2.a: Técnicas de Búsqueda basadas en Poblaciones

Javier Ramírez Pulido
javierramirezp@correo.ugr.es

3ºCSI GRUPO A-2
20501292n VIERNES 17:30 – 19:30

INDICE

 <u>Breve descripción/formulación del problema</u>	3
 <u>Desarrollo de la practica</u>	4
 <u>Explicación de genéticos y memeticos</u>	6
 <u>Pseudocodigos</u>	10
 <u>Experimento y análisis de los resultados</u>	22
○ Descripción de los casos del problema empleados	
○ Resultados obtenidos	
○ Análisis de resultados	
 <u>Referencias bibliográficas</u>	16

Descripción del problema.

El problema de la Máxima Diversidad es un problema de optimización combinatoria que consiste en, partiendo de un conjunto de N elementos, formar agrupaciones con la mayor diversidad posible. Estas agrupaciones podrían tener diferentes dimensiones, pero en nuestro caso será de tamaño m (m elementos) y el conjunto original S deberá tener estrictamente más elementos que aquellos que se pretenden seleccionar ($N > m$).

Con diversidad nos referimos a la suma de las distancias entre los pares de elementos seleccionados. Esta distancia d_{ij} entre el elemento i y el elemento j se obtiene de la distancia euclidiana:

$$d_{ij} = \sqrt{\sum_{k=1}^t (a_{ik} - a_{jk})^2}.$$

Que se almacenará en una matriz $D = (d_{ij})$ de dimensión $n \times n$ que contiene las distancias entre ellos. Estas distancias cumplen que $d_{ij} = d_{ji}$ y que $d_{ii} = 0$. Las distancias dependen del contexto de la aplicación en que se utilicen. Este problema es NP-Completo, lo que quiere decir que tiene una alta complejidad computacional. De esta forma, el tiempo de resolución crece exponencialmente con el tamaño del problema.

La complejidad del problema viene dada por la combinatoria $\binom{n}{m}$ donde n es el número de elementos totales y m el número de elementos a escoger. Así, para un ejemplo de 10 elementos de los que buscamos seleccionar 3 maximizando la diversidad, tendríamos $\binom{10}{3} = 120$ soluciones posibles, lo cual es asequible para un algoritmo exacto, pero para casos en los que el número de soluciones es elevado deja de ser factible.

Otros tipos de medidas de la diversidad existentes son la similitud del coseno y la medida de similitud en grupos de solucionadores de problemas en los que se pueden obtener valores positivos y negativos, para los casos en los que se considere que este tipo de medida es muy restrictivo y no adecuada para algunas aplicaciones del problema.

Otros nombres que ha recibido el problema son *problema de k -partición* o *problema del particionamiento equitativo*.

Breve explicación del procedimiento considerado para desarrollar la práctica

El desarrollo de la práctica ha estado basado en los siguientes pasos tras recibir la clase de explicación:

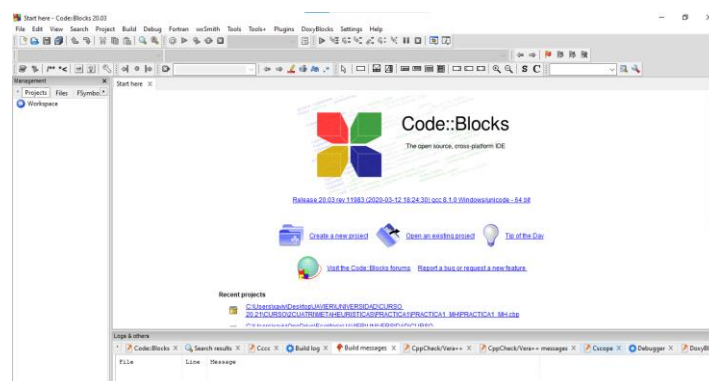
- Revisión del vídeo de explicación y aclaración del desarrollo de la práctica
- Lectura individual del pdf “Guion P2a Poblaciones MDP MHs 2020-21.pdf”
- Lectura individual del pdf “Seminario 2.a. Trayectorias Simples” Es el seminario 3, pero el nombre del archivo cuenta con una errata creo.
- Uso del código de BL realizado para la práctica anterior.

En este caso, la implementación del código ha sido desde un inicio realizada por mí y, en algún momento inicial, consultada a algunas prácticas más de Algorítmica del curso anterior o al pseudocódigo de las diapositivas proporcionadas en la plataforma.

El lenguaje elegido ha sido C++ y el entorno de trabajo es el IDE Code::Blocks 20.03 debido a mi experiencia de uso en la asignatura de Inteligencia Artificial previamente.

La entrega del código se realizará en una carpeta llamada *software* que contendrá todos los archivos necesarios para su ejecución. La forma de realizar la correcta ejecución del programa será:

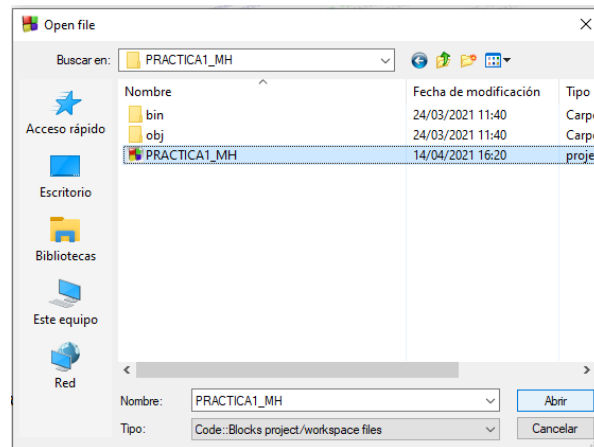
- Abrir CodeBlocks



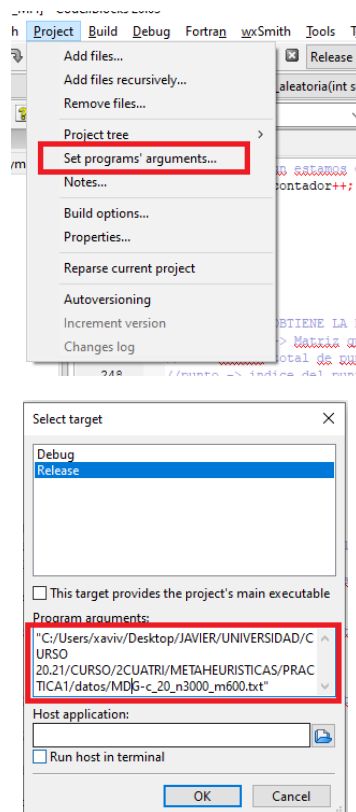
- Seleccionar *Open an existing Project*



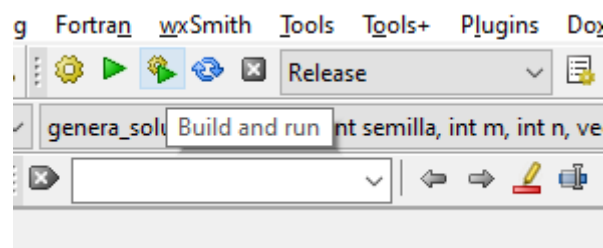
- Nos dirigimos al directorio en el que se encuentre el ejecutable .cbp y lo abrimos



- Una vez abierto, como argumento debemos meter la ruta del archivo del que queremos sacar los datos, que en mi caso sería
 "C:/Users/xaviv/Desktop/JAVIER/UNIVERSIDAD/CURSO
 20.21/CURSO/2CUATRI/METAHEURISTICAS/PRACTICA1/datos/archivo.txt" [SEMILLA]



