UNIVERSIDAD DE GRANADA
E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

Práctica 2.a: Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Asignación Cuadrática

Álvaro Fernández García

DNI: 75572750V
3º Curso
Grupo 3: Miércoles 17:30 - 19:30
alvaro89@correo.ugr.es

Granada, curso 2017-2018

Índice

1. Formulación del Problema **2**

2. Consideraciones generales **3**

2.1. Representación de la solución 3

2.2. Representación de los Flujos y Distancias 3

2.3. Función objetivo 3

2.4. Operador de selección 4

2.5. Operador de mutación 5

2.6. Generación aleatoria de la población inicial 5

2.7. Operador de cruce basado en posiciones 5

2.8. Operador de cruce OX 6

3. Algoritmo Genético Generacional (AGG) **7**

4. Algoritmo Genético Estacionario (AGE) **8**

5. Algoritmos Meméticos **9**

5.1. Algoritmo Memético basado en probabilidades 11

5.2. Algoritmo Memético basado en la optimización de los mejores 11

6. Estructuración y Realización de la práctica **12**

7. Experimentos y análisis de resultados **14**

7.1. Tablas de resultados 14

7.2. Resultados Globales en el QAP 16

7.3. Conclusiones 16

1. Formulación del Problema

El problema de asignación cuadrática (en inglés, quadratic assignment problem, QAP) es uno de los problemas de optimización combinatoria más conocidos. Fue planteado por Koopmans y Beckmann en 1957 como un modelo matemático para un conjunto de actividades económicas indivisibles.

En él se dispone de n unidades y n localizaciones en las que situarlas, por lo que el problema consiste en encontrar la asignación óptima de cada unidad a una localización. La nomenclatura "cuadrático" proviene de la función objetivo que mide la bondad de una asignación, la cual considera el producto de dos términos, la distancia entre cada par de localizaciones y el flujo que circula entre cada par de unidades. El QAP se puede formular como:

$$\underset{\pi \in \prod_n}{min} \left(\sum_{i=1}^{N} \sum_{j=1}^{N} f_{ij} \cdot d_{\pi(i)\pi(j)} \right)$$

donde:

- π es una solución al problema que consiste en una permutación que representa la asignación de la unidad i a la localización $\pi(i)$.
- f_{ij} es el flujo que circula entre la unidad i y la j.
- d_{kl} es la distancia existente entre la localización k y la l.

Se trata de un problema NP-completo. Incluso la resolución de problemas pequeños de tamaño superior a 25 se considera una tarea computacionalmente muy costosa.

Está asociado a las siguientes restricciones:

- No se puede asignar una misma unidad a dos localizaciones distintas.
- No se puede asignar más de una unidad a una misma localización.

Su espacio de búsqueda es de $n!$, donde n es el número de unidades y localizaciones.

En los siguientes ejemplos de aplicaciones se puede observar que resolver este problema para un gran número de instancias es de vital importancia, y a la vez que tratar de resolver el problema mediante técnicas completas puede resultar infactible por el alto número de instancias.

- Diseño de centros comerciales donde se quiere que el público recorra la menor cantidad de distancia para llegar a tiendas de intereses comunes para un sector del público.
- Diseño de terminales en aeropuertos, en donde se quiere que los pasajeros que deban hacer un transbordo recorran la distancia mínima entre una y otra terminal teniendo en cuenta el flujo de personas entre ellas.
- Procesos de comunicaciones.
- Diseño de teclados de computadora, en donde se quiere por ejemplo ubicar las teclas de una forma tal en que el desplazamientos de los dedos para escribir textos regulares sea el mínimo.
- Diseño de circuitos eléctricos, en donde es de relevante importancia dónde se ubican ciertas partes o chips con el fin de minimizar la distancia entre ellos, ya que las conexiones son de alto costo.

Como dato curioso, la versión de Koopmans y Beckmann tenía como entrada tres matrices $F = (f_{ij})$, $D = (d_{kl})$, $B = (b_{ik})$ del tipo real, donde (f_{ij}) especifica el flujo entre las instalaciones i y j, (d_{kl}) especifica la distancia entre las instalaciones k y l y (b_{ik}) el costo de instalar la instalación i en la localización k.

Como vemos, en la actualidad no se suele tener en cuenta la matriz B en los problemas de QAP.

2. Consideraciones generales

En este apartado, se describirán todos aquellos aspectos del problema que son utilizados en los algoritmos y que son comunes a todos.

2.1. Representación de la solución

Para representar una solución del QAP, se ha considerado un vector permutación del conjunto

$$N = \{0, 1, \ldots, n-1\}$$

donde n representa el número de unidades/localizaciones de nuestro problema.

Según esta interpretación, cada posición del vector permutación representa una unidad, y el contenido de esa posición representa la localización asociada a dicha unidad. Por ejemplo:

0	1	2	3
0	3	1	2

vendría a significar que la unidad 0 se asigna a la localización 0, la unidad 1 a la localización 3, la unidad 2 a la localización 1 y la unidad 3 a la localización 2.

A nivel de código, se ha utilizado una estructura que contiene el vector permutación, el coste de la misma y una bandera *"needReevaluation"* que indica si es necesario o no recalcular el coste de la solución. La estructura se llama **QAP_solution**.

