Metaheurística - Práctica 2.a

Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Asignación Cuadrática

3º Grado Ingeniería Informática, Grupo 3 (Miércoles)
Salvador Corts Sánchez, 75935233C
salvacorts@correo.ugr.es

Índice

1. Descripción del problema

El problema de asignación cuadrática (en inglés, quadratic assignment problem, QAP) es uno de los problemas de optimización combinatoria más conocidos. 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:

$$QAP = \underset{\pi \in \prod_N}{min} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\pi(i)\pi(j)} \right)$$

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.

2. Consideraciones comunes a los algoritmos utilizados

Esta práctica ha sido diseñada como una librería de metaheurísticas per se. Es decir, existe un tipo de objeto **Solution** y un tipo de objeto **Solver** del cual heredarán los objetos que implementan las diversas metaheurísticas. Cada metaheurística deberá implementar la función *Solve* que devuelve un objeto **Solution**.

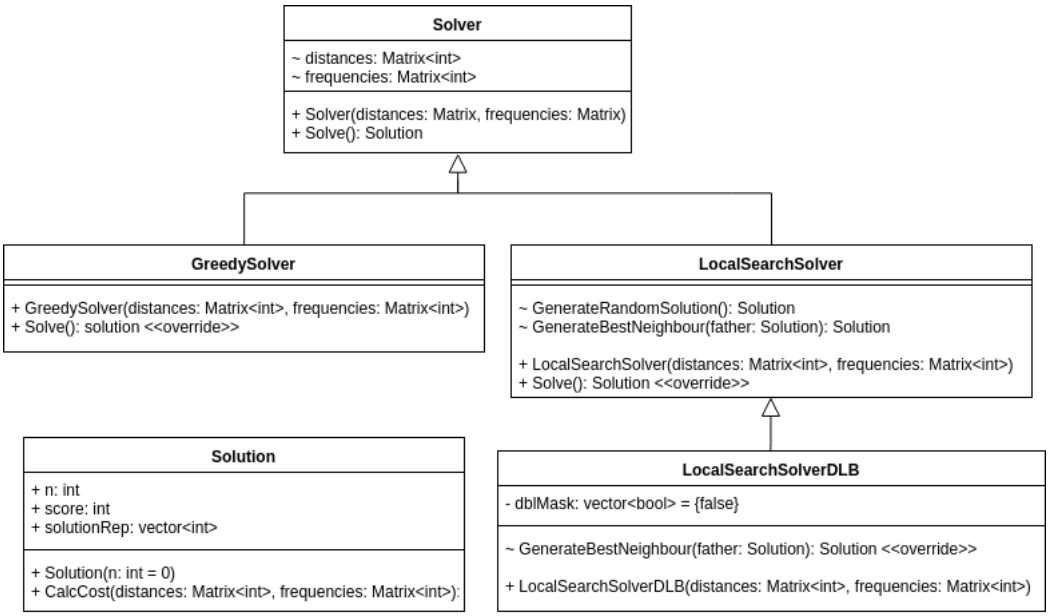


Diagrama de clases

2.1. Clase Solver

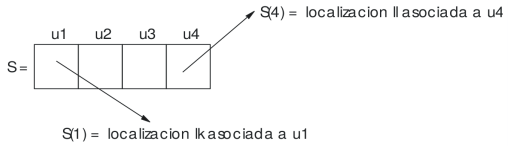
Esta clase debe ser heredada por las metaheurísticas a implementar. Su representación consta de dos matrices:

- **Distancias**: Matriz de distancias entre un punto i y otro j.
- **Frecuencias**: Matriz de flujo entre un objeto i y otro j.

Tiene una función virtual llamada *Solve* que ha de ser implementada por los objetos que hereden de **Solver**. Es la interfaz común a todos los objetos de tipo Solver para obtener una Solución.

2.2. Clase Solution

Sirve para representar una solución, la cual, se implementa como un vector donde cada posición i representa un objeto y alberga la localización j donde debe ser colocado dicho objeto i.



Existe una función *CalcCost* que calcula el coste de dicha solución como:

$$cost = \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} f_{ij} d_{\pi(i)\pi(j)}$$

donde:

- π es la solución al problema.
- 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.

Dado que el cálculo del coste de la solución es bastante costoso, $O(n^2)$, Esta función debe llamarse manualmente al menos una vez para obtener el coste y que este se guarde en la representación de la clase.

3. Algoritmo Greedy

Se basa en el cálculo de los potenciales de flujo y distancia definidos como:

$$\hat{f}_i = \sum_{j=1}^{n} f_{ij} \qquad \hat{d}_i = \sum_{j=1}^{n} d_{ij}$$

El algoritmo irá seleccionando la unidad i libre con mayor $\hat{f}_i$ y le asignará la localización j libre con menor $\hat{d}_j$. Su implementación en pseudocódigo es la siguiente:

```
1  # Calcula los potenciales
2  dp = fp = vector(n)
3  for i = 1 to n do
4  |   f̂_i = d̂_i = 0
5  |   for j = 0 to n do
6  |   |   f̂_i = f̂_i + f_ij
7  |   |   d̂_i = d̂_i + d_ij
8  |   end
9  |   dp_i = d̂_i
10 |   fp_i = f̂_i
11 end
12 # Calcula la mejor combinación. π es la representación de la solución
13 locAssigned = unitAssigned = vector(n){0}
14 for i = 1 to n do
15 |   bestf̂ = −∞;   bestf̂_index = 0
16 |   bestd̂ = ∞;   bestd̂_index = 0
17 |   for j = 0 to n do
18 |   |   f̂_i = fp_j;   d̂_i = dp_j
19 |   |   if f̂_i > bestf̂ and unitAssigned_j ≠ 1 then
20 |   |   |   bestf̂ = f̂_i;   bestf̂_index = j
21 |   |   end
22 |   |   if d̂_i < bestd̂ and locAssigned_j ≠ 1 then
23 |   |   |   bestd̂ = d̂_i;   bestd̂_index = j
24 |   |   end
25 |   end
26 |   π(bestf̂_index) = bestd̂_index
27 |   unitAssigned_{bestf̂_index} = locAssigned_{bestd̂_index} = 1
28 end
```