- Ya podemos construir y ejecutar el programa

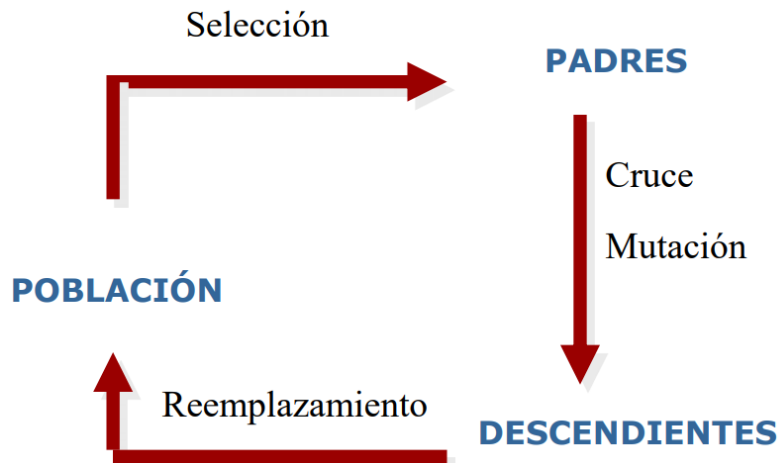


Explicación de los problemas

Algoritmos genéticos

Los algoritmos genéticos son algoritmos de optimización de búsqueda y aprendizaje inspirados en los procesos de evolución natural y evolución genética.

El esquema general que siguen estos algoritmos corresponde al siguiente:



Y necesita de una población de partida, un operador de selección de padres, de cruce, de mutación y de reemplazo.

El procedimiento a seguir es la generación de 50 elementos de la población, que en nuestro caso serán llamados cromosomas y se corresponderán con soluciones generadas aleatoriamente. Es decir, serán 50 vectores binarios de n elementos (cantidad total de puntos en el problema) y m 1's en las posiciones de los m puntos que se seleccionan (cantidad de elementos que se seleccionan por solución), representando el vector de genes de cada 'cromosoma'.

Tras esta generación, se selecciona una cantidad de padres dependiente del subtipo de algoritmo genético que estemos desarrollando a través de un sorteo. Esto es, seleccionando tantas parejas aleatorias como se desee y realizando enfrentamientos entre los dos elementos de cada pareja que se resolverán según el coste que proporcione esa solución. El coste de cada solución equivale a la distancia total entre todos los puntos que se han seleccionado aleatoriamente como parte del vector de genes del cromosoma. Por ejemplo, para el cromosoma $\text{Crom1} = \{0, 1, 0, 1, 1, 0, 1\}$, los elementos que formarían parte de la solución son los equivalentes a las posiciones en las que hay un 1 (1, 3, 4, 6) y el coste de este cromosoma sería la distancia total entre todos los puntos.

Una vez se seleccionan los padres con ese proceso al principio aleatorio y luego por sorteo, toca realizar los cruces. Estos van a depender del tipo, de la probabilidad y la reparación del cruce. Para el caso del cruce posicional no existe reparación, pero si una probabilidad de cruce. Los padres que se cruzan tienen dos hijos que, de entrada, tienen los mismos 1's y 0's que tengan los padres en las posiciones en las que ambos tienen el mismo elemento. Para aquellos genes que son diferentes en cada padre, existen dos formas de resolver el conflicto:

- Cruce uniforme: Por cada posición en la que los padres no concuerdan, se genera un número aleatorio que determina de qué padre se obtiene el valor a introducir en esa

posición. Esto puede provocar que el hijo resultante tenga una cantidad de elementos seleccionados diferente a m , lo cual lo hace una solución inválida. Esto se soluciona con un operador de reparación que:

- Caso en el que sobran elementos: Va quitando el elemento de la solución que más aporte tenga en cada momento para aportar diversidad.
- Caso en el que faltan elementos: Recorre el vecindario buscando el mejor elemento a introducir y este es añadido al vector de genes.
- Cruce posicional: Los elementos dispares en los padres se obtienen de uno de ellos, se introducen en un vector que se reordena aleatoriamente y se introduce en los huecos en blanco del hijo. Esto quita la ventaja de poder introducir elementos del vecindario que sean buenos, pero ahorra tener que reparar los hijos ya que los elementos salen de un mismo padre que ya parte con m elementos seleccionados.

La aplicación de este tipo de algoritmos se lleva a cabo en problemas como el de agente viajero. Conociendo el funcionamiento general de los algoritmos genéticos, veamos las cuatro alternativas que vamos a desarrollar nosotros en esta práctica:

- **Modelo generacional uniforme:** Se seleccionan 50 padres, la probabilidad de cruce es de 0.7, la de mutación es $0.1/n$, se realiza el cruce uniforme, se repara la población y se muta, se introduce el mejor elemento de la población en la que se obtiene tras la mutación sustituyendo a un elemento aleatorio y esta población sustituye a la original antes de repetir el proceso.
- **Modelo generacional posicional:** Se seleccionan 50 padres, la probabilidad de cruce es de 0.7, la de mutación es $0.1/n$, se realiza el cruce posicional, se muta, se introduce el mejor elemento de la población en la que se obtiene tras la mutación sustituyendo a un elemento aleatorio y esta población sustituye a la original antes de repetir el proceso.
- **Modelo estacionario uniforme:** Se seleccionan 2 padres, la probabilidad de cruce es de 1, la de mutación es $0.1/n$, se realiza el cruce uniforme, se repara la población y se muta y se introduce el mejor hijo obtenido en la población original antes de repetir el proceso.
- **Modelo estacionario posicional:** Se seleccionan 2 padres, la probabilidad de cruce es de 1, la de mutación es $0.1/n$, se realiza el cruce posicional, se muta y se introduce el mejor hijo obtenido en la población original antes de repetir el proceso.

Algoritmos meméticos

Son algoritmos poblacionales que pueden verse como una variante de los algoritmos genéticos. La idea principal es incorporar la mayor cantidad de conocimiento del dominio que sea posible durante el proceso de generación de una nueva población. No son mas que técnicas de optimización que combinan conceptos de otras metaheurísticas. Sus aplicaciones van desde el problema del viajante comercio hasta el problema de la partición de grafos, pasando por algunos problemas como la partición de números, coloreado de grafos, asignaciones cuadráticas o el de la mochila multidimensional.

En nuestro caso, la combinación será entre el algoritmo generacional planteado anteriormente y la búsqueda local implementada en la práctica anterior. Esto es porque los algoritmos evolutivos son buenos exploradores, pero malos explotadores, mientras que los algoritmos de búsqueda local son malos exploradores y buenos explotadores.

El procedimiento será escoger el genético que mejor resultado nos dé de los cuatro anteriores y aplicarle BL siguiendo diferentes criterios. Las alternativas que plantearemos son:

- **AM-(10,1.0):** Cada 10 generaciones, se aplica la BL sobre todos los cromosomas de la población.
- **AM-(10,0.1):** Cada 10 generaciones, se aplica la BL sobre un subconjunto de cromosomas de la población seleccionado aleatoriamente.
- **AM-(10,0.1mej):** Cada 10 generaciones, aplicar la BL sobre los $0.1 \cdot n^{\circ}$ cromosomas mejores cromosomas de la población actual.

Esquema de representación

Para los siete algoritmos, la solución será almacenada en un objeto de tipo Cromosoma. Aunque lo que se obtiene para la evaluación es el coste y el tiempo, también existe la opción de mostrar la solución con los elementos seleccionados.

Este objeto Cromosoma contará con 3 atributos:

- Vector de enteros que contendrá el vector de genes en binario que representa la solución de seleccionados de este elemento de la población.
- Variable de tipo double que contendrá el coste que ofrece el vector de genes de este cromosoma concreto.
- Viable booleana con la función de flag que indica en cada momento si este cromosoma necesita una reevaluación de su coste por la modificación de algún gen o no.

Todos los modelos y algoritmos partirán de un conjunto de objetos Cromosoma que simule una población de partida y la salida de todos ellos será un único objeto Cromosoma seleccionado de esta población tras todas las modificaciones necesarias que cumpla con ser el que mayor coste tiene.