2.2. Representación de los Flujos y Distancias

Para guardar la información asociada a los flujos y distancias se han utilizado dos matrices $n \times n$ de números en coma flotante, ambas atributos de clase:

- F: Representa la matriz de flujos: la posición f_{ij} representa el flujo existente entre la unidad i y la j. Es una matriz simétrica cuya diagonal principal es nula.
- D: Representa la matriz de distancias: la posición d_{ij} representa la distancia existente entre la localización i y la j. Es una matriz simétrica cuya diagonal principal es nula.

$$F = \begin{pmatrix} 0 & \ldots & f_{1n} \\ \vdots & \ddots & \vdots \\ f_{n1} & \ldots & 0 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & \ldots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{n1} & \ldots & 0 \end{pmatrix}$$

2.3. Función objetivo

La función objetivo (que nos permite calcular el coste total de una solución para determinar su bondad) no es más que una traducción a código de la función matemática explicada en la sección 1:

$$coste(\pi) = \sum_{i=1}^{N} \sum_{j=1}^{N} f_{ij} \cdot d_{\pi(i)\pi(j)}$$

Además, como el algoritmo finaliza cuando se alcanza cierto número de evaluaciones realizadas, cada vez que se llama a esta función se incrementa en uno un contador (*currentEvaluations*, un atributo de clase), para poder así tener control sobre esta condición de parada. Una vez dicho esto, su pseudocódigo es el siguiente:

Algorithm 1 Función para evaluar una solución

Input: Recibe una solución del problema. F y D son atributos de clase.

```
1:  function EvaluateSolution(π)   ▷ Donde π es la solución, F la matriz de flujos y D la de distancias
2:      cost ← 0
3:      for i = 1 to n do
4:          for j = 1 to n do
5:              cost ← cost + F_ij · D_π(i)π(j)
6:          end for
7:      end for
8:      π.coste ← cost
9:      π.needReevaluation ← false                                  ▷ Ya ha sido reevaluada
10:     currentEvaluations ← currentEvaluations + 1                 ▷ Una evaluación más realizada
11: end function
```

Por último, y para facilitar un poco el trabajo, se ha creado una función que se encarga de evaluar todas aquellas soluciones de la población que lo necesiten:

Algorithm 2 Función para evaluar toda la población

Input: Utiliza la población, que es un atributo de clase.

```
1: function EvaluatePopulation
2:     for i = 1 to TamañoPobación do
3:         if población[i].needReevaluation then          ▷ Si su coste debe ser recalculado
4:             EvaluateSolution(población[i])             ▷ Se llama a la función anterior
5:         end if
6:     end for
7: end function
```

2.4. Operador de selección

Como operador de selección para los algoritmos genéticos se ha utilizado el torneo binario. Se seleccionan de forma aleatoria dos candidatos de la población y se compara su coste. Aquel que sea una solución más prometedora, (en este problema, el que tenga menor coste), pasará a formar parte de la población intermedia y el otro será ignorado. Este proceso se repite tantas veces como se desee, según el tamaño que se requiera para la población intermedia, dado como parámetro. Su pseudocódigo es el siguiente:

Algorithm 3 Operador de selección

Input: Recibe el tamaño que debe tener la población intermedia.
Output: Devuelve un vector con la población intermedia.

```
1:  function Select(TamañoPIntermedia)
2:      for i = 1 to TamañoPIntermedia do
3:          uno ← EnteroAleatorio(1, TamañoPoblación)
4:          otro ← EnteroAleatorio(1, TamañoPoblación)
5:          if población[uno].coste < población[otro].coste then
6:              pIntermedia ← pIntermedia ∪ {población[uno]}
7:          else
8:              pIntermedia ← pIntermedia ∪ {población[otro]}
9:          end if
10:     end for
11:     return pIntermedia
12: end function
```

2.5. Operador de mutación

El operador de mutación es el mismo operador de vecino que se utilizó en la búsqueda local durante la práctica anterior. Dada una posición del vector permutación y él mismo, se genera otra aleatoria y se intercambian sus contenidos. Su pseudocódigo es el siguiente:

Algorithm 4 Operador de mutación

Input: Recibe una solución π, y la posición del primer gen que muta.

```
1: function MUTATE(π, pos)
2:     otraPos ← EnteroAleatorio(1, TamañoProblema)      ▷ Repetir mientras pos = otraPos
3:     π(pos) ← π(otraPos)
4:     π(otraPos) ← π(pos)
5:     π.needReevaluation ← true                          ▷ Hay que recalcular su coste
6: end function
```

Nota: La instrucción 2 se repite porque no tendría sentido que realizásemos un intercambio de la posición n por la posición n. Es poco probable que se genere el mismo número pero no imposible.

2.6. Generación aleatoria de la población inicial

Para generar la población inicial de forma aleatoria se ha optado por el siguiente procedimiento:

Algorithm 5 Generar una población inicial aleatoria

```
1: function GENERATERANDOMINITIALPOPULATION
2:     for j = 1 to TamañoPoblación do
3:         index ← {1, ..., n}                           ▷ Crear un vector de índices
4:         lim ← TamañoProblema
5:         for i = 1 to TamañoProblema do
6:             in ← EnteroAleatorio(1, lim)
7:             π(i) ← index(in)
8:             Borrar index(in) y decrementar lim
9:         end for
10:        EvaluateSolution(π)                            ▷ Calculamos su coste
11:        población ← población ∪ {π}                    ▷ Añadimos la solución a la población
12:    end for
13: end function
```

2.7. Operador de cruce basado en posiciones

Uno de los dos operadores de cruce que se han estudiado en esta práctica. Este en concreto, está basado en una idea sencilla: aquellas posiciones que contengan el mismo valor en ambos padres se mantienen en el hijo (para así preservar las asignaciones prometedoras). Por otro lado, las asignaciones restantes se seleccionan en un orden aleatorio para completar el hijo.