Algorithm 1: greedy.cpp - GreedySolver::Solve

4. Algoritmo de Búsqueda Local

Vamos a utilizar una **búsqueda local del primer mejor**. Cuando se genera una solución vecina que mejora a la actual, se toma esta como solución y se pasa a la siguiente iteración. Se detiene la búsqueda cuando no se genera ningún vecino mejor que la solución actual. La implementación de dicha idea, que será la función *Solve*, se puede ver como:

```
1 π = GenerateInitialSolution() # Será aleatoria
2 do
3 |   π' = GenerateBestNeighbour(π)
4 |   if ∃π' then π = π';
5 while ∃π';
```

Algorithm 2: localSearch.cpp - LocalSearchSolver::Solve

A fin de minimizar el riesgo de quedarnos en un óptimo local, vamos a partir de una solución aleatoria en vez de partir de una solución greedy. Dicha solución aleatoria se genera de la siguiente manera:

```
1 assigned = vector(n){0}
2 for i = 1 to n do
3 |   do
4 |   |   r = random() mod n
5 |   while assigned_r ≠ 0;
6 |   π(i) = r
7 |   assigned_r = 1
8 end
```

Algorithm 3: solution.cpp - Solution::GenerateRandomsolution

Como se comentó anteriormente, el proceso de cálculo del coste de la solución es de orden cuadrático por lo que realizar dicho calculo con cada vecino es sumamente costoso; En su lugar, vamos a considerar una **factorización** (con eficiencia $O(n)$) teniendo en cuenta solo los cambios realizados por el movimiento de intercambio para generar el vecino. El incremento del coste de cambiar el elemento en la posición r r por el de s se define como:

$$\Delta C(\pi, r, s) = \sum_{k=1, k\neq r,s}^{n} \begin{bmatrix} f_{rk}(d_{\pi(s)\pi(k)} - d_{\pi(r)\pi(k)}) + f_{sk}(d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)}) + \\ f_{kr}(d_{\pi(k)\pi(s)} - d_{\pi(k)\pi(r)}) + f_{ks}(d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)}) \end{bmatrix}$$

Si $\Delta C(\pi, r, s) < 0$, el resultado de cambiar r por s es favorable, es decir, el costo es menor por lo que tomaremos el vecino resultante de este cambio como solución actual y generamos nuevos vecinos a partir de este.

La función que hace uso de esta factorización para explorar los vecinos de una solución se implementaría como:

```
1  def GenerateBestNeighbour(π):
2      for r = 1 to n/2 do
3          for s = r + 1 to n do
4              if ΔC(π, r, s) < 0 then
5                  π' = π
6                  t = π'(r)
7                  π'(r) = π'(s)
8                  π'(s) = t
9                  return π'
10             end
11         end
12     end
13 end
```

Algorithm 3: localSearch.cpp-LocalSearchSolver::GenerateBestNeighbour

Como vemos, podemos reducir considerablemente el numero de iteraciones totales iterando en r en el primer bucle hasta $n/2$ y en el segundo desde $r + 1$ hasta n, ya que así podemos evitar comparar dos veces el mismo movimiento. Es lo mismo cambiar r por s que s por r.

4.1. Búsqueda Local con *Don't Look Bits*

Como estamos utilizando una **búsqueda local del primer mejor**, podemos definir una lista de candidatos a la que llamamos ***Don't Look Bits*** que reducirá significativamente el tiempo de ejecución.

Se trata de un un vector de bits inicialmente a 0, esto nos indica que todos los movimientos pueden ser considerados. Si tras probar todos los movimientos asociados un bit no hemos encontrado ninguna mejora, cambiaremos el valor de dicho bit a 1, indicando que esta unidad no debe ser tenida en cuenta hasta que dicha unidad asociada a ese bit se vea implicada en un movimiento que mejora la solución actual, en cuyo caso el bit será nuevamente 0.

Podemos ejemplificar este algoritmo con el siguiente pseudocódigo:

```
1  dlbMask = vector(n){0} # Don't look bits
2  def GenerateBestNeighbour(π):
3      for r = 1 to n do
4          if dlbMask_r ≠ 0 then continue;
5          for s = 1 to n do
6              if ΔC(π, r, s) < 0 then
7                  π' = π
8                  t = π'(r)
9                  π'(r) = π'(s)
10                 π'(s) = t
11                 dlbMask_r = dlbMask_s = 0
12                 return π'
13             end
14         end
15         dlbMask_r = 1
16     end
17 end
```

5. Algorítmos Genéticos

Se han implementado dos tipos de algorítmos genéticos:

- Generacionales.
- Estacionarios.

Ambos comparten los operadores de **Mutación**, **Cruce** y **Evaluación** así como la del **método de búsqueda**. La diferencia entre ambos está en los operadores de **Selección** y **Remplazo** que se explicarán a fondo mas tarde.

Estos operadores actuan sobre una Población compuesta de n cromosomas. Es decir, un conjunto (vector) de n soluciones diferentes que modificaremos, mezclaremos y compararemos a fin de encontrar la solución más óptima posible sorteando óptimos locales.