Función objetivo

La función objetivo en nuestro caso es la de maximizar el coste que puede proporcionar un único individuo de una población cumpliendo las restricciones de tener m puntos seleccionados que no se repitan.

Esta función a nivel de programa sería aquella que calcula el coste del vector de genes de un Cromosoma, ya sea porque se acaba de crear o porque se ha modificado uno existente. Esta función objetivo se utiliza para cada algoritmo un total de 100000 veces para que todos los algoritmos, independientemente de las generaciones que realicen, tengan las mismas oportunidades. Se llama 'calcula_coste' y es la siguiente:

algoritmo calcula_coste

variables locales

tipo double *distancia_total_solucion* <- 0

tipo entero *punto_seleccionado* <- 0, *punto_restante* <- 0

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

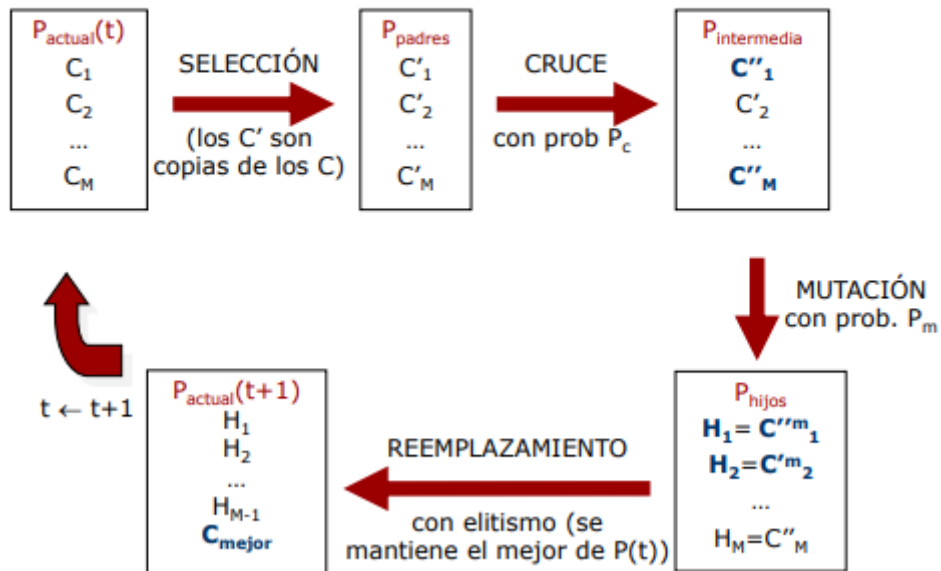
tipo vector de enteros *Solucion* (vector con los elementos de la solución)

inicio

1. *Mientras no hayamos recorrido todos los elementos de la solución (punto_seleccionado < tamaño(Solucion) - 1)*
 - a. *Nos situamos en el siguiente elemento del vector Solucion (punto_restante <- siguiente(punto_seleccionado))*
 - b. *Iteramos por todos los puntos desde punto_restante hasta el final (punto_restante < tamaño(Solucion))*
 - i. *Vamos sumando cada distancia entre punto_restante y punto_seleccionado y lo sumamos a distancia_total_solucion (distancia_total_solucion <- distancia_total_solucion + matriz_dist_{punto_seleccionado, punto_restante})*
2. *Devolvemos la distancia total de esa solución (Devolvemos distancia_total_solucion)*

Pseudocódigos

Modelo Generacional



algoritmo generacional uniforme

variables locales

tipo vector de Cromosomas poblacion, padres, poblacion_intermedia, poblacion_mutada

tipo entero *contador* <- 0

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros m (número de elementos a seleccionar), n (número total de elementos)

inicio

1. Poblacion <- operador_generacion_poblacion(50 elementos)
2. Mientras no se realicen las 100000 evaluaciones de la función objetivo
 - a. Padres <- Operador de selección de padres (50)
 - b. Poblacion_intermedia <- operador de cruce uniforme
 - c. Reparación de poblacion intermedia
 - d. Poblacion_mutada <- operador de mutacion
 - e. Introducir(poblacion_mutada, mejor_elemento(poblacion))
 - f. Poblacion <- poblacion_mutada
3. Devolvemos mejor_elemento(poblacion)

algoritmo generacional posicional

variables locales

tipo vector de Cromosomas poblacion, padres, poblacion_intermedia, poblacion_mutada

tipo entero *contador* <- 0

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *m* (número de elementos a seleccionar), *n* (número total de elementos)

inicio

1. Poblacion <- operador_generacion_poblacion(50 elementos)
2. Mientras no se realicen las 100000 evaluaciones de la función objetivo
 - a. Padres <- Operador de selección de padres (50)
 - b. Poblacion_intermedia <- operador de cruce posicional
 - c. Poblacion_mutada <- operador de mutacion
 - d. Introducir(poblacion_mutada, mejor_elemento(poblacion))
 - e. Poblacion <- poblacion_mutada
3. Devolvemos mejor_elemento(poblacion)

algoritmo operador_generacion_poblacion

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *m* (número de elementos a seleccionar), *n* (número total de elementos),
&evaluaciones (evaluaciones totales de la función objetivo)

tipo vector de Cromosomas &poblacion (vector en el que se devolverá la población generada)

inicio

1. Vaciar(poblacion)
4. Para 50 iteraciones
 - a. Creo un cromosoma aleatorio crom1
 - b. Aumento evaluaciones (evaluaciones <- evaluaciones + 1)
 - c. Añadir(poblacion, crom1)

algoritmo operador seleccionar padres

variables como parámetros

tipo entero cantidad_padres (cantidad de padres que queremos generar)

tipo vector de Cromosomas poblacion (vector del que se sacaran los padres)

variables locales

vector de Cromosoma padres

tipo entero segundo_padre, primer_padre, contador <- 0

inicio

1. Mientras no se seleccionen la cantidad de padres deseada (contador < cantidad_padres)
 - a. Primer_padre <- numero_aleatorio[0, tamaño(poblacion)]
 - b. Segundo_padre <- (numero_aleatorio[0, tamaño(poblacion)] ≠ primer_padre)
 - c. Añadir(padres, mejor_elemento(primer_padre, segundo_padre))
2. Devolver padres

algoritmo operador cruce uniforme

variables como parámetros

tipo entero n (numero de elementos que tiene el vector genes de cada cromosoma)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de Cromosomas padres (vector de padres que se van a cruzar)

variables locales

vector de Cromosoma poblacion_intermedia

tipo double prob_cruce

tipo entero num_esperado_cruce, contador <- 0

inicio

1. Si hay 50 padres
 - a. Prob_cruce <- 0.7
2. Si hay 2 padres
 - a. Prob_cruce <- 1
3. Num_esperado_cruce <- prob_cruce * cantidad_padres/2
4. Mientras no se produzcan todos los hijos (contador < 2*num_esperado_cruces)
 - a. Para cada gen g hasta n
 - i. Si padre[contador].gen[g] ≠ hijo[contador+1].gen[g]
 1. Si numero aleatorio sale par
 - a. Hijo[contador+1].cambiar_gen(g)
 - ii. Si padre[contador+1].gen[g] ≠ hijo[contador].gen[g]
 1. Si numero aleatorio sale par
 - a. Hijo[contador].cambiar_gen(g)
 - iii. Contador <- contador + 2
5. Devolvemos poblacion_intermedia

algoritmo operador cruce posicional

variables como parámetros

tipo entero n (numero de elementos que tiene el vector genes de cada cromosoma)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de Cromosomas padres (vector de padres que se van a cruzar)