Padre$_1$ = (1 2 **3** 4 5 **7** **6** 8 9)
Padre$_2$ = (4 5 **3** 1 8 **7** **6** 9 2)
Hijo' = (* * 3 * * 7 6 * *)
Restos: {1, 2, 4, 5, 8, 9} → Orden aleatorio: {9, 1, 2, 4, 8, 5}
Hijo = (**9** **1** 3 **2** **4** 7 6 **8** **5**)

Pese a que la idea es sencilla, su implementación es más compleja:

Algorithm 6 Operador de cruce basado en posiciones

Input: Recibe por referencia a los dos padres implicados en el cruce.

```
1:  function CrossPosition(π1, π2)
2:      H1.needReevaluation ← true                ▷ Los dos hijos deben reevaluarse
3:      H2.needReevaluation ← true
4:      for i = 1 to TamañoProblema do            ▷ Buscar las partes iguales en los padres
5:          if π1[i] = π2[i] then
6:              H1[i] ← π1[i]
7:              H2[i] ← π2[i]
8:              FreePos[i] ← false    ▷ Nos servirá para saber que posiciones de la solución están libres
9:          else
10:             FreePos[i] ← true
11:             Resto ← Resto ∪ π1[i]    ▷ Aquí guardaremos las localizaciones no iguales en los padres
12:         end if
13:     end for
14:     Resto1 ← BarajarAleatorio(Resto)
15:     Resto2 ← BarajarAleatorio(Resto)
16:     for i = 1 to TamañoProblema do
17:         if not FreePos[i] then
18:             H1[i] ← SiguienteElementoEn(Resto1)
19:             H2[i] ← SiguienteElementoEn(Resto2)
20:         end if
21:     end for
22:     π1 ← H1                                   ▷ Sustituir a los padres
23:     π2 ← H2
24: end function
```

2.8. Operador de cruce OX

En esta práctica he optado por utilizar como segundo operador OX en lugar de PMX. Su idea es la siguiente: Mantenemos en el hijo un subconjunto central generado aleatoriamente de asignaciones del padre. El resto de elementos no pertenecientes a ese subconjunto central se ordenan de acuerdo al orden en el que aparece en el segundo padre. Por último, se asignan a partir de una posición libre cualquiera del hijo (por ejemplo, en la práctica he considerado asignarlas a partir del último elemento del conjunto central). Un ejemplo vale más que mil palabras:

$$\text{Padre}_1 = (7\ 3 \parallel 1\ 8\ 2 \parallel 4\ 6\ 5)$$
$$\text{Padre}_2 = (4\ 3 \parallel 2\ 8\ 6 \parallel 7\ 1\ 5)$$
$$\text{Hijo'} = (*\ * \parallel 1\ 8\ 2 \parallel *\ *\ *)$$
$$\text{Restos: } \{7, 3, 4, 6, 5\} \rightarrow \text{Orden según Padre}_2\text{: } \{4, 3, 6, 7, 5\}$$
$$\text{Hijo}_1 = (7\ 5 \parallel 1\ 8\ 2 \parallel 4\ 3\ 6)$$

Quizás la implementación más compleja de toda la práctica. Su pseudocódigo es el siguiente:

Algorithm 7 Operador de cruce OX

Input: Recibe por referencia a los dos padres implicados en el cruce.

```
1:  function CrossOX(π1, π2)
2:      H1.needReevaluation ← true                    ▷ Los dos hijos deben reevaluarse
3:      H2.needReevaluation ← true
4:      FreePos ← {true, ..., true}
5:      centro ← TamañoProblema/2                     ▷ Computar el conjunto central [low, top]
6:      low ← EnteroAleatorio(2, centro)              ▷ Se deja al menos un elemento en los extremos
7:      top ← EnteroAleatorio(centro + 1, TamañoProblema-1)
8:      FreePos[low, ..., top] ← false                ▷ Marcar esas posiciones como ocupadas
9:      for i = 1 to TamañoProblema do
10:         if not FreePos[i] then
11:             H1[i] ← π1[i]                         ▷ Heredar la parte central
12:             H2[i] ← π2[i]
13:         else
14:             resto1 ← resto1 ∪ {π1[i]}             ▷ Guardar los elementos que no están en el conjunto central
15:             resto2 ← resto2 ∪ {π2[i]}
16:         end if
17:     end for
18:     tmp ← resto1.OrdenarSegún(resto2)
19:     for i = top + 1 to low − 1 do                 ▷ Utiliza el módulo para incrementar i
20:         H1[i] ← SiguienteElementoEn(tmp)
21:     end for
22:     tmp ← resto2.OrdenarSegún(resto1)
23:     for i = top + 1 to low − 1 do                 ▷ Utiliza el módulo para incrementar i
24:         H2[i] ← SiguienteElementoEn(tmp)
25:     end for
26:     π1 ← H1                                       ▷ Sustituir a los padres
27:     π2 ← H2
28: end function
```

Nota: Los bucles de las líneas 18 y 22 son difíciles de explicar en pseudocódigo. En código son como sigue:

```
for(int i=top+1; i<low || i>top; i=(i+1)%_sizeProblem)
```

Esto permite empezar a recorrer desde $top + 1$, cuando se llega al final volver al principio y continuar hasta $low - 1$.

3. Algoritmo Genético Generacional (AGG)

Una vez enunciadas todas las funciones descritas en el apartado anterior, la implementación del algoritmo genético generacial (o AGG para abreviar), es muy sencilla. Para comprenderlo correctamente, únicamente debemos aclarar las siguientes cuestiones:

- Se tiene una probabilidad de cruce de 0.7. No obstante, en lugar de generar un número aleatorio por cada pareja de cromosomas para ver si cruzan o no, optimizaremos el algoritmo calculando a priori el número de cruces que, según la esperanza matemática, deben producirse. Concretamente calcularemos: Número de cruces esperados = (Número de cromosomas de la población / 2) · 0.7. En cada iteración del algoritmo se realizará exactamente el mismo número de cruces y siempre las mismas posiciones: el primero con el segundo, el tercero con el cuarto, el quinto con el sexto...
- Con respecto a las mutaciones, se realizarán de manera similar al apartado anterior. Siempre se realizarán el mismo número de mutaciones. En este caso, las mutaciones se calcularán a priori como: Número de mutaciones = Número de cromosomas · Cantidad de genes por cromosoma · 0.001. Por

cada mutación a realizar, se seleccionará de forma aleatoria tanto el cromosoma como el gen del mismo a mutar.