Método de búsqueda de un algorítmo genético

Partiremos de una población generada aleatoriamente creando, con la función descrita en el apartado 4. (Búsqueda Local), tantos cromosomas soluciones como tamaño de la población deseemos.

```
1 def GenerateRandomPopulation():
2 |   P = Population(n){0} # Población de n cromosolmas
3 |   for i = 1 to n do
4 |   |   # P_i es el cromosoma i de la población P
5 |   |   P_i = GenerateRandomSolution()
6 |   end
7 |   return p
8 end
```

Algorithm 4: *genetic.cpp* - GeneticAlg::CreateRandomPopulation

Sobre esta población inicial aplicaremos los operadores hasta que se haya alcanzado el número máximo de evaluaciones definidas.

```
1  def Solve():
2  |  P = GenerateRandomPopulation()
3  |  π = Evaluate(P)
4  |  π_best = r
5  |  while evaluaciones ≠ limite do
6  |  |  P' = Select(P)
7  |  |  P' = Cross(P')
8  |  |  P' = Mutate(P')
9  |  |  P' = Replace(P, P')
10 |  |  π = Evaluate(P)
11 |  |  if π mejor que π_best then
12 |  |  |  π_best = π
13 |  |  end
14 |  end
15 |  return π_best
16 end
```

Algorithm 5: *genetic.cpp* - GeneticAlg::Solve

Operador de Mutación

Para el problema de *QAP*, la mutación consiste en intercambiar el valor de dos genes de un cromosoma, es decir, cambiar el valor de dos posiciones de una solución.

Para evitar el coste computacional de generar una gran cantidad de números aleatorios, a partir de una probabilidad p_{mutar}, vamos a calcular la esperanza del número n_{mutar} de cromosomas que mutarán de nuestra población como:

$$n_{mutar} = \lceil n \cdot p_{mutar} \cdot len(\pi) \rceil$$

Donde:

- n: Tamaño de la población.
- $len(\pi)$: Tamaño de la solución.

Empezaremos a mutar los n_{mutar} cromosomas contigüos desde un cromosoma aleatroio de nuestra población.

```
1  def Mutate(P):
2      P' = P
3      n_mutar = ⌈n · p_mutar · len(P_0)⌉
4      i = random in [0, n)
5      for j = 1 to n_mutar do
6          r1 = random in [0, len(P'_0))
7          r2 = random in [0, len(P'_0))
8          if coste P_i desconocido then
9              Calcula coste P_i
10             evaluaciones++
11         end
12         π_o = P'_i
13         t = P'_i(r1)
14         P'_i(r1) = P'_i(r2)
15         P'_i(r2) = t
16         coste P'_i(s) = coste π_o + ΔC(π_o, r1, r2)
17     end
18     return P'
19 end
```

Algorithm 6: *genetic.cpp* - GeneticAlg::Mutate

Operador de Evaluación

Con este operador calcularemos los costes de los cromosomas de una población P que no hayan sido calculados aún. Trás calcular los costes de todos los cromosomas, devolveremos la solución mas óptima de la población, es decir, el cromósoma con el menor coste asociado.

```
1  def Evaluate(P):
2      best_score = ∞
3      for i = 1 to n do
4          if coste P_i desconocido then
5              Calcula coste P_i
6              evaluaciones++
7          end
8          if coste P_i < best_score then
9              best_score = coste P_i
10             π_best = P_i
11         end
12     end
13     return π_best
14 end
```

Algorithm 7: *genetic.cpp* - GeneticAlg::Evaluate

Operador de Cruce

Gracias a este operador, podremos simpular el cruce de genes que se da en la naturaleza. A partir de dos cromosomas padres, combinaremos sus genes para crear dos nuevos cromosomas hijos.

De nuevo, a fin de minimizar el coste computacional de generar números aleatorios, a partir de una probabilidad p_{cruce}, vamos a calcular la esperanza del número n_{cruce} de cromosomas consecutivos que se cruzarán entre si como:

$$n_{cruce} = \lceil n \cdot p_{cruce} \rceil$$

Se han desarrollado dos tipos de cruces distintos: **basado en posición** y **OX**.

- **Cruce basado en posición**

  Aquellas posiciones que contengan el mismo valor en ambos padres se mantienen en el hijo. Las asignaciones restantes se seleccionan en un orden aleatorio para completar el hijo. Por ejemplo:

$$Padre_1 = (1,2,3,4,5,6,7,8,9) \quad Padre_2 = (4,5,3,1,8,7,6,9,2)$$
$$coincidencias = (*,*,3,*,*,7,6,*,*)$$
$$resto\_Padre_1 = (1,2,4,5,8,9) \rightarrow Barajamos : (9,1,2,4,8,5)$$
$$resto\_Padre_2 = (4,5,1,8,9,2) \rightarrow Barajamos : (5,1,4,2,9,8)$$
$$hijo_1 = (9,1,3,2,4,7,6,8,5) \quad hijo_2 = (5,1,3,4,2,7,6,9,8)$$

  La implementación es la siguiente:

```
1  def Cross(P):
2      P' = P;   n_cruce = ⌈n · p_cruce⌉
3      for i = 1 to n_cruce do
4          equals = vector(len(P_i)){0};   nonEquals = ∅
5          for j = 1 to len(P_i) do
6              if P_i(j) == P_{i+1}(j) then
7                  π'(j) = π''(j) = P_i(j)
8                  equals_j = 1
9              end
10             else
11                 nonEquals = nonEquals ∪ P_i(j)
12             end
13         end
14         shuffle(nonEquals)
15         k = 0
16         for j = 1 to len(P_i) do
17             if equals_j ≠ 1 then
18                 π'(j) = nonEquals_k
19                 π''(j) = nonEquals_{len(nonEquals)−1−k}
20                 k = k + 1
21             end
22         end
23         P'_i = π';    P'_{i+1} = π''
24         i = i + 2
25     end
26     return P'
27 end
```