variables locales

vector de Cromosoma poblacion_intermedia

tipo double prob_cruce

tipo entero num_esperado_cruce, contador <- 0

tipo vector de enteros padre_auxiliar

inicio

6. Si hay 50 padres
 - a. Prob_cruce <- 0.7
7. Si hay 2 padres
 - a. Prob_cruce <- 1
8. Num_esperado_cruce <- prob_cruce * cantidad_padres/2
9. Mientras no se produzcan todos los hijos (contador < 2*num_esperado_cruces)
 - a. Para cada gen g hasta n
 - i. Si padre[contador+1].gen[g] ≠ hijo[contador].gen[g]
 1. Añadir(padre_auxiliar, padre[contador].gen[g])
 2. Hijo[contador].cambiar_gen(g, -1)
 3. Hijo[contador+1].cambiar_gen(g, -1)
 - b. Mezclar(padre_auxiliar)
 - c. Para cada gen g de Hijo[contador]
 - i. Si Hijo[contador].gen(g) es -1
 1. Hijo[contador].cambiar_gen(g, padre_auxiliar.siguiente)
 - d. Mezclar(padre_auxiliar)
 - e. Para cada gen g de Hijo[contador+1]
 - i. Si Hijo[contador+1].gen(g) es -1
 1. Hijo[contador+1].cambiar_gen(g, padre_auxiliar.siguiente)
 - f. Contador <- contador + 2
10. Devolvemos poblacion_intermedia

algoritmo operador mutacion

variables como parámetros

tipo entero n (numero de elementos que tiene el vector genes de cada cromosoma), m (cantidad de elementos seleccionados que debe tener cada solucion / 1's en cada vector de genes), &evaluaciones (contador de las veces que se ha llamado a la función objetivo)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de Cromosomas poblacion_intermedia (vector que vamos a mutar)

variables locales

vector de Cromosoma poblacion_mutada <- poblacion_intermedia

tipo entero contador <- 0, unos_pasados <- 0, num_esperado_mutacion <- 0.1/n, gen_aleatorio, cromosoma_aleatorio

tipo booleano busca_0, uno_cambiado

inicio

1. Num_esperado_mutacion = probab_mutacion * 50 * n
2. Mientras no se mute la cantidad deseada (contador < num_esperado_mutacion)
 - b. Cromosoma_aleatorio <- numero_aleatorio[0, tamaño(poblacion_mutada)]
 - c. gen_aleatorio <- numero_aleatorio[0, tamaño(poblacion_mutada)]
 - d. Para cada gen g de poblacion_intermedia[cromosoma_aleatorio] y mientras (busca_0 && NO uno_cambiado)
 - i. Si busca_0 y poblacion_mutada[cromosoma_aleatorio].gen(g) = 0
 1. poblacion_mutada[cromosoma_aleatorio].cambiar_gen(g, 0)
 2. busca_0 <- falso
 - ii. Si NO uno_cambiado y poblacion_mutada[cromosoma_aleatorio].gen(g) = 1
 1. Unos_pasados <- unos_pasados + 1
 2. Si he pasado el mismo numero de unos que el que busco (gen_aleatorio = unos_pasados)
 - a. poblacion_mutada[cromosoma_aleatorio].cambiar_gen(g, 1)
 - b. uno_cambiado <- verdad
 - e. contador <- contador + 1
3. Para cada cromosoma crom en poblacion_mutada
 - a. Si crom.reevaluar
 - i. Crom.reevaluar_coste
 - ii. Evaluaciones <- evaluaciones + 1
4. Devolver poblacion_mutada

algoritmo operador reparación cruce uniforme

variables como parámetros

tipo entero m (cantidad de 1's que debe tener el vector genes del cromosoma)

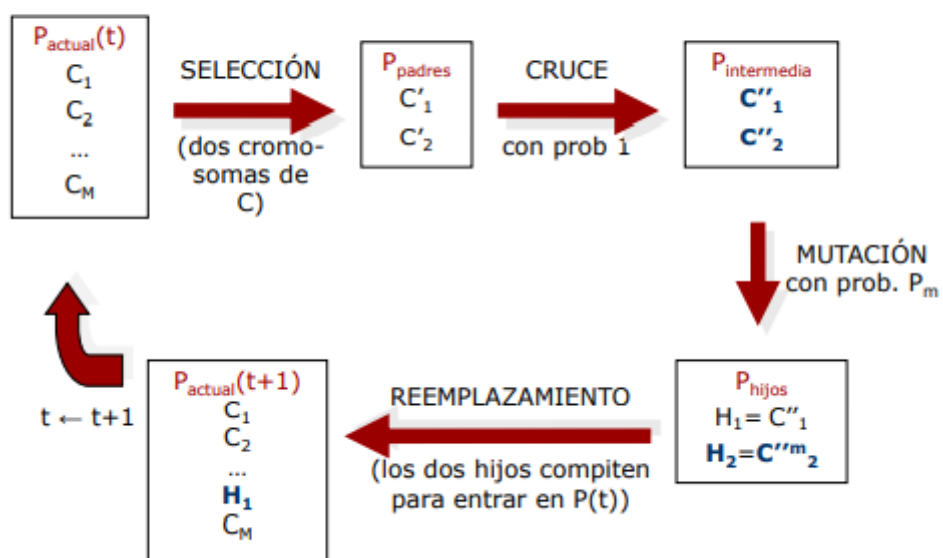
tipo matriz de double $matriz_dist$ (distancia entre todos los puntos)

tipo Cromosomas $cromosoma_reparar$ (cromosoma que arreglamos)

inicio

3. Si sobran 1 ($cantidad_unos > m$)
 - a. Mientras haya mas unos de los que queremos
 - i. $Mejor_elemento \leftarrow$ Buscamos el elemento dentro del vector de genes que mayor aporte tiene
 - ii. $Eliminar(cromosoma_arreglar, mejor_elemento)$
4. Si faltan 1 ($cantidad_unos < m$)
 - a. Mientras haya menos unos de los que queremos
 - i. $Mejor_elemento \leftarrow$ Buscamos en el vecindario de elementos no seleccionados el que mayor aporte tendria estando dentro
 - ii. $Añadir(cromosoma_arreglar, mejor_elemento)$

Modelo Estacionario



algoritmo estacionario uniforme

variables locales

tipo vector de Cromosomas poblacion, padres, poblacion_intermedia, poblacion_mutada

tipo entero *contador* <- 0

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *m* (número de elementos a seleccionar), *n* (número total de elementos)

inicio

1. Poblacion <- operador_generacion_poblacion(50 elementos)
2. Mientras no se realicen las 100000 evaluaciones de la función objetivo
 - a. Padres <- Operador de selección de padres (2)
 - b. Poblacion_intermedia <- operador de cruce uniforme
 - c. Reparación de poblacion intermedia
 - d. Poblacion_mutada <- operador de mutacion
 - e. Introducir(poblacion, mejor_elemento(poblacion_mutada))
3. Devolvemos mejor_elemento(poblacion)

algoritmo estacionario posicional

variables locales

tipo vector de Cromosomas poblacion, padres, poblacion_intermedia, poblacion_mutada

tipo entero *contador* <- 0

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *m* (número de elementos a seleccionar), *n* (número total de elementos)

inicio

1. Poblacion <- operador_generacion_poblacion(50 elementos)
4. Mientras no se realicen las 100000 evaluaciones de la función objetivo
 - a. Padres <- Operador de selección de padres (2)
 - b. Poblacion_intermedia <- operador de cruce posicional
 - c. Poblacion_mutada <- operador de mutacion
 - d. Introducir(poblacion, mejor_elemento(poblacion_mutada))
3. Devolvemos mejor_elemento(poblacion)

algoritmo memetico (10, 1.0)

variables locales

tipo vector de Cromosomas poblacion, padres, poblacion_intermedia, poblacion_mutada

tipo entero *contador* <- 0, *generación* <- 1

tipo vector de enteros S, poblacion_decimal

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *m* (número de elementos a seleccionar), *n* (número total de elementos)

