

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	3
2. Consideraciones comunes a los algoritmos utilizados	4
2.1. Clase Solver	5
2.2. Clase Solution	6
3. Algoritmo Greedy	7
4. Algoritmo de Búsqueda Local	8
4.1. Búsqueda Local con <i>Don't Look Bits</i>	10
5. Algoritmos Genéticos	11
5.1. Algoritmo Genético Generacional. <i>AGG</i>	18
5.2. Algoritmo Genético Estacionario. <i>AGE</i>	21
6. Algoritmos Meméticos	24
7. Procedimiento considerado para desarrollar la práctica	26
8. Experimentos y análisis de resultados	28
8.1. Resultados Greedy	29
8.2. Resultados Búsqueda Local	30
8.3. Resultados Búsqueda Local con <i>Don't Look Bits</i>	31
8.4. Resultados AGG con cruce basado en posición	33
8.5. Resultados AGG con cruce OX	34
8.6. Resultados AGE con cruce basado en posición	35
8.7. Resultados AGE con cruce OX	36
8.8. Resultados Memético (10, 1)	37
8.9. Resultados Memético (10, 0.1)	38
8.10. Resultados Memético (10, 0.1 mejores)	39
8.11. Comparación de los resultados. Conclusiones.	40

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:

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

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**.

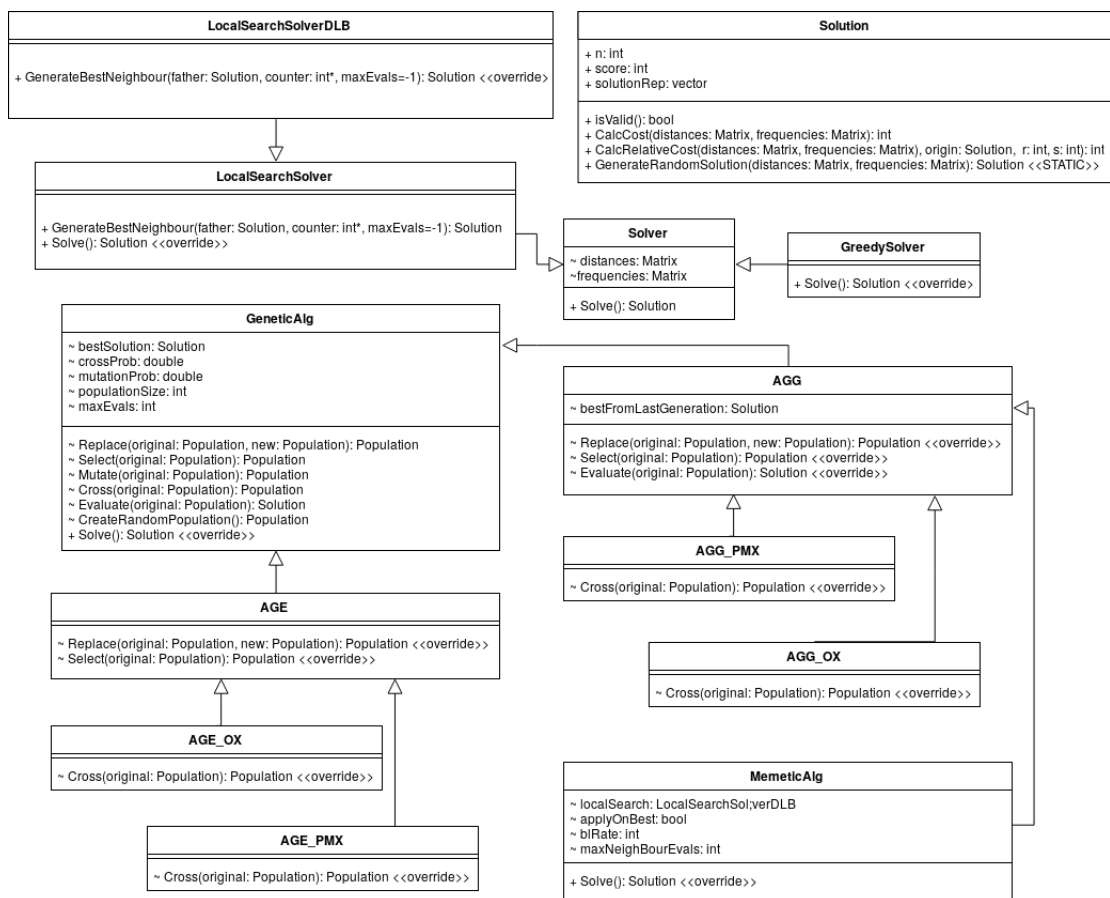


Diagrama de clases

2.1. Clase Solver

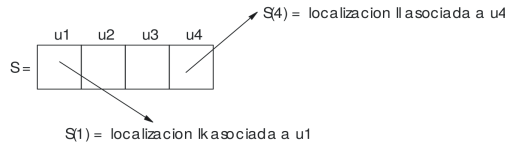
Esta clase debe ser heredada por las metaheurísticas a implementar. Su representación consta de dos matrices:

- **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:

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

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   $dp = fp = \text{vector}(n)$ 
3  for  $i = 1$  to  $n$  do
4       $\hat{f}_i = \hat{d}_i = 0$ 
5      for  $j = 0$  to  $n$  do
6           $\hat{f}_i = \hat{f}_i + f_{ij}$ 
7           $\hat{d}_i = \hat{d}_i + d_{ij}$ 
8      end
9       $dp_i = \hat{d}_i$ 
10      $fp_i = \hat{f}_i$ 
11 end
12 # Calcula la mejor combinación.  $\pi$  es la representación de la solución
13  $locAssigned = unitAssigned = \text{vector}(n)\{0\}$ 
14 for  $i = 1$  to  $n$  do
15      $best\hat{f} = -\infty$ ;  $best\hat{f}_{index} = 0$ 
16      $best\hat{d} = \infty$ ;  $best\hat{d}_{index} = 0$ 
17     for  $j = 0$  to  $n$  do