Algorithm 8: *genetic.cpp* - GeneticAlg::Cross

- **Cruce OX**

  Los dos hijos comparten los centros de los padres. El resto de elementos que no estan en la parte central, se ordenan segun el otro padre.

$$Padre_1 = (7,3,1,8,2,4,6,5) \quad Padre_2 = (4,3,2,8,6,7,1,5)$$
$$centro\_padre_1 = (*,*,1,8,2,*,*,*) \quad centro\_padre_2 = (*,*,2,8,6,*,*,*)$$
$$resto\_Padre_1 = (7,3,4,6,5) \rightarrow Odenamos\ segun\ Padre_2 : (4,3,6,7,5)$$
$$resto\_Padre_2 = (4,3,7,1,5) \rightarrow Odenamos\ segun\ Padre_1 : (7,3,1,4,5)$$
$$hijo_1 = (7,5,1,8,2,4,3,6) \quad hijo_2 = (4,5,2,8,6,7,3,1)$$

  La implementación mas sencilla consiste en un bucle anidado con complejidad $O(n^2)$, sin embargo, sacrificando un poco de espacio en memoria, podemos conseguir hacer esta operación con una complejidad lineal, es decir $O(n)$.

  Para ello, utilizaremos dos tipos de estructuras de datos auxiliares:

  - Dos vectores de tamaño igual al tamaño de la solución del problema ($len(\pi)$). Nos servirá como tabla hash para asociar a cada valor del padre el índice en el que se encuentra en la solución de este. Utilizaremos un vector por padre.
  - Dos sets que se ordenarán a partir de los vectores decritos arriba. En el primer set, meteremos los valores externos al centro del segundo padre y se ordenarán en función del índice que tengan en el vector de índices del primer padre. En el segundo, haremos lo mismo pero al contrario.

  Una vez hayamos completado ambas estructuras de datos, copiamos los valores ordenados de los sets en cada hijo.

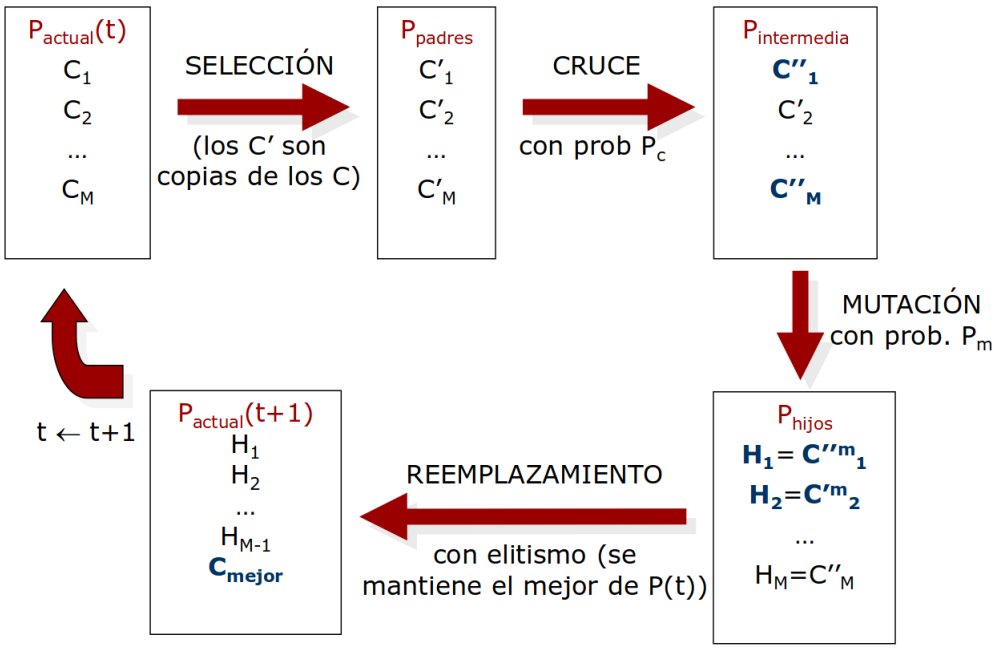
Su implementación es la siguiente:

```
1  def CrossOX(P):
2      P' = P;   n_cruce = ⌈n · p_cruce⌉;   step = len(P_0)/3
3      start = ⌊step⌋;   end = ⌊step⌋ + ⌈step⌉
4      for i = 1 to n do
5          S1 = vector(len(P_i));   S2 = vector(len(P_{i+1}))
6          for j = 1 to len(P_i) do
7              S1_{P_{i_j}} = j;   S2_{P_{i+1_j}} = j
8              if j ≥ start and j ≤ end then
9                  π'_j = P_{i_j};   π''_j = P_{i+1_j}
10             end
11         end
12         # Ordena de menor a mayor con S1 y S2
13         orderedByS1 = set();   orderedByS2 = set()
14         for j = 1 to len(P_i) do
15             if j == start then
16                 j = end;   continue
17             end
18             orderedByS1 = orderedByS1 ∪ P_{i+1_j}
19             orderedByS2 = orderedByS2 ∪ P_{i_j}
20         end
21         k = 0 for j = end+1 ; j ≠ start ; j = (j+1) mod len(P_i) do
22             if j == start then
23                 j = end;   continue
24             end
25             π'_j = orderedByS2_k;   π''_j = orderedByS1_k  k = k + 1
26         end
27         P'_i = π';   P'_{i+1} = π''  i = i + 2
28     end
29     return P'
30 end
```

Algorithm 9: *genetic.cpp* - CrossOX

5.1. Algorítmo Genético Generacional *(AGG)*

Durante cada iteración, se crea una población completa con nuevos candidatos. La nueva población reemplaza directamente a la población anterior. Al ser elitista, se mantiene la mejor solución obtenida hasta el momento intercambiando dicha solución por la peor de la nueva población.