inicio

2. Poblacion <- operador_generacion_poblacion(50 elementos)
- 5 Mientras no se realicen las 100000 evaluaciones de la función objetivo
 - a. Padres <- Operador de selección de padres (2)
 - b. Poblacion_intermedia <- operador de cruce uniforme
 - c. Reparación de poblacion intermedia
 - d. Poblacion_mutada <- operador de mutacion
 - e. Si van 10 generaciones
 - i. Para cada cromosoma crom de la poblacion
 1. S <- Obtengo_vecinos
 2. Poblacion_decimal <- SolucionBL(poblacion_decimal, S)
 3. Crom <- introducir_genes(Poblacion_decimal)
 - f. Introducir(poblacion_mutada, mejor_elemento(poblacion))
 - g. Poblacion <- poblacion_mutada
- 4 Devolvemos mejor_elemento(poblacion)

algoritmo memetico (10, 0.1)

variables locales

tipo vector de Cromosomas poblacion, padres, poblacion_intermedia, poblacion_mutada

tipo entero *contador* <- 0, *generación* <- 1

tipo vector de enteros S, poblacion_decimal

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *m* (número de elementos a seleccionar), *n* (número total de elementos)

inicio

3. Poblacion <- operador_generacion_poblacion(50 elementos)
- 6 Mientras no se realicen las 100000 evaluaciones de la función objetivo
 - a. Padres <- Operador de selección de padres (2)
 - b. Poblacion_intermedia <- operador de cruce uniforme
 - c. Reparación de poblacion intermedia
 - d. Poblacion_mutada <- operador de mutacion
 - e. Si van 10 generaciones
 - i. Para cada cromosoma aleatorio crom del 10%poblacion
 1. S <- Obtengo_vecinos
 2. Poblacion_decimal <- SolucionBL(poblacion_decimal, S)
 3. Crom <- introducir_genes(Poblacion_decimal)
 - f. Introducir(poblacion_mutada, mejor_elemento(poblacion))
 - g. Poblacion <- poblacion_mutada
- 5 Devolvemos mejor_elemento(poblacion)

algoritmo memetico (10, 0.1mej)

variables locales

tipo vector de Cromosomas poblacion, padres, poblacion_intermedia, poblacion_mutada

tipo entero *contador* <- 0, *generación* <- 1

tipo vector de enteros S, poblacion_decimal, mejores

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *m* (número de elementos a seleccionar), *n* (número total de elementos)

inicio

4. Poblacion <- operador_generacion_poblacion(50 elementos)
- 7 Mientras no se realicen las 100000 evaluaciones de la función objetivo
 - a. Padres <- Operador de selección de padres (2)
 - b. Poblacion_intermedia <- operador de cruce uniforme
 - c. Reparación de poblacion intermedia
 - d. Poblacion_mutada <- operador de mutacion
 - e. Si van 10 generaciones
 - i. Para cada cromosoma crom del 10%poblacion_mutada
 1. Para todos los cromosomas crom2 de poblacion_mutada
 - a. Si crom2 es la mejor hasta ahora
 - i. Si crom2 no pertenece a mejores
 1. Guardamos crom2 y su coste
 2. Añadir(mejores, mejor crom2 que no estuviese en mejores)
 - ii. Para cada elemento mej de mejores
 1. S <- Obtengo_vecinos(poblacion_mutada[mej])
 2. Poblacion_decimal <- SolucionBL(poblacion_decimal, S)
 3. Poblacion_mutada[mej] <- introducir_genes(Poblacion_decimal)
 - f. Introducir(poblacion_mutada, mejor_elemento(poblacion))
 - g. Poblacion <- poblacion_mutada
 - 6 Devolvemos mejor_elemento(poblacion)

algoritmo BL

variables locales

tipo double *total_distancia_punto_actual*, *aporte_mas_bajo*

tipo entero *índice_aporte_mas_bajo*, *simula_i*, *simula_j*

tipo booleano *seguir*

variables como parámetros

tipo entero *m* (número de elementos a seleccionar), *n* (número total de elementos), *contador* (número de veces que ha sido llamada la función recursivamente para no pasarnos del máximo)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *Solucion* (vector con los elementos de la solución), *S* (vector con los elementos no seleccionados)

inicio

1. *simula_i* <- *simula_j* <- *índice_aporte_mas_bajo* <- 0, *aporte_mas_bajo* <- 1000000, *seguir* <- true
(Inicializamos las variables)

2. Bucle que recorra todos los elementos de la solución

2.1 *total_distancia_punto_actual* <- sumatoria de la distancia del punto actual al resto de la solución

2.2 Si *total_distancia_punto_actual* es menor que *aporte_mas_bajo* (Si el punto actual aporta menos coste a la solución que el elemento que menos aportaba)

2.1.2 *aporte_mas_bajo* <- *total_distancia_punto_actual* (Actualizamos la menor distancia que hemos visto hasta ahora)

2.1.3 *índice_aporte_mas_bajo* <- *punto actual* (guardamos el índice del elemento)

3. fin del bucle

4. Bucle que recorre los elementos no seleccionados mientras 'seguir' sea verdadero (con esto buscamos de los no seleccionados cual es el que mejora la solución si sustituye al elemento que menos aporta ya calculado) o hayamos evaluado 400 vecinos

4.1 Calculamos la distancia del elemento no seleccionado actual a todos los del conjunto de solución excepto a aquel que queremos quitar

4.2 Si este elemento aporta más que el que menos aporta de la solución

4.2.1 *seguir* <- false (para que no busque más vecinos. No queremos el mejor, queremos el primero)

4.2.2 *Solucion* <- *eliminar(índice_aporte_mas_bajo)*

4.2.3 *Solucion* <- *añadir(elemento actual)*

4.2.4 *S* <- *eliminar(elemento actual)*

5. fin del bucle

6. Si seguir es falso (se ha encontrado ningún elemento que mejore la solución) y contador es menor que 99999

6.1 llamamos recursivamente al algoritmo BL

7.2 fin del bucle

Fin

Experimento y análisis de los resultados

Descripción de los casos del problema empleados

Existen casos del problema que han sido estudiados con anterioridad para comprobar el funcionamiento de los algoritmos de resolución. A pesar de existir diferentes grupos de casos, nos centraremos en los utilizados en esta práctica, que son los casos MDG.

Los archivos de este tipo contienen en la primera línea dos valores, el primero de ellos que indica la cantidad de puntos totales con los que cuenta el problema y el segundo la cantidad de elementos que vamos a escoger para nuestra solución. A partir de ahí, cada una de las $n(n-1)/2$ líneas contiene 3 valores, los dos primeros enteros que marcan los índices de los puntos entre los que existe la distancia representada con el tercer valor, de tipo double, de la fila.

En nuestro caso, utilizamos 30 ficheros diferentes de los cuales 10 son del tipo MDG-a, los cuales tienen matrices de 500x500, $m = 50$ y distancias aleatorias en $[0,10]$; 10 son del tipo MDG-b con matrices de 2000x2000, $m = 200$ y distancias aleatorias en $[0,1000]$; y, por último, 10 son del tipo MDG-c con matrices de 3000x3000, de las cuales 2 tienen $m=300$, 3 tienen $m=400$, 3 con $m=500$ y 2 con $m=600$.

En el caso de la semilla, se utilizan los ficheros *random.cpp* y *random.h* aportados por los profesores y el valor con el que se inicializa es pasado como parámetro. En nuestro caso, las tablas se han rellenado usando como semilla el entero con valor 5.