18          $\hat{f}_i = fp_j$ ;  $\hat{d}_i = dp_j$ 
19         if  $\hat{f}_i > best\hat{f}$  and  $unitAssigned_j \neq 1$  then
20              $best\hat{f} = \hat{f}_i$ ;  $best\hat{f}_{index} = j$ 
21         end
22         if  $\hat{d}_i < best\hat{d}$  and  $locAssigned_j \neq 1$  then
23              $best\hat{d} = \hat{d}_i$ ;  $best\hat{d}_{index} = j$ 
24         end
25     end
26      $\pi(best\hat{f}_{index}) = best\hat{d}_{index}$ 
27      $unitAssigned_{best\hat{f}_{index}} = locAssigned_{best\hat{d}_{index}} = 1$ 
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  $\pi = \text{GenerateInitialSolution}()$  # Será aleatoria
2 do
3    $\pi' = \text{GenerateBestNeighbour}(\pi)$ 
4   if  $\exists \pi'$  then  $\pi = \pi'$ ;
5 while  $\exists \pi'$ ;

```

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  $assigned = \text{vector}(n)\{0\}$ 
2 for  $i = 1$  to  $n$  do
3   do
4      $r = \text{random}() \bmod n$ 
5     while  $assigned_r \neq 0$ ;
6      $\pi(i) = r$ 
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 por el de s se define como:

$$\Delta C(\pi, r, s) = \sum_{k=1, k \neq r, s}^n \left[f_{rk}(d_{\pi(s)\pi(k)} - d_{\pi(r)\pi(k)}) + f_{sk}(d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)}) + f_{kr}(d_{\pi(k)\pi(s)} - d_{\pi(k)\pi(r)}) + f_{ks}(d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)}) \right]$$

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( $\pi$ ):
2   for  $r = 1$  to  $n/2$  do
3     for  $s = r + 1$  to  $n$  do
4       if  $\Delta C(\pi, r, s) < 0$  then
5          $\pi' = \pi$ 
6          $t = \pi'(r)$ 
7          $\pi'(r) = \pi'(s)$ 
8          $\pi'(s) = t$ 
9         return  $\pi'$ 
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 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( $\pi$ ):
3     for r = 1 to n do
4         if dlbMaskr ≠ 0 then continue;
5         for s = 1 to n do
6             if  $\Delta C(\pi, r, s) < 0$  then
7                  $\pi' = \pi$ 
8                  $t = \pi'(r)$ 
9                  $\pi'(r) = \pi'(s)$ 
10                 $\pi'(s) = t$ 
11                dlbMaskr = dlbMasks = 0
12                return  $\pi'$ 
13            end
14        end
15        dlbMaskr = 1
16    end
17 end