Se ha desarrollado dos variantes elitistas: una con el operador de cruce basado en posición, y otra con el operador de cruce OX.

Solo varían los operadores de selección y remplazo que se describen a continuación. El resto de operadores son los descritos anteriormente.

Operador de Selección

Se usará un torneo binario, consistente en elegir aleatoriamente dos individuos de la población y seleccionar el mejor de ellos. Se aplicarán tantos torneos como individuos existan en la población.

La implementación es la siguiente:

```
1  def Select(P):
2      for i = 1 to n do
3          r1 = random in [0, n)
4          r2 = random in [0, n)
5          if coste P_r1 desconocido then
6              Calcula coste P_r1
7              evaluaciones++
8          end
9          if coste P_r2 desconocido then
10             Calcula coste P_r2
11             evaluaciones++
12         end
13         if coste P_r1 ≤ P_r2 then
14             P'_i = P_r1
15         end
16         else
17             P'_i = P_r2
18         end
19     end
20     return P'
21 end
```

Algorithm 10: *agg.cpp* - AGG::Select

Operador de Remplazo

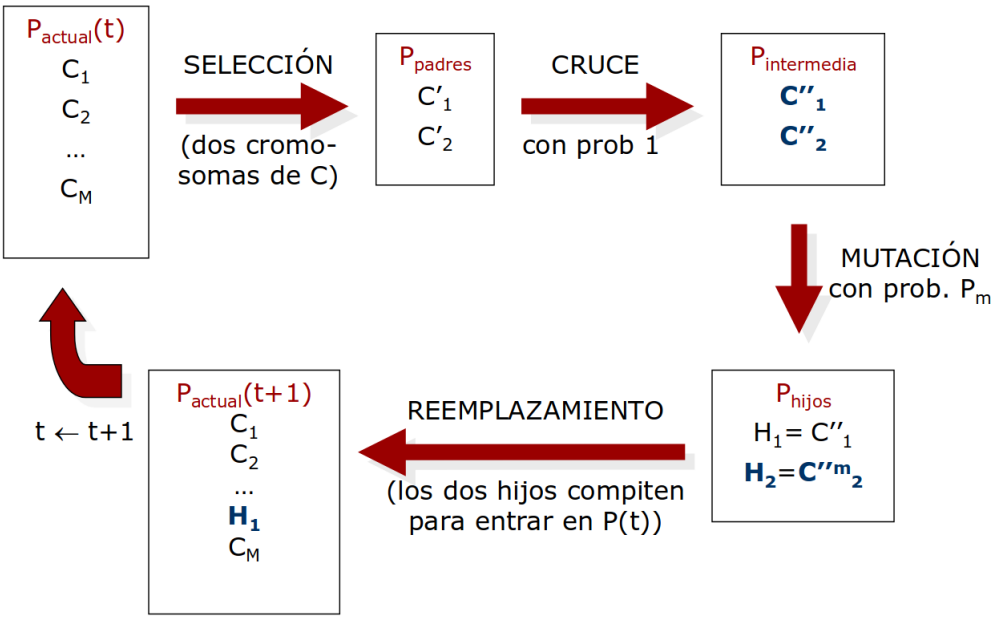
Se remplaza la anterior población por la nueva manteniendo la mejor solución obtenida hasta el momento. El pseudocódigo que implementa dicha funcionalidad es:

```
1  def Replace(P, P'):
2      P'' = P'
3      peor_coste = peor_indice = 0
4      if ∃π_best then
5          for i = 1 to n do
6              if coste P''_i desconocido then
7                  Calcula coste P''_i
8                  evaluaciones++
9              end
10             (AGG) if peor_coste < costeP''_i then
11                 peor_coste = costeP''_i
12                 peor_indice = i
13             end
14         end
15         P''_peor_indice = π_best
16     end
17     return P''
18 end
```

Algorithm 11: *agg.cpp* - AGG::Replace

5.2. Algorítmo Genético Estacionario *(AGE)*

Durante cada iteración se escogen dos padres de la población y se les aplican los operadores genéticos. Este modelo es elitista. Además, produce una convergencia rápida ya que se remplazan los peores cromosomas de la población por los dos padres escogidos y modificados anteriormente, solo si estos mejoran a los peores de la población anterior.



De nuevo, se han implementados dos variantes; una que utiliza un cruce basado en posición y otra en cruce OX.

Al igual que en el AGG, solo cambian los operadores de selección y remplazo. No obstante notese que los padres escogidos siempre cruzan ya que la probabilidad de cruce es 1 en los AGE.

Operador de Selección

De nuevo se utiliza un torneo binario. Se aplicarán 2 torneos para escoger dos padres con los que operar posteriormente.

La implementación es la siguiente:

```
1  def Select(P):
2      for i = 1 to 2 do
3          r1 = random in [0, n)
4          r2 = random in [0, n)
5          if coste P_r1 desconocido then
6              Calcula coste P_r1
7              evaluaciones++
8          end
9          if coste P_r2 desconocido then
10             Calcula coste P_r2
11             evaluaciones++
12         end
13         if coste P_r1 ≤ P_r2 then
14             P'_i = P_r1
15         end
16         else
17             P'_i = P_r2
18         end
19     end
20     return P'
21 end
```

Algorithm 12: *age.cpp* - AGE::Select

Operador de Remplazo

Se remplazan los (dos) peores cromosomas de la población original por los cromosomas de la nueva población solo si los segundos mejoran los primeros.

Partiendo de una implementación básica con complejidad temporal $O(n^2)$; De nuevo se ha conseguido una implementación que sacrificando espacio consigue una eficiencia $O(n)$.