Los parámetros considerados en los 7 algoritmos (**modelo generacional uniforme, modelo estacionario uniforme, modelo generacional posicional, modelo estacionario posicional, algoritmo memético (10, 1.0), algoritmo memético (10, 0.1) y algoritmo memético (10, 0.1mej)**) han sido los mismos:

- m (entero) -> número de 1's que debe tener 'genes' de cada cromosoma de la población (cantidad de elementos que se escogen por solución).
- n (entero): número de elementos en total que tiene nuestro vector 'genes' en cada cromosoma de la población.
- distancias (matriz de punteros de double): Matriz que contiene todas las distancias.

Resultados obtenidos

AGG - uniforme			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	7620,1200	2,73	3746,00
MDG-a_2_n500_m50	7549,0400	2,86	3781,00
MDG-a_3_n500_m50	7575,6700	2,37	3362,00
MDG-a_4_n500_m50	7598,6700	2,21	3743,00
MDG-a_5_n500_m50	7609,9700	1,87	3497,00
MDG-a_6_n500_m50	7645,0900	1,65	3727,00
MDG-a_7_n500_m50	7667,2300	1,34	3674,00
MDG-a_8_n500_m50	7664,3000	1,12	3589,00
MDG-a_9_n500_m50	7626,9200	1,84	3727,00
MDG-a_10_n500_m50	7739,0800	0,53	3674,00
MDG-b_21_n2000_m200	11209200,0000	0,80	40343,00
MDG-b_22_n2000_m200	11193600,0000	0,83	41496,00
MDG-b_23_n2000_m200	11181400,0000	1,05	39543,00
MDG-b_24_n2000_m200	11151000,0000	1,24	39938,00
MDG-b_25_n2000_m200	11167200,0000	1,14	38980,00
MDG-b_26_n2000_m200	11163900,0000	1,14	41240,00
MDG-b_27_n2000_m200	11196000,0000	0,97	40473,00
MDG-b_28_n2000_m200	11183100,0000	0,86	39187,00
MDG-b_29_n2000_m200	11193700,0000	0,92	38599,00
MDG-b_30_n2000_m200	11189600,0000	0,95	39823,00
MDG-c_1_n3000_m300	24753800	0,52	107661,00
MDG-c_2_n3000_m300	24679500	0,91	95334,00
MDG-c_8_n3000_m400	43170900	0,61	119533,00
MDG-c_9_n3000_m400	43138700	0,69	145490,00
MDG-c_10_n3000_m400	43150500	0,75	119263,00
MDG-c_13_n3000_m500	66602300	0,61	147314,00
MDG-c_14_n3000_m500	66627600	0,53	154360,00
MDG-c_15_n3000_m500	66551400	0,66	155002,00
MDG-c_19_n3000_m600	95183700	0,47	193576,00
MDG-c_20_n3000_m600	95078700	0,59	182490,00

AGG - posicional			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	6989,3400	10,78	2540,00
MDG-a_2_n500_m50	6866,6300	11,65	2604,00
MDG-a_3_n500_m50	6929,3600	10,70	2722,00
MDG-a_4_n500_m50	6892,8100	11,29	3565,00
MDG-a_5_n500_m50	6964,7400	10,19	2666,00
MDG-a_6_n500_m50	6896,5500	11,28	2834,00
MDG-a_7_n500_m50	6870,8600	11,59	2802,00
MDG-a_8_n500_m50	6831,0700	11,87	2650,00
MDG-a_9_n500_m50	6860,5900	11,70	2498,00
MDG-a_10_n500_m50	6980,9700	10,27	2883,00
MDG-b_21_n2000_m200	10431200,0000	7,69	24170,00
MDG-b_22_n2000_m200	10453500,0000	7,38	24095,00
MDG-b_23_n2000_m200	10391800,0000	8,04	20115,00
MDG-b_24_n2000_m200	10460000,0000	7,36	25369,00
MDG-b_25_n2000_m200	10447100,0000	7,52	23459,00
MDG-b_26_n2000_m200	10403900,0000	7,87	21446,00
MDG-b_27_n2000_m200	10413900,0000	7,89	23852,00
MDG-b_28_n2000_m200	10361100,0000	8,15	24421,00
MDG-b_29_n2000_m200	10383400,0000	8,09	23853,00
MDG-b_30_n2000_m200	10389200,0000	8,03	22292,00
MDG-c_1_n3000_m300	23183700	6,83	62620,00
MDG-c_2_n3000_m300	23112500	7,20	56837,00
MDG-c_8_n3000_m400	40945500	5,74	65522,00
MDG-c_9_n3000_m400	41045800	5,51	76547,00
MDG-c_10_n3000_m400	40920000	5,88	66639,00
MDG-c_13_n3000_m500	63614100	5,07	77556,00
MDG-c_14_n3000_m500	63743400	4,83	77327,00
MDG-c_15_n3000_m500	63768200	4,81	78171,00
MDG-c_19_n3000_m600	91504400	4,32	87752,00
MDG-c_20_n3000_m600	91388400	4,45	89444,00

AGE - uniforme			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	7182,8900	8,31	8787,00
MDG-a_2_n500_m50	7090,9800	8,76	8782,00
MDG-a_3_n500_m50	7209,3300	7,09	8773,00
MDG-a_4_n500_m50	7106,9500	8,54	10476,00
MDG-a_5_n500_m50	7160,4000	7,67	8249,00
MDG-a_6_n500_m50	7104,4400	8,61	8612,00
MDG-a_7_n500_m50	7178,3200	7,64	8949,00
MDG-a_8_n500_m50	7128,3400	8,03	8442,00
MDG-a_9_n500_m50	7161,9300	7,83	8411,00
MDG-a_10_n500_m50	7101,1600	8,73	8351,00
MDG-b_21_n2000_m200	10801400,0000	4,41	118423,00
MDG-b_22_n2000_m200	10811200,0000	4,21	117130,00
MDG-b_23_n2000_m200	10761300,0000	4,77	117914,00
MDG-b_24_n2000_m200	10749900,0000	4,79	119888,00
MDG-b_25_n2000_m200	10772900,0000	4,63	116687,00
MDG-b_26_n2000_m200	10833700,0000	4,06	118195,00
MDG-b_27_n2000_m200	10779000,0000	4,66	114650,00
MDG-b_28_n2000_m200	10723400,0000	4,93	115064,00
MDG-b_29_n2000_m200	10788200,0000	4,51	116811,00
MDG-b_30_n2000_m200	10905800,0000	3,46	123529,00
MDG-c_1_n3000_m300	24023400	3,46	316298,00
MDG-c_2_n3000_m300	23889000	4,08	277216,00
MDG-c_8_n3000_m400	42172000	2,91	338316,00
MDG-c_9_n3000_m400	42279000	2,67	362211,00
MDG-c_10_n3000_m400	42046000	3,29	338074,00
MDG-c_13_n3000_m500	65220500	2,68	400739,00
MDG-c_14_n3000_m500	65195000	2,66	409635,00
MDG-c_15_n3000_m500	65143400	2,76	407657,00
MDG-c_19_n3000_m600	93658700	2,07	468826,00
MDG-c_20_n3000_m600	93647100	2,09	464217,00