```

5. Algoritmos Genéticos

Se han implementado dos tipos de algoritmos 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 **Reemplazo** 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 algoritmo 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  $n$  cromosomas
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    $P = \text{GenerateRandomPopulation}()$   
3    $\pi = \text{Evaluate}(P)$   
4    $\pi_{best} = \pi$   
5   while evaluaciones  $\neq$  limite do  
6      $P' = \text{Select}(P)$   
7      $P' = \text{Cross}(P')$   
8      $P' = \text{Mutate}(P')$   
9      $P' = \text{Replace}(P, P')$   
10     $\pi = \text{Evaluate}(P)$   
11    if  $\pi$  mejor que  $\pi_{best}$  then  
12       $\pi_{best} = \pi$   
13    end  
14  end  
15  return  $\pi_{best}$   
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 contiguos desde un cromosoma aleatorio de nuestra población.

```

1 def Mutate(P):
2     P' = P
3     n_mutar = ⌈n · p_mutar · len(P₀)⌉
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        evaluaciones++
18    end
19    return P'
20 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    $best\_score = \infty$ 
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       |    $best\_score = \text{coste } P_i$ 
10      |    $\pi_{best} = P_i$ 
11    end
12  end
13  return  $\pi_{best}$ 
14 end

```

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

Operador de Cruce

Gracias a este operador, podremos simular 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:

$$\begin{aligned}
 \text{Padre}_1 &= (1, 2, 3, 4, 5, 6, 7, 8, 9) & \text{Padre}_2 &= (4, 5, 3, 1, 8, 7, 6, 9, 2) \\
 \text{coincidencias} &= (*, *, 3, *, *, 7, 6, *, *) \\
 \text{resto_Padre}_1 &= (1, 2, 4, 5, 8, 9) \rightarrow \text{Barajamos} : (9, 1, 2, 4, 8, 5) \\
 \text{resto_Padre}_2 &= (4, 5, 1, 8, 9, 2) \rightarrow \text{Barajamos} : (5, 1, 4, 2, 9, 8) \\
 \text{hijo}_1 &= (9, 1, 3, 2, 4, 7, 6, 8, 5) & \text{hijo}_2 &= (5, 1, 3, 4, 2, 7, 6, 9, 8)
 \end{aligned}$$

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(Pi)){0};  nonEquals = ∅
5          for j = 1 to len(Pi) do
6              if Pi(j) == Pi+1(j) then
7                  π'(j) = π''(j) = Pi(j)
8                  equalsj = 1
9              end
10             else
11                 nonEquals = nonEquals ∪ Pi(j)
12             end
13         end
14         shuffle(nonEquals)
15         k = 0
16         for j = 1 to len(Pi) do
17             if equalsj ≠ 1 then
18                 π'(j) = nonEqualsk
19                 π''(j) = nonEqualslen(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 un tercio de los valores de los padres. El resto de elementos que no están en la parte central, se ordenan según el otro padre.

$$\begin{aligned}
 \text{Padre}_1 &= (7, 3, 1, 8, 2, 4, 6, 5) & \text{Padre}_2 &= (4, 3, 2, 8, 6, 7, 1, 5) \\
 \text{comparte_padre}_1 &= (*, *, 1, 8, 2, *, *, *) & \text{comparte_padre}_2 &= (*, *, 2, 8, 6, *, *, *) \\
 \text{resto_Padre}_1 &= (7, 3, 4, 6, 5) \rightarrow \text{Ordenamos según Padre}_2 : (4, 3, 6, 7, 5) \\
 \text{resto_Padre}_2 &= (4, 3, 7, 1, 5) \rightarrow \text{Ordenamos según Padre}_1 : (7, 3, 1, 4, 5) \\
 \text{hijo}_1 &= (7, 5, 1, 8, 2, 4, 3, 6) & \text{hijo}_2 &= (4, 5, 2, 8, 6, 7, 3, 1)
 \end{aligned}$$