La representación en pseudocódigo de este operador es la siguiente:

```
1  def Replace(P, P'):
2      P'' = P
3      peores = set() # Se ordenan los valores en función del coste
4      for i = 1 to len(P'') do
5          if coste P''_i desconocido then
6              Calcula coste P''_i
7              evaluaciones++
8          end
9          if peores == ∅ or costeP''_i > costepeores_0 then
10             if len(peores) ≥ len(P') then
11                 Elimina peores_0 de peores
12             end
13             peores = peores ∪ P''_i
14         end
15     end
16     for i = 1 to len(P') do
17         if coste P'_i desconocido then
18             Calcula coste P'_i
19             evaluaciones++
20         end
21         for j = 1 to len(peores) do
22             if coste peores_j > coste P'_i then
23                 cambiar el cromosoma peores_j de P'' por P_i
24                 eliminar peores_j de peores
25             end
26         end
27     end
28     return P''
29 end
```

Algorithm 13: *age.cpp* - AGE::Replace

6. Algorítmos Meméticos

Explicar solve

7. Procedimiento considerado para desarrollar la práctica

Esta práctica ha sido desarrollada en **C++** como una librería de metaheurísticas. La estructura del proyecto es la siguiente:

```
/Software
├── bin/ ..... Archivos ejecutables
│   └── practica1 ..... Ejecutable principal de la práctica
├── build/ ..... Directorio para compilación con CMake
├── doc/ ..... Otra documentación y código LaTeX de este documento
├── include/ ..... Cabeceras
├── instancias/ ..... Instancias de problemas sobre QAP
│   ├── *.dat ..... Definición de un problema
│   └── *.sln ..... Solución a un problema
├── src/ ..... Código fuente
├── CMakeLists.txt ..... Instrucciones de compilación para CMake
└── readme.txt ..... Instrucciones de uso
```

Para compilar este proyecto necesitamos las herramientas *g++* y *CMake*. Para instalarlas en un sistema **Ubuntu** o derivado, ejecutamos:

```
sudo apt-get install g++ cmake
```

Podemos compilar el proyecto con dos niveles de optimización:

- **Debug**: Sin optimización y con símbolos de depuración. Ejecutar CMake con opción *-D CMAKE_BUILD_TYPE=Debug*
- **Release**: Máxima optimización en la compilación. Por defecto.

Se compila con las siguientes instrucciones:

```
cd build/
# Para debug: cmake -D CMAKE_BUILD_TYPE=Debug ..
cmake ..
make clean
make
cd ..
```

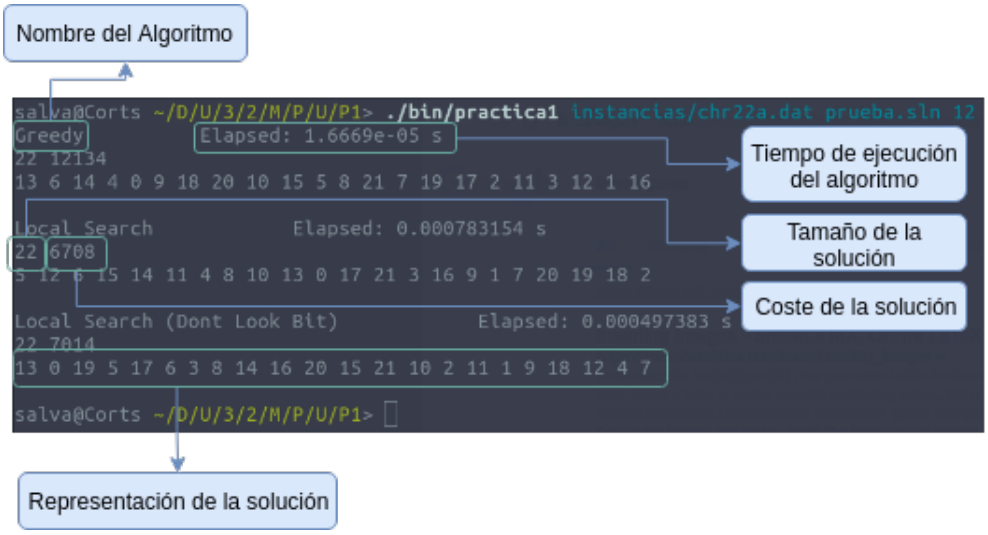
La sintaxis de ejecución es la siguiente[1]:

```
./bin/main instancias/<problema>.dat <semilla>
```

Por ejemplo:

```
./bin/main instancias/chr22a.dat 12
```

La salida del programa tiene la siguiente estructura:



[1]El parámetro semilla es opcional, por defecto $semilla = 7$

8. Experimentos y análisis de resultados

Con el fin de comparar los algoritmos implementados con los ya existentes, vamos a calcular para cada algoritmo los siguientes parámetros:

- ***Desv***: Media de las desviaciones en porcentaje, del valor obtenido por cada método en cada instancia respecto al mejor valor conocido para ese caso.

$$\frac{1}{|casos|}\sum_{i=1}^{casos} 100\frac{valoralgoritmo_i - mejorValor_i}{mejorValor_i}$$

  Obtendremos los mejores valores conocidos de *QAPLIB*[2]

- ***Tiempo***: se calcula como la media del tiempo de ejecución empleado por el algoritmo para resolver cada caso del problema.

Cuanto menor es el valor de Desv para un algoritmo, mejor calidad tiene dicho algoritmo. Por otro lado, si dos métodos obtienen soluciones de la misma calidad (tienen valores de Desv similares), uno será mejor que el otro si emplea menos tiempo en media.