AGE - posicional			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	6597,2500	15,79	4042,00
MDG-a_2_n500_m50	6647,1000	14,47	5027,00
MDG-a_3_n500_m50	6635,5300	14,48	4597,00
MDG-a_4_n500_m50	6571,9000	15,42	3148,00
MDG-a_5_n500_m50	6671,8700	13,97	4205,00
MDG-a_6_n500_m50	6633,4100	14,67	4690,00
MDG-a_7_n500_m50	6729,8600	13,41	4058,00
MDG-a_8_n500_m50	6851,2000	11,61	4787,00
MDG-a_9_n500_m50	6733,4900	13,34	4345,00
MDG-a_10_n500_m50	6653,4100	14,48	3866,00
MDG-b_21_n2000_m200	10360400,0000	8,31	42040,00
MDG-b_22_n2000_m200	10329000,0000	8,49	45186,00
MDG-b_23_n2000_m200	10256600,0000	9,23	48761,00
MDG-b_24_n2000_m200	10318500,0000	8,61	43760,00
MDG-b_25_n2000_m200	10331800,0000	8,54	50198,00
MDG-b_26_n2000_m200	10362100,0000	8,24	45989,00
MDG-b_27_n2000_m200	10351000,0000	8,44	45840,00
MDG-b_28_n2000_m200	10296300,0000	8,72	49889,00
MDG-b_29_n2000_m200	10332500,0000	8,54	50006,00
MDG-b_30_n2000_m200	10293000,0000	8,88	47335,00
MDG-c_1_n3000_m300	23167600	6,90	121053,00
MDG-c_2_n3000_m300	23076700	7,34	90159,00
MDG-c_8_n3000_m400	40994300	5,62	123864,00
MDG-c_9_n3000_m400	40794500	6,09	123317,00
MDG-c_10_n3000_m400	40803600	6,15	134722,00
MDG-c_13_n3000_m500	63683100	4,97	129882,00
MDG-c_14_n3000_m500	63697600	4,90	150305,00
MDG-c_15_n3000_m500	63665700	4,97	144428,00
MDG-c_19_n3000_m600	91610700	4,21	154680,00
MDG-c_20_n3000_m600	91538100	4,29	154991,00

AM - (10,1.0)			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n50	7778,1800	0,71	11227,00
MDG-a_2_n50	7683,5300	1,13	11090,00
MDG-a_3_n50	7721,1000	0,49	11027,00
MDG-a_4_n50	7755,9600	0,18	11328,00
MDG-a_5_n50	7697,3700	0,75	10812,00
MDG-a_6_n50	7673,9900	1,28	11203,00
MDG-a_7_n50	7735,5300	0,47	11145,00
MDG-a_8_n50	7693,5300	0,74	10683,00
MDG-a_9_n50	7754,2700	0,20	10925,00
MDG-a_10_n50	7775,6400	0,06	11267,00
MDG-b_21_n2	11185600,0000	1,01	90860,00
MDG-b_22_n2	11192200,0000	0,84	89109,00
MDG-b_23_n2	11195400,0000	0,93	88496,00
MDG-b_24_n2	11142600,0000	1,31	90218,00
MDG-b_25_n2	11205300,0000	0,80	86318,00
MDG-b_26_n2	11145700,0000	1,30	90071,00
MDG-b_27_n2	11184600,0000	1,07	90002,00
MDG-b_28_n2	11159900,0000	1,06	90334,00
MDG-b_29_n2	11190400,0000	0,95	90524,00
MDG-b_30_n2	11206900,0000	0,79	88128,00
MDG-c_1_n30	24765800	0,48	223476,00
MDG-c_2_n30	24749100	0,63	193829,00
MDG-c_8_n30	43210800	0,52	292652,00
MDG-c_9_n30	43218300	0,51	285870,00
MDG-c_10_n30	43108900	0,84	288872,00
MDG-c_13_n30	66683500	0,49	396866,00
MDG-c_14_n30	66726800	0,38	410194,00
MDG-c_15_n30	66712600	0,42	399757,00
MDG-c_19_n30	95301800	0,35	526914,00
MDG-c_20_n30	95181100	0,48	544010,00

AM - (10,0.1)			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n50	7738,6400	1,22	4356,00
MDG-a_2_n50	7680,5800	1,17	4174,00
MDG-a_3_n50	7716,6100	0,55	4144,00
MDG-a_4_n50	7664,0600	1,37	4271,00
MDG-a_5_n50	7649,2900	1,37	3999,00
MDG-a_6_n50	7699,6100	0,95	4331,00
MDG-a_7_n50	7702,6400	0,89	4242,00
MDG-a_8_n50	7715,2100	0,46	4026,00
MDG-a_9_n50	7703,4000	0,86	4206,00
MDG-a_10_n50	7752,2200	0,36	4203,00
MDG-b_21_n2	11186200,0000	1,01	39414,00
MDG-b_22_n2	11197400,0000	0,79	39453,00
MDG-b_23_n2	11199700,0000	0,89	38013,00
MDG-b_24_n2	11191900,0000	0,88	39472,00
MDG-b_25_n2	11142700,0000	1,36	39682,00
MDG-b_26_n2	11162600,0000	1,15	38595,00
MDG-b_27_n2	11173700,0000	1,17	40861,00
MDG-b_28_n2	11226200,0000	0,48	40516,00
MDG-b_29_n2	11215700,0000	0,72	39801,00
MDG-b_30_n2	11177500,0000	1,05	37900,00
MDG-c_1_n30	24763100	0,49	97069,00
MDG-c_2_n30	24679900	0,91	97583,00
MDG-c_8_n30	43210600	0,52	123818,00
MDG-c_9_n30	43170500	0,62	120316,00
MDG-c_10_n30	43116100	0,83	122508,00
MDG-c_13_n30	66726100	0,43	163250,00
MDG-c_14_n30	66699600	0,42	161820,00
MDG-c_15_n30	66697800	0,44	152697,00
MDG-c_19_n30	95365100	0,28	204533,00
MDG-c_20_n30	95308700	0,35	207949,00

AM - (10,0.1mej)			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	7687,6500	1,87	4120,00
MDG-a_2_n500_m50	7707,2400	0,83	4156,00
MDG-a_3_n500_m50	7719,6600	0,51	3914,00
MDG-a_4_n500_m50	7600,0900	2,19	3985,00
MDG-a_5_n500_m50	7613,4400	1,83	4190,00
MDG-a_6_n500_m50	7619,0900	1,99	4155,00
MDG-a_7_n500_m50	7713,7500	0,75	4039,00
MDG-a_8_n500_m50	7585,5000	2,13	3869,00
MDG-a_9_n500_m50	7663,6300	1,37	3944,00
MDG-a_10_n500_m50	7717,6700	0,81	4146,00
MDG-b_21_n2000_m200	11190500,0000	0,97	46921,00
MDG-b_22_n2000_m200	11190800,0000	0,85	40752,00
MDG-b_23_n2000_m200	11184000,0000	1,03	38663,00
MDG-b_24_n2000_m200	11167100,0000	1,10	40292,00
MDG-b_25_n2000_m200	11184900,0000	0,98	41774,00
MDG-b_26_n2000_m200	11157300,0000	1,20	37876,00
MDG-b_27_n2000_m200	11216200,0000	0,79	39543,00
MDG-b_28_n2000_m200	11205800,0000	0,66	43554,00
MDG-b_29_n2000_m200	11178700,0000	1,05	36577,00
MDG-b_30_n2000_m200	11192100,0000	0,92	40750,00
MDG-c_1_n3000_m300	24713900	0,68	93742,00
MDG-c_2_n3000_m300	24713000	0,77	92343,00
MDG-c_8_n3000_m400	43203400	0,54	120768,00
MDG-c_9_n3000_m400	43174900	0,61	123135,00
MDG-c_10_n3000_m400	43179800	0,68	127314,00
MDG-c_13_n3000_m500	66722200	0,44	151238,00
MDG-c_14_n3000_m500	66749400	0,34	158649,00
MDG-c_15_n3000_m500	66762100	0,34	157083,00
MDG-c_19_n3000_m600	95228000	0,42	187153,00
MDG-c_20_n3000_m600	95229700	0,43	189537,00

Algoritmo	Desv	Tiempo (ms)
AGG – uniforme	1.16	61872.17
AGG – posición	8.13	33308.37
AGE – uniforme	5.14	168310.4
AGE – posición	9.44	61305.67
AM – (10, 1.0)	0.71	152240.23
AM – (10, 0.1)	0.8	62906.73
AM – (10, 0.1mej)	0.97	61606.07

Algoritmo	Desv	Tiempo
Greedy	9,22	717,59 ms
BL	2,11	933,26 ms

Análisis de resultados

Para el análisis de resultados comenzaremos comparando entre ellos cada uno de los 7 algoritmos desarrollados en esta práctica para, más tarde, analizar el desempeño de estos con respecto de Greedy y BL implementados en la práctica anterior.