La implementación más 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 descritos arriba. En el primer set, meteremos los valores externos a la parte compartida 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:

```

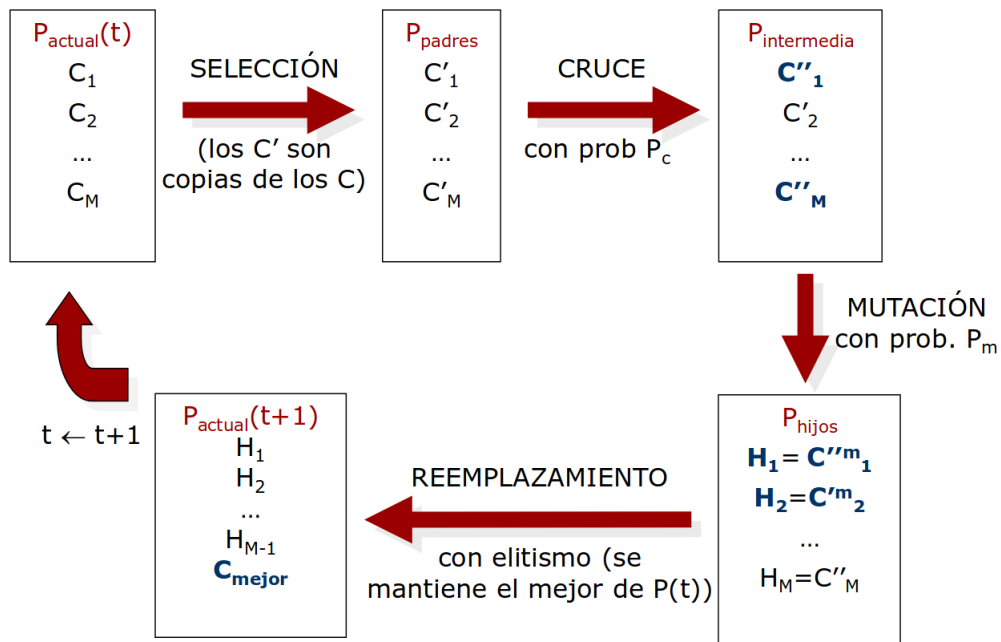
1 def CrossOX(P):
2    $P' = P$ ;  $n_{cruce} = \lceil n \cdot p_{cruce} \rceil$ ;  $step = len(P_0)/3$ 
3    $start = \text{random entre } (0, n - \lfloor step \rfloor)$ 
4    $end = start + \lceil step \rceil$ 
5   for  $i = 1$  to  $n$  do
6      $S1 = \text{vector}(len(P_i))$ ;  $S2 = \text{vector}(len(P_{i+1}))$ 
7     for  $j = 1$  to  $len(P_i)$  do
8        $S1_{P_{i_j}} = j$ ;  $S2_{P_{i+1_j}} = j$ 
9       if  $j \geq start$  and  $j \leq end$  then
10        |  $\pi'_j = P_{i_j}$ ;  $\pi''_j = P_{i+1_j}$ 
11      end
12    end
13    # Ordena de menor a mayor con S1 y S2
14     $orderedByS1 = \text{set}()$ ;  $orderedByS2 = \text{set}()$ 
15    for  $j = 1$  to  $len(P_i)$  do
16      if  $j == start$  then
17        |  $j = end$ ; continue
18      end
19       $orderedByS1 = orderedByS1 \cup P_{i+1_j}$ 
20       $orderedByS2 = orderedByS2 \cup P_{i_j}$ 
21    end
22     $k = 0$  for  $j = end + 1$  ;  $j \neq start$  ;  $j = (j + 1) \bmod len(P_i)$  do
23      if  $j == start$  then
24        |  $j = end$ ; continue
25      end
26       $\pi'_j = orderedByS2_k$ ;  $\pi''_j = orderedByS1_k$   $k = k + 1$ 
27    end
28     $P'_i = \pi'$ ;  $P'_{i+1} = \pi''$   $i = i + 2$ 
29  end
30  return  $P'$ 
31 end

```

Algorithm 9: *genetic.cpp* - CrossOX

5.1. Algoritmo 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  $P_{r1} \leq P_{r2}$  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 reemplaza la anterior población por la nueva reemplazando la peor solución de esta por la mejor de la generación anterior. El pseudocódigo que implementa dicha funcionalidad es:

```