Parámetros del experimento:

- *El valor de la semilla para este experimento es 7.*
- *Se compilara con parámetro* ***debug*** *a fin de contrastar aún mas los resultados del tiempo.*

[2]http://anjos.mgi.polymtl.ca/qaplib//inst.html

8.1. Resultados Greedy

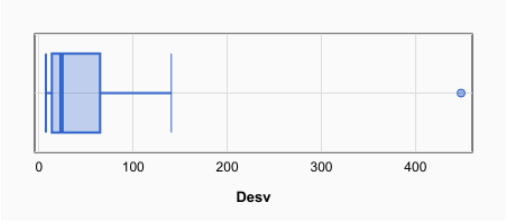
Cuadro 1: goo.gl/yr6uN9

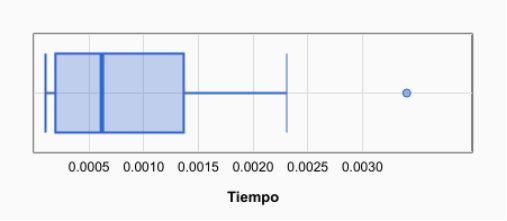
Algoritmo Greedy					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	97,11	0,000126925	**Sko100a**	13,36	0,000630563
Chr22b	134,48	0,000125567	**Sko100f**	13,77	0,00230857
Chr25a	448,47	0,000175691	**Tai100a**	13,80	0,000616869
Esc128	140,63	0,00117301	**Tai100b**	32,79	0,000626035
Had20	7,71	0,000102693	**Tai150b**	24,97	0,00136394
Lipa60b	27,59	0,000289296	**Tai256c**	120,48	0,00340613
Lipa80b	28,58	0,000382927	**Tho40**	29,96	0,000384573
Nug28	22,88	0,00020704	**Tho150**	16,98	0,00204635
Sko81	15,50	0,0013694	**Wil50**	11,68	0,00017995
Sko90	13,61	0,00162198	**Wil100**	7,33	0,000619954

Como podríamos esperar, el algoritmo greedy es muy rápido, aunque las soluciones obtenidas distan bastante de la mejor conocida. Podríamos discutir su utilidad desde un punto de vista práctico; ¿En qué escenarios puede sernos útil una búsqueda greedy?

1. Si ofrecemos un servicio donde la optimalidad de la solución a un problema es lo de menos y tenemos que responder muchas solicitudes con pocos recursos de manera muy rápida.

2. Como algoritmo auxiliar de cara a obtener una solución inicial para posteriormente tratar de mejorarla mediante el uso de algoritmos mas complejos. No obstante, esto no siempre es buena idea pues corremos el riesgo de caer en óptimos locales. Hay que usarlos con algoritmos que sean capaces de esquivar con relativa facilidad óptimos locales.





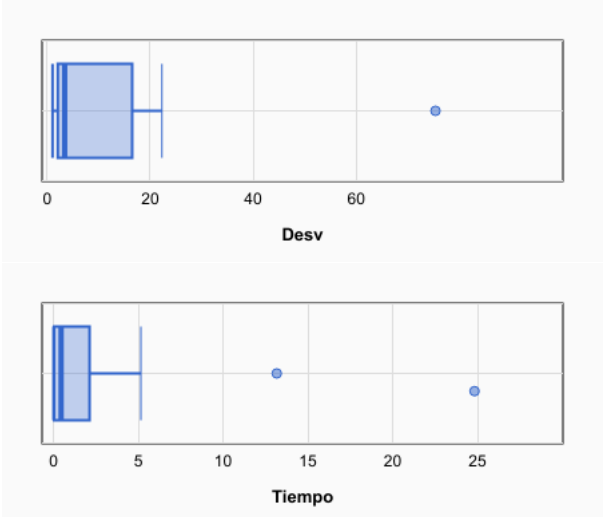
8.2. Resultados Búsqueda Local

Cuadro 2: goo.gl/SJJh1a

Algoritmo Búsqueda Local					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	17,41	0,00255601	**Sko100a**	2,53	1,54504
Chr22b	15,63	0,00382979	**Sko100f**	2,21	2,05617
Chr25a	75,40	0,00940898	**Tai100a**	3,35	0,806562
Esc128	18,75	0,360157	**Tai100b**	4,94	3,26802
Had20	1,44	0,0058809	**Tai150b**	3,50	24,8278
Lipa60b	19,79	0,114542	**Tai256c**	1,26	5,15624
Lipa80b	22,30	0,291904	**Tho40**	5,34	0,0626425
Nug28	8,94	0,0114589	**Tho150**	2,78	13,1679
Sko81	2,70	0,540895	**Wil50**	1,96	0,10151
Sko90	1,86	1,46495	**Wil100**	1,07	2,19904

Aunque no se obtienen soluciones óptimas, estas están muy cerca de serlo. Podemos apreciar que este algoritmo trabaja muy bien con problemas relativamente pequeños pero dado que, a diferencia del greedy, el tiempo crece significativamente en función del tamaño del problema, utilizarlo como algoritmo de propósito general para solucionar problemas no es lo idóneo.

Sin embargo, en combinación con otros algoritmos como los Genéticos podemos obtener resultados aún mejores y en un tiempo mas aceptable dado que restringimos el espacio de búsqueda para este algoritmo.