En lo referido a esta práctica, no hay más que ver la tabla de comparación de desviaciones y tiempos para ver que el que mejores resultados da, por lo general, es el algoritmo memético (10, 0.1), que es aquel en el que cada 10 generaciones se aplica BL a todos los cromosomas de la población. Esto podía ser previsible en el sentido de que los meméticos parten del algoritmo que mejores resultados daba de entre los cuatro primeros y, tras esto, se aplica otro algoritmo que mejora cada elemento de la población, por lo que se favorece continuamente que los valores obtenidos vayan mejorando con cada generación. Además de ser el mejor en coste, está acompañado de los buenos resultados obtenidos de los otros dos algoritmos AM que, siguiendo el mismo razonamiento expuesto, también debían obtener un coste más cercano al óptimo que cualquiera de las cuatro primeras opciones.

Algo que puede sonar incoherente es el hecho de que el algoritmo memético que aplica BL a los 10 mejores cromosomas de la población tras estas 10 generaciones sea el que obtenga peores resultados. Un razonamiento comprensible sería pensar que si además de aplicar BL para mejorar ciertos elementos de la población, este se aplica a los mejores, va a haber

cromosomas con un coste extremadamente bueno y esto resultaría en un coste poco mejorable, pero en realidad el efecto es el contrario, pues el algoritmo BL ya tenía ciertas limitaciones en la práctica anterior (que se introduzca un elemento del vecindario que sea bueno, pero no el mejor) y en esta se le incluye otra que es la evaluación de un máximo de 400 vecinos, por lo que este algoritmo de BL no va a exprimir las mejoras todo lo que podría. Si tras esto, este se aplica a soluciones que de entrada ya son buenos (o al menos mejores que el resto) el margen de mejora es mínimo, por ello entre mejorar toda una población (que favorece los cruces que siguen a esa mejora) y mejorar solo algunos buenos hace que en general, generación tras generación, la población total sufra pocos beneficios.

Una vez analizados los meméticos, observemos al detalle el comportamiento de los generacionales y los estacionarios. Las cuatro soluciones tienen dos grandes diferencias combinadas entre ellos, que son la selección de 50 padres o 2, y la aplicación de un cruce uniforme o posicional. Cada característica tiene un efecto diferente en el conjunto de soluciones que observaremos por separado.

De entrada, vemos que los dos mejores en coste son aquellos que utilizan un cruce uniforme frente a aquellos que lo hacen de forma posicional. Esta primera gran diferencia es debida al operador de reparación tras el cruce de padres. Como ya sabemos, este cruce consiste en la selección de un determinado número de padres que se cruzan con una probabilidad y, tras esto, se genera una población de hijos con el riesgo de contener una cantidad de 1's en su vector de genes que sea incorrecto. Este error se soluciona añadiendo (en el caso de que le falten elementos seleccionados a la solución) los vecinos que más aportarían, lo cual tiene alguna semejanza con BL. Viéndolo con más detalle, si nos quedamos cortos con el número de 1's que tiene nuestro vector de genes, este se rellena con aquellos elementos que maximizan la diversidad al máximo; sin embargo, el cruce posicional no permite esta opción. Evitar la reparación tiene sus ventajas en cuanto a tiempo, lo cual veremos más adelante, pero anula totalmente la opción de meter elementos nuevos y mejores, pues la cantidad de elementos seleccionados por cada solución de la población no tiene margen de error. Esta diversidad que aporta la reparación permite evolucionar más rápido a los algoritmos que hacen uso de un cruce posicional.

Por otra parte, los generacionales son mejores en coste que los dos estacionarios, respectivamente. Es la otra gran diferencia que podemos analizar, y esto es debido a que en los generacionales la población evoluciona completamente manteniendo el mejor elemento de la población predecesora mientras que los estacionarios mantienen la población idéntica de una generación a otra, con la única diferencia de añadir el mejor hijo de los dos obtenidos de la selección de dos padres aleatorios (con un torneo). Esto hace que el primer caso tenga muchas evoluciones simultáneas y el segundo solo evolucione un elemento por generación, lo cual, obviamente, resulta en una mejora lenta que en 100000 evoluciones no alcanza valores tan buenos como los generacionales.

Ahora, teniendo en cuenta el tiempo de ejecución, el mejor algoritmo es el modelo estacionario posicional. El motivo de esto puede verse explicando el por qué el resto de algoritmos es más lento. Empezamos por los modelos con cruce uniforme, los cuales cuentan con un operador de reparación como mencionamos anteriormente que hace que pierdan bastante tiempo en reparaciones. Tras esto, sabiendo que los tres algoritmos meméticos parten del generacional uniforme y aplican BL adicionalmente a una parte total o parcial de la

población, era de prever que el tiempo sería igual o mayor que el de este. Entre ellos, la principal diferencia de tiempo es ocasionada por la aplicación de BL. En el caso del AM – (10, 1.0), la aplicación de la Búsqueda Local se hace cada 10 generaciones a la población completa, mientras que en los otros dos solamente se realiza a 10 cromosomas del conjunto. Esto hace que el tiempo del primero mencionado sea más del doble que el resto de meméticos. Por otra parte, en la práctica anterior vimos que BL obtenía de media menos de 1 segundo de ejecución y, con la limitación de la cantidad de vecinos máximos a recorrer y la supresión del cálculo del coste (que ahora se realiza en otra parte del programa) el tiempo es bastante menor a un segundo, lo cual hace que, la aplicación del algoritmo a 10 elementos cada 10 generaciones únicamente, no marque mucha diferencia con el generacional uniforme del que parten. Por último, la diferencia del estacionario posicional con el generacional posicional reside en que, por cada generación, se cruzan solamente 2 padres en lugar de 50, lo cual hace que el número de operaciones sea bastante menor y obtenga mejores tiempos.

Tras esto, comparando estos siete algoritmos con Greedy y BL de la práctica anterior, en cuanto a resultados era de esperar que casi todos son mejores que Greedy pues este cogía una solución buena, pero no la óptima. El único caso en el que los resultados son peores es en el modelo estacionario posicional, que cuenta con las peores versiones de cada variante de los cuatro primeros algoritmos. No solo cambia un único elemento por generación, ralentizando la evolución de la población, sino que además no cuenta con la reparación que introduce mejores elementos posibles. El resto de algoritmos son mejores y tanto los meméticos como el generacional uniforme son mejores que BL. Esto se debe a que generacional uniforme cuenta con la mejor de ambas variantes, ya que evoluciona la población completa por cada generación y cuenta con una reparación que produce en cada gen mejoras notables. En el caso de los meméticos, el resultado es mejor ya que hacen uso de BL sobre poblaciones que evolucionan, por lo que como muy bajo, el resultado iba a ser igual que BL, pero contando además con cruces y mutaciones.

En general, se esperaba que los resultados fuesen mejor en coste y peor en tiempo. En coste porque es como realizar 50 experimentos con cada algoritmo, y evolucionar estos y terminar cogiendo el mejor (demasiados beneficios en esta obtención de soluciones) y en tiempo porque la generación y repetición de procesos es costosa y es prácticamente multiplicar el tiempo de los algoritmos anteriores por cada cromosoma que se genere y evalúe.

Bibliografía

[Algoritmos Genéticos I: Conceptos Básicos \(ugr.es\)](#)

[temageneticos.dvi \(ehu.es\)](#)

[9.9 Algoritmos Meméticos \(inaoep.mx\)](#)

[Algoritmo memético - Wikipedia, la enciclopedia libre](#)

[Tema09-AlgoritmosMemeticos-12-13.pdf \(ugr.es\)](#)