- El algoritmo incorporará elitismo. En cada nueva población se incluirá la mejor solución de la población anterior. Además, por simplificar, siempre se incluirá dicha solución en la ultima posición de la población (independientemente de la bondad de la solución que haya en esa posición).
- La población $p(t)$, es reemplazada completamente por la población $p(t+1)$, mejore o no.

Dicho esto, el pseudocódigo del algoritmo es el siguiente:

Algorithm 8 Algoritmo Genético Generacional para el QAP

Output: Devuelve la mejor solución encontrada durante el proceso.

```
1:  function AGG
2:      crucesEsperados ← ceil((TamañoPoblación/2) · 0.7)          ▷ Redondeo a la alza
3:      mutacionesEsperadas ← ceil(TamañoPoblación · TamañoProblema · 0.001)
4:      GenerateRandomInitialPopulation()                          ▷ Iniciar población
5:      mejorSolución ← BuscarMejorSolución(población)
6:      while currentEvaluations < 50000 do          ▷ Contador situado en EvaluateSolution
7:          pIntermedia ← Select(TamañoPoblación)                  ▷ Seleccionar
8:          k ← 0
9:          for i = 1 to crucesEsperados do                        ▷ Cruzar
10:             OperadorCruce(pIntermedia[k], pIntermedia[k+1])
11:             k ← k + 2
12:         end for
13:         for i = 1 to mutacionesEsperadas do                    ▷ Mutar
14:             cromosoma ← EnteroAleatorio(1, TamañoPoblación)
15:             gen ← EnteroAleatorio(1, TamañoProblema)
16:             Mutate(pIntermedia[cromosoma], gen)
17:         end for
18:         población ← pIntermedia                                ▷ Reemplazar
19:         EvaluatePopulation()                                   ▷ Evaluar
20:         población[TamañoPoblación] ← mejorSolución             ▷ Elitismo
21:         mejorSolución ← BuscarMejorSolución(población)
22:     end while
23:     return mejorSolución
24: end function
```

Concretamente, se han hecho dos versiones de este algoritmo: una de ellas utiliza el operador de cruce basado en posición, y otra el OX. En el pseudocódigo se ha enunciado de forma genérica.

4. Algoritmo Genético Estacionario (AGE)

En el algoritmo genético estacionario ya no se genera una población intermedia. Únicamente se seleccionan en cada iteración dos padres, ambos cruzan, mutan y dan lugar a dos hijos. Dichos hijos competirán a continuación para entrar en la población. Este algoritmo es más sencillo si cabe que el anterior y sus consideraciones son las siguientes:

- La probabilidad de cruce es 1, es decir, los dos padres seleccionados siempre cruzan.
- Para seleccionar a los dos padres, se lanzan dos torneos binarios.
- La probabilidad de mutación es la misma que antes: 0.001, y al igual que antes también calcularemos el número de mutaciones que se realizarán a priori: Número de mutaciones = Número de cromosomas · Cantidad de genes por cromosoma · 0.001. En este caso, el número de cromosomas es 2.

- El elitismo ya está implícito en los AGE (ya que elimina a los peores, conservando a los mejores).
- Cuando se han generado ambos hijos, la competición por entrar en la población se ha abordado de la siguiente forma: se añaden ambos hijos a la población, se ordena la población según el coste de menor a mayor y se eliminan los dos últimos cromosomas (que son los peores). Con esto nos olvidamos de buscar los peores de la población, comprobar si son mejores los hijos, etc.

Aquí se muestra el pseudocódigo:

Algorithm 9 Algoritmo Genético Estacionario para el QAP

Output: Devuelve la mejor solución encontrada durante el proceso.

```
1:  function AGE
2:      mutacionesEsperadas ← ceil(2 · TamañoProblema · 0.001)        ▷ Redondeo a la alza
3:      GenerateRandomInitialPopulation()                              ▷ Iniciar población
4:      while currentEvaluations < 50000 do          ▷ Contador situado en EvaluateSolution
5:          pIntermedia ← Select(2)                                    ▷ Seleccionar
6:          OperadorCruce(pIntermedia[1], pIntermedia[2])              ▷ Cruzar
7:          for i = 1 to mutacionesEsperadas do                        ▷ Mutar
8:              cromosoma ← EnteroAleatorio(1, 2)
9:              gen ← EnteroAleatorio(1, TamañoProblema)
10:             Mutate(pIntermedia[cromosoma], gen)
11:         end for
12:         EvaluateSolution(pIntermedia[1])                           ▷ Evaluar
13:         EvaluateSolution(pIntermedia[2])
14:         población ← población ∪ {pIntermedia[1], pIntermedia[2]}   ▷ Competir
15:         Ordenar(población)
16:         BorrarDosÚltimasSoluciones(población)
17:     end while
18:     mejorSolución ← BuscarMejorSolución(población)
19:     return mejorSolución
20: end function
```

Al igual que antes, se han implementado dos versiones: una para cada operador de cruce.

5. Algoritmos Meméticos

Un algoritmo memético resulta de hibridar un algoritmo genético generacional junto con una búsqueda local. Concretamente, se ha mezclado el código del AGG desarrollado en apartados anteriores, junto con la búsqueda local implementada en la primera práctica.

Quizás uno de los grandes problemas que tienen los algoritmos meméticos, es determinar cuándo va a realizarse la búsqueda local y sobre qué cromosomas de la población va aplicarse.

En la práctica, se han desarrollado dos variantes fundamentales de algoritmo memético:

- La primera, recibe una probabilidad (pLS) de aplicar la búsqueda local a un cromosoma (de esta, se han realizado dos experimentos: uno con probabilidad 1, es decir, se le aplica LS a todos los cromosomas, y otro con probabilidad 0.1, se aplica LS a un subconjunto de los cromosomas).
- La segunda, recibe un porcentaje, y aplica LS a las porcentaje · N mejores soluciones de la población.

En ambas modalidades, la búsqueda local se ejecutará cada 10 generaciones del AGG.

Dicho esto, antes de ver pseudocódigos, y al igual que hemos hecho con los genéticos, vamos a dar unas consideraciones generales comunes a ambas modalidades:

- La búsqueda Local implementada es idéntica a la realizada en la práctica anterior, es decir, incorpora la factorización del coste, el mecanismo de "Don't Look Bits" y el operador de vecino es igual al de mutación de los algoritmos genéticos. No obstante también tiene algunas variaciones aunque no significativas: El algoritmo de LS finaliza tras las 400 evaluaciones (en lugar de 50000); Ya no se genera una solución aleatoria inicial, sino que parte de una dada como argumento y por último, dentro de la función que calcula ΔCoste, se incrementa también el contador *"currentEvaluations"* (como no se llama explícitamente a la función para evaluar soluciones debido a la factorización del coste, se ha considerado que ΔCoste es la equivalente). Su pseudocódigo es el siguiente:

Algorithm 10 Algoritmo de Búsqueda Local Primero Mejor para QAP

Input: Recibe la solución de la que parte la búsqueda local

```
1:  function LS(π)
2:      evaluaciones ← 0
3:      DLB ← {0, ..., 0}                                   ▷ Se inicializa el vector DLB
4:      repeat
5:          for i = 1 to TamañoProblema and evaluaciones < 400 do
6:              if DLB(i) = 0 then
7:                  for j = 1 to TamañoProblema and evaluaciones < 400 do
8:                      variation ← DeltaCost(π, i, j)          ▷ Dentro se incrementa currentEvaluations
9:                      evaluaciones ← evaluaciones + 1
10:                     if variation < 0 then                   ▷ Si la variación es negativa, el vecino es mejor
11:                         ApplyMove(π, i, j, variation)       ▷ Se aplica el operador
12:                         DLB(i) ← 0          ▷ Como han producido un buen movimiento, se desactivan
13:                         DLB(j) ← 0
14:                         Salir de los dos bucles más internos   ▷ Búsqueda Primero el Mejor!
15:                     end if
16:                 end for
17:                 if no se produjo ninguna mejora al intercambiar π(i) then
18:                     DLB(i) ← 1
19:                 end if
20:             end if
21:         end for
22:
23:     until evaluaciones < 400 and se encuentre una mejor solución
24: end function
```

- Como operador de cruce, se indica en el enunciado que se escoja aquel que produzca mejores resultados. El operador escogido ha sido el Operador de cruce basado en posición.

5.1. Algoritmo Memético basado en probabilidades

Algorithm 11 Algoritmo Memético basado en probabilidades para el QAP

Input: Recibe la frecuencia en la que se aplica la LS y la probabilidad de aplicarla.
Output: Devuelve la mejor solución encontrada durante el proceso.

```
1: function AM(iters, pLS)
2:     timeout ← 0                                          ▷ Lleva cada cuanto se aplica la LS
3:     crucesEsperados ← ceil((TamañoPoblación/2) · 0.7)                    ▷ Redondeo a la alza
4:     mutacionesEsperadas ← ceil(TamañoPoblación · TamañoProblema · 0.001)
5:     GenerateRandomInitialPopulation()                                    ▷ Iniciar población
6:     for i = 1 to TamañoPoblación do                                      ▷ Primera optimización
7:         if RealAleatorio(0,1) ≤ pLS then
8:             LS(población[i])
9:         end if
10:    end for
11:    mejorSolución ← BuscarMejorSolución(población)
12:    while currentEvaluations < 50000 do          ▷ Contador situado en EvaluateSolution y ΔCoste
13:        pIntermedia ← Select(TamañoPoblación)                            ▷ Seleccionar
14:        k ← 0
15:        for i = 1 to crucesEsperados do                                  ▷ Cruzar
16:            CrossPosition(pIntermedia[k], pIntermedia[k+1])
17:            k ← k + 2
18:        end for
19:        for i = 1 to mutacionesEsperadas do                              ▷ Mutar
20:            cromosoma ← EnteroAleatorio(1, TamañoPoblación)
21:            gen ← EnteroAleatorio(1, TamañoProblema)
22:            Mutate(pIntermedia[cromosoma], gen)
23:        end for
24:        población ← pIntermedia                                          ▷ Reemplazar
25:        EvaluatePopulation()                                             ▷ Evaluar
26:        if timeout = iters then                                          ▷ Optimizar
27:            for i = 1 to TamañoPoblación do
28:                if RealAleatorio(0,1) ≤ pLS then
29:                    LS(población[i])
30:                end if
31:            end for
32:            timeout ← 0
33:        end if
34:        población[TamañoPoblación] ← mejorSolución                       ▷ Elitismo
35:        mejorSolución ← BuscarMejorSolución(población)
36:        timeout ← timeout + 1
37:    end while
38:    return mejorSolución
39: end function
```

Como ya hemos dicho antes, se han realizado dos experimentos para este modelo: uno con pLS=1 y otro con pLS=0.1.

5.2. Algoritmo Memético basado en la optimización de los mejores

Este presenta algunas variaciones con respecto al anterior: ya no recibe una probabilidad de LS, sino un porcentaje para determinar a cuantos de los mejores se aplica la optimización local y en orden de seleccionar a los mejores cromosomas, previamente a aplicar la LS, se ordena la población de mejores a peores soluciones

atendiendo al coste de las mismas.

Aquí se muestra el último pseudocódigo:

Algorithm 12 Algoritmo Memético basado en en la optimización de los mejores para el QAP

Input: Recibe la frecuencia en la que se aplica la LS y el porcentaje de la población al que aplicarla.
Output: Devuelve la mejor solución encontrada durante el proceso.

```
1:  function AMBest(iters, porcentaje)
2:      timeout ← 0                                          ▷ Lleva cada cuanto se aplica la LS
3:      crucesEsperados ← ceil((TamañoPoblación/2) · 0.7)    ▷ Redondeo a la alza
4:      mutacionesEsperadas ← ceil(TamañoPoblación · TamañoProblema · 0.001)
5:      optimizaciones ← ceil(TamañoPoblación · porcentaje)  ▷ Calcular el nº de optimizaciones
6:      GenerateRandomInitialPopulation()                    ▷ Iniciar población
7:      Ordenar(población)
8:      for i = 1 to optimizaciones do                       ▷ Primera optimización
9:          LS(población[i])
10:     end for
11:     mejorSolución ← BuscarMejorSolución(población)
12:     while currentEvaluations < 50000 do   ▷ Contador situado en EvaluateSolution y ΔCoste
13:         pIntermedia ← Select(TamañoPoblación)            ▷ Seleccionar
14:         k ← 0
15:         for i = 1 to crucesEsperados do                  ▷ Cruzar
16:             CrossPosition(pIntermedia[k], pIntermedia[k+1])
17:             k ← k + 2
18:         end for
19:         for i = 1 to mutacionesEsperadas do              ▷ Mutar
20:             cromosoma ← EnteroAleatorio(1, TamañoPoblación)
21:             gen ← EnteroAleatorio(1, TamañoProblema)
22:             Mutate(pIntermedia[cromosoma], gen)
23:         end for
24:         población ← pIntermedia                          ▷ Reemplazar
25:         EvaluatePopulation()                             ▷ Evaluar
26:         if timeout = iters then                          ▷ Optimizar
27:             Ordenar(población)
28:             for i = 1 to optimizaciones do
29:                 LS(población[i])
30:             end for
31:             timeout ← 0
32:         end if
33:         población[TamañoPoblación] ← mejorSolución       ▷ Elitismo
34:         mejorSolución ← BuscarMejorSolución(población)
35:         timeout ← timeout + 1
36:     end while
37:     return mejorSolución
38: end function
```

En la práctica, se ha aplicado la optimización al 10 % de las mejores soluciones.

6. Estructuración y Realización de la práctica

Los algoritmos han sido implementados desde cero en C++. Para ello, se ha creado un espacio de nombres llamado PBasedQAP bajo el que se encuentran la estructura QAP_solution ya mencionada en el primer apartado y la clase PBasedALG (Population Based Algorithm). Dicha clase contiene la implementación de todos los algoritmos mencionados en esta práctica, junto con todas las funciones necesarias, además de

los atributos de clase tales como tamaños, flujos, distancias, probabilidades... También se ha utilizado el generador de números aleatorios proporcionado en la web de la asignatura.

El proyecto está estructurado de la siguiente forma:

```
root/
├── Memoria.pdf
└── software/
    ├── bin/
    │   ├── QAP_AG
    │   └── QAP_AM
    ├── data/
    │   ├── *.dat
    │   └── *.sln
    ├── include/
    │   ├── PBasedALG.h
    │   └── random.h
    ├── obj/
    │   └── *.o
    ├── src/
    │   ├── mainAG.cpp
    │   ├── mainAM.cpp
    │   ├── PBasedALG.cpp
    │   └── random.cpp
    └── Makefile
```

- Memoria.pdf: Este mismo documento.
- software: Directorio que contiene el programa en sí.
- bin: Directorio que contiene los ejecutables:
  - QAP_AG: Ejecutable de los 4 algoritmos genéticos implementados.
  - QAP_AM: Ejecutable de los 3 algoritmos meméticos implementados.
- data: Directorio que contiene los datos de QAPLIB.
  - *.dat: Datos del problema.
  - *.sln: Solucion al problema.
- include: Contiene los archivos de cabecera: random.h y PBasedALG.h (este último es importante si se desea consultar aspectos de implementación).
- obj: Contiene los archivos objeto generados tras la compilación.
- src: Contiene los códigos fuente: la implementación de PBasedALG, ambos main y el generador de números aleatorios.
- Makefile: para poder compilar.

Para utilizar el binario basta con introducir en la términal lo siguiente:

```
bash $> cd software/bin/
bash $> ./QAP_AG ../data/<filename>.dat <seed> <mode>
bash $> ./QAP_AM ../data/<filename>.dat <seed> <mode>
```

Donde seed representa la semilla para el generador de números aleatorios y mode tomará como valores 0 (para imprimir solo los costes y tiempo, es decir información reducida) o 1 (para mostrar también el vector permutación obtenido).

Para la realización de la práctica se han lanzado ambos ejecutables para cada uno de los archivos *.dat con una semilla de 88. La información reducida se ha utilizado para agilizar la recolección de tiempos y costes en un fichero.

Por si fuese necesario, también se indica a continuación la orden para compilar los ejecutables:

```
bash $> cd software/
bash $> make  //Para compilar
bash $> make clean  //Para borrar los ejecutables y los archivos objeto
```

7. Experimentos y análisis de resultados

Como ya se ha mencionado en el apartado anterior, para evaluar la bondad de los algoritmos implementados se ha procedido a la ejecución de dichos algoritmos para todos y cada uno de los 20 ficheros de datos de la librería QAPLIB con una semilla de 88 para el generador de números aleatorios.

De cada ejecución se ha medido tanto el coste de la solución obtenida por todos los métodos, como el tiempo que tardan los algoritmos en calcular la solución.

Para determinar como de buenas son las soluciones encontradas, se ha procedido a calcular la desviación existente entre el coste de la solución presente en los ficheros *.sln, y el coste que encuentra nuestro algoritmo. por último, para obtener una visión global de los algoritmos, se ha calculado la media en porcentaje de la desviación obtenida en cada caso así como la media aritmética del tiempo:

$$\frac{1}{|casos|} \cdot \sum_{i \in casos} 100 \cdot \frac{valorAlgoritmo_i - mejorValor_i}{mejorValor_i}$$

7.1. Tablas de resultados

Algoritmo de AGG-posición					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	9.78	0.138648	Sko100a	3.2	1.54548
Chr22b	16.53	0.112349	Sko100f	3.62	1.55708
Chr25a	52.9	0.123969	Tai100a	5.53	1.53714
Esc128	18.75	2.57287	Tai100b	10.11	1.58834
Had20	1.16	0.098329	Tai150b	11.54	3.43344
Lipa60b	21.24	0.640473	Tai256c	3.66	9.53391
Lipa80b	23.04	1.01068	Tho40	6.73	0.293453
Nug28	4.65	0.145395	Tho150	9.72	3.42863
Sko81	3.82	1.03706	Wil50	2.01	0.444963
Sko90	4.35	1.28427	Wil100	2.36	1.5454

Algoritmo de AGG-OX					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	19.82	0.218906	Sko100a	9.04	1.88834
Chr22b	22.47	0.171876	Sko100f	9.45	1.89024
Chr25a	142.2	0.203947	Tai100a	9.99	1.88522
Esc128	84.38	2.93161	Tai100b	22.22	1.89406
Had20	1.85	0.156712	Tai150b	17.65	3.90995
Lipa60b	23.66	0.782945	Tai256c	5.34	10.6563
Lipa80b	26.01	1.27651	Tho40	13.53	0.414132
Nug28	11.27	0.238285	Tho150	12.63	3.90486
Sko81	10.21	1.30376	Wil50	5.63	0.585384
Sko90	10.01	1.56417	Wil100	5.24	1.88821

Algoritmo de AGE-posición					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	13.48	0.162453	Sko100a	2.11	1.5707
Chr22b	8.72	0.138803	Sko100f	2.31	1.56722
Chr25a	36.25	0.167855	Tai100a	4.05	1.56261
Esc128	18.75	2.64919	Tai100b	2.46	1.58388
Had20	0.23	0.128604	Tai150b	6.23	3.31736
Lipa60b	20	0.643304	Tai256c	2.12	9.40061
Lipa80b	22.01	1.038	Tho40	3.71	0.341133
Nug28	4.65	0.187441	Tho150	4.32	3.32278
Sko81	1.94	1.07981	Wil50	1.4	0.479468
Sko90	2.25	1.30251	Wil100	1.26	1.57108

Algoritmo de AGE-OX					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	30.34	0.224766	Sko100a	11.01	1.95311
Chr22b	16.18	0.217344	Sko100f	9.77	1.95471
Chr25a	107.74	0.243789	Tai100a	10.18	1.94608
Esc128	59.38	3.01333	Tai100b	28.44	1.95855
Had20	2.86	0.195302	Tai150b	19.85	4.05631
Lipa60b	24.24	0.841341	Tai256c	6.94	10.8538
Lipa80b	26.2	1.32432	Tho40	13.26	0.476951
Nug28	9.6	0.272947	Tho150	13.2	3.9857
Sko81	11.66	1.35189	Wil50	5.69	0.642835
Sko90	10.31	1.61385	Wil100	5.78	1.96018

Algoritmo de AM-(10,1.0)					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	11.31	0.022871	Sko100a	10.06	0.106832
Chr22b	6.88	0.024195	Sko100f	10.85	0.105568
Chr25a	35.62	0.024879	Tai100a	7.05	0.105849
Esc128	84.38	0.19061	Tai100b	25.55	0.105344
Had20	0	0.023669	Tai150b	16.41	0.185434
Lipa60b	22.25	0.057379	Tai256c	6.62	0.428935
Lipa80b	23.23	0.082572	Tho40	9.07	0.03166
Nug28	4.34	0.024513	Tho150	12.98	0.182304
Sko81	8.67	0.081353	Wil50	5.5	0.042199
Sko90	11.09	0.092581	Wil100	5.7	0.109636

Algoritmo de AM-(10,0.1)					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	10.4	0.038977	Sko100a	5.29	0.326542
Chr22b	6.97	0.039024	Sko100f	6.62	0.328063
Chr25a	38.67	0.044771	Tai100a	5.89	0.325842
Esc128	21.88	0.521118	Tai100b	11.42	0.325635
Had20	0.72	0.026757	Tai150b	8.66	0.630145
Lipa60b	20.46	0.131272	Tai256c	2.76	1.68671
Lipa80b	22.21	0.202036	Tho40	3.45	0.072149
Nug28	3.87	0.040827	Tho150	7.8	0.629786
Sko81	6.16	0.212123	Wil50	2.59	0.109514
Sko90	5.69	0.26162	Wil100	3.2	0.326332

Algoritmo de AM-(10,0.1mej)					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	11.31	0.037961	Sko100a	6.49	0.315421
Chr22b	13.88	0.038075	Sko100f	6.02	0.316272
Chr25a	42.47	0.042686	Tai100a	6.29	0.315381
Esc128	34.38	0.51383	Tai100b	12.4	0.317232
Had20	0	0.026712	Tai150b	8.29	0.656192
Lipa60b	21.58	0.13519	Tai256c	2.84	1.89048
Lipa80b	23.35	0.214828	Tho40	8.99	0.071007
Nug28	4.34	0.039598	Tho150	7.27	0.654546
Sko81	6.35	0.22051	Wil50	2.47	0.09945
Sko90	5.63	0.264247	Wil100	3.03	0.317526

Nota: Los tiempos aparecen en segundos. Además son reducidos debido a las optimizaciones de compilación.

7.2. Resultados Globales en el QAP

Algoritmo	**Desv**	**Tiempo**
Greedy	55,11	0,00011
BL	9,11	0,078
AGG-posición	10,74	1,60
AGG-OX	23,13	1,89
AGE-posición	7,91	1,61
AGE-OX	21,13	1,95
AM-(10,1.0)	15,88	0,10
AM-(10,0.1)	9,74	0,31
AM-(10,0.1mej)	11,37	0,32

Nota: Se han actualizado los tiempos del algoritmo greedy y la búsqueda local bajo las mismas optimizaciones de compilación aplicadas en esta práctica.

7.3. Conclusiones

Dado que hay una gran cantidad de algoritmos de los que hablar, vamos a estructurar el análisis por bloques, en los que se compararán y explicarán distintos aspectos de los experimentos realizados:

- **Valoración general de los Algoritmos Genéticos**: De forma genérica, (y salvo la excepción del algoritmo genético estacionario con el operador de posición), los resultados obtenidos por este tipo de metaheurística poblacional no son especialmente de calidad. Esto tiene dos motivos: el primero de todos

es que por norma general, los Algoritmos Genéticos requieren de una gran cantidad de generaciones para poder llegar a converger correctamente, y quizás limitar el algoritmo a 50000 evaluaciones es quedarse corto. El segundo motivo, es que también por norma general, a los algoritmos genéticos les cuesta especialmente las representaciones de orden, y el QAP, vistos los resultados, es un claro ejemplo de ello.

- **Operadores de cruce:** Continuando con los algoritmos genéticos, los resultados ponen de manifiesto la importacia de elegir correctamente los operadores para el algoritmo, concretamente, en las tablas se refleja la importacia de un buen operador de cruce. El operador OX obtiene de media unos resultados que son un 137 % peores que los de el operador de posición. Como justificación, el hecho de que el operador de posición coloque de forma aleatoria las asignaciones no comunes a ambos padres, le hace introducir una mayor diversidad en las primeras ejecuciones, la cual disminuye a medida que las soluciones empiezan a converger y a ser similares entre sí. En definitiva, se ajusta bastante bien a los requisitos que requiere un algoritmo genético. En contrapartida, el OX, al ser un operador genérico para representaciones de orden, no tiene ninguna garantía de que de buenos resultados en un problema concreto (por no decir además, que es un operador más costoso a nivel de cómputo, lo que se refleja en los tiempos).

- **Generacional VS Estacionario:** Con respecto a qué funciona mejor para este problema, si un algoritmo genético generacional o estacionario, las tablas muestran que los resultados de un estacionario son ligéramente mejores que los de un generacional (realmente, la media de la desviación solo mejora aproximadamente 3 unidades). Para este problema, la alta presión selectiva, así como el elitismo propio de un estacionario (ya que elimina a los peores, conservando a los mejores), parece funcionar muy bien en el QAP. Con respecto a cuestiones de tiempo, realmente no existen diferencias significativas, entre ambos algoritmos, los tiempos son muy similares.

- **Genéticos VS Búsqueda Local:** ¿Realmente merece la pena utilizar un genético frente a la búsqueda local de la primera práctica? Obviando el greedy, ya que da unos resultados desastrosos, y el operador OX (que tampoco es especialmente bueno para este problema), la media en la desviación de la búsqueda local es 9.11, mientras que en los genéticos es de 9.325 (AGG + AGE con posición). En mi opinión, manteniendo los parámetros del genético tal y como están en esta práctica, merece más la pena la búsqueda local, ya que es más sencilla de implementar, tarda muchisímo menos en ejecutarse y da unos resultados similares. No obstante, si afinásemos los parámetros del genético (semilla, evaluaciones, probabilidades...) muy posiblemente se obtengan mejores resultados, pero claro, el tiempo será mayor.

- **Algoritmos Meméticos:** Se supone que los algoritmos meméticos, por norma general suelen mejorar a un algoritmo genético y a la búsqueda local, ya que son una hibridación de ambos, pero en este caso, los resultados son peores que los algoritmos genéticos en solitario (teniendo en cuenta el operador de posición, que es el mismo usado en los meméticos) e incluso que la búsqueda local. ¿Cómo es posible? La respuesta está nuevamente en los parámeros del algoritmo. Una población de 10 cromosomas para el genético es muy poco, los cruces son escasos, con una probabilidad de 0.001 las mutaciones son prácticamente nulas. Que la búsqueda local realice 400 evaluaciones también es muy reducido, por no decir que además contribuye a las evaluaciones globales del genético, y ya hemos dicho antes que 50000 evaluaciones es quedarse corto. En definitiva es normal que los resultados no sean mejores, aunque seguramente, si se ajustasen mejor los parámetros, los algoritmos meméticos obtendrían soluciones de mucha mejor calidad.

  Por último, decir que los tiempos del memético son muy razonables, ya que la búsqueda local es rápida, y como hemos dicho también contribuye a las evaluaciones totales del genético, por lo que realizará menos generaciones y en consecuencia, los tiempos serán menores.

- **Comparación entre las distintas variantes del memético:** El mejor de los tres meméticos resulta de aplicar la búsqueda local a un 10 % aleatorio de la población. ¿Por qué es el mejor? El aplicar búsqueda local a todas las soluciones puede dar lugar a una convergencia prematura, eliminando todo el trabajo que hace el genético. Por otro lado, pese a lo que pueda parecer por intuición, aplicar la búsqueda local solo a los mejores no tiene por qué dar buenos resultados, puede que todos sean puntos cercanos a un óptimo local. En definitiva, la aleatoridad proprociona mejores resultados porque permite

una mayor diversificación, aplicando la búsqueda local a distintos puntos del espacio de búsqueda y mantiendo la diversidad de otras soluciones.

Para finalizar, simplemente resaltar algunos aspectos generales de las tablas: Primero podemos ver que al igual que en la práctica anterior, existen unos problemas que son especialmente complejos de resolver, en los que no se obtienen resultados de calidad para ningún algoritmo. Estos son: Char25a, Esc128, Lipa60b y Lipa80b de forma general (luego cada algoritmo tendrá otros ejemplos para los que también tiene problemas, aunque estos se repiten de forma general). Además, también debemos destacar el hecho de que los algoritmos meméticos, pese a la mala configuración de los parámetros, obtiene el óptimo en el problema Had20 (el más reducido), lo que puede ser un indicio de que con mejores parámetros, los resultados pueden también incrementarse.