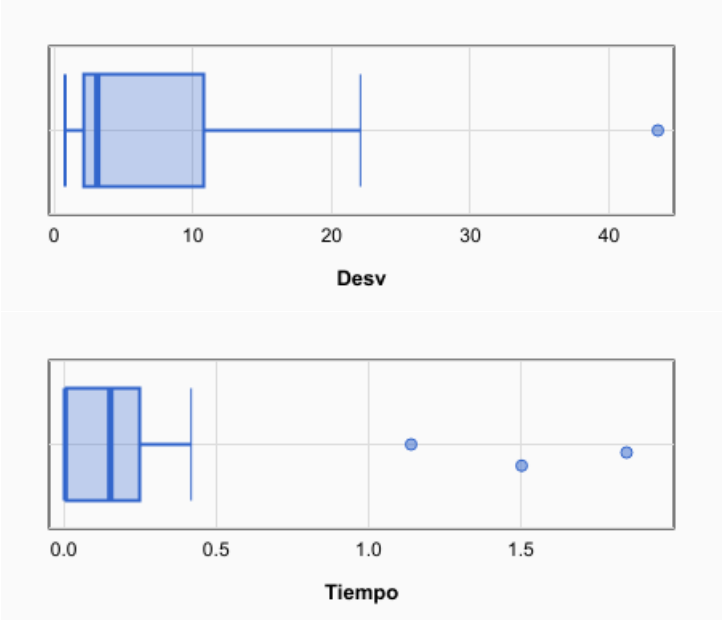


8.3. Resultados Búsqueda Local con *Don't Look Bits*

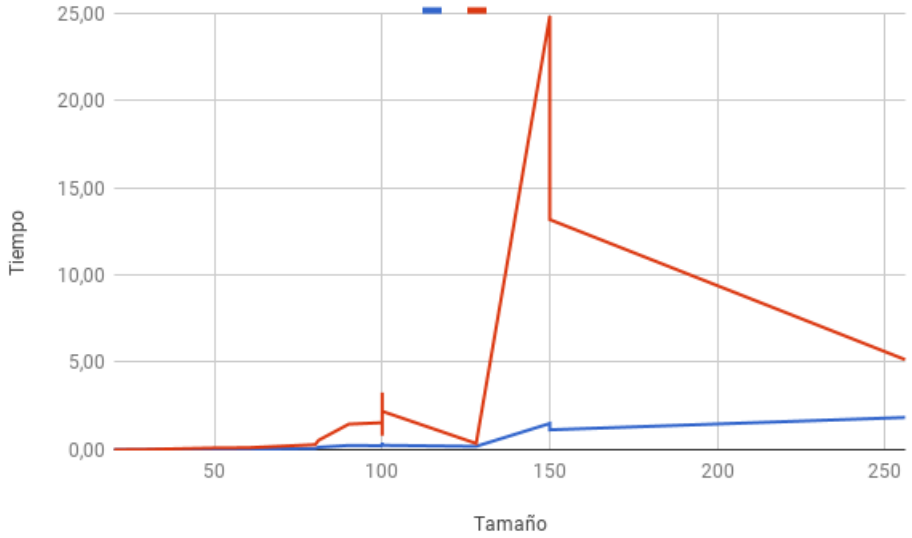
Cuadro 3: goo.gl/9frEmN

Algoritmo Búsqueda Local DLB					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	18.23	0,0020404	**Sko100a**	2.78	0,229239
Chr22b	9.07	0,0059782	**Sko100f**	2.5	0,226626
Chr25a	43.52	0,00274717	**Tai100a**	4.12	0,164417
Esc128	12.5	0,193508	**Tai100b**	4.89	0,418418
Had20	2.11	0,00561109	**Tai150b**	2.59	1,5031
Lipa60b	20.23	0,0302634	**Tai256c**	0.79	1,84737
Lipa80b	22.08	0,0758918	**Tho40**	6.52	0,0116882
Nug28	3.45	0,0058682	**Tho150**	2.18	1,14028
Sko81	2.1	0,141866	**Wil50**	1.7	0,0221609
Sko90	2.67	0,246332	**Wil100**	1.26	0,252535

Podemos observar que la técnica de *Don't look bits* mejora significativamente los tiempos de ejecución de la versión básica de la búsqueda local.



En el siguiente gráfico podemos ver como evoluciona el tiempo de ejecución en función del tamaño tanto de la Búsqueda Local (rojo) como de su variante con *Don't Look Bits* (azul).



Se puede ver como el tamaño del problema no es lo único que afecta al tiempo de ejecución; La dificultad del problema en si (que es mucho mas difícil de medir) influye también. Por ejemplo, para la búsqueda local básica, se tarda más en *Tai150b* de tamaño 150 (24,83 segundos) que en *Tai256c* de tamaño 256 (5,15 segundos).

8.4. Comparación de los resultados. Conclusiones.

Algoritmo	Desv	Tiempo
Greedy	61,08	0,00088787315
BL	10,66	2,80
BL. Don't Look Bits	8.26	0,33

Como podemos ver, el tiempo de ejecución del algoritmo greedy es varias magnitudes menor que los tiempos de los otros dos algoritmos. Sin embargo, la poca calidad de sus soluciones hace que, por si sola, no sea una herramienta idónea de cara a resolver problemas.

Tanto la Búsqueda Local básica como su variante con *Dont't look bits* nos ofrecen resultados mucho mas óptimos que el greedy.

Como la desviación de BL y BL con DLB es similar, debemos fijarnos en el tiempo medio de ejecución. La variante *DLB* es 8.5 veces más rápida[3] que la original. Aunque la variante *Don't Look Bits* ofrece resultados ligeramente mejores, esto no debe ser tomado como referencia, pues la naturaleza aleatoria de cara a generar la solución de partida hace que no podamos asegurar que uno encuentra siempre una solución mejor que la otra.

Cabe destacar que el experimento ha sido compilado sin optimización (***Debug***). La realidad es que con un nivel máximo de optimización (***Release***), aún en problemas de mayor tamaño y complejidad como ***Tai150b*** obtenemos unos tiempos mucho menores que hace que utilizar un greedy en producción sea aun menos viable.

Un ejemplo ilustrativo de la diferencia de tiempos en función del nivel de optimización:

Tai150b	**Nivel de Optimización**	
Algoritmo	**Debug**	**Release**
BL	24.174	1.55584
BL DLB	1.54554	0.105939

[3] $2{,}80/0{,}33 = 8.\overline{48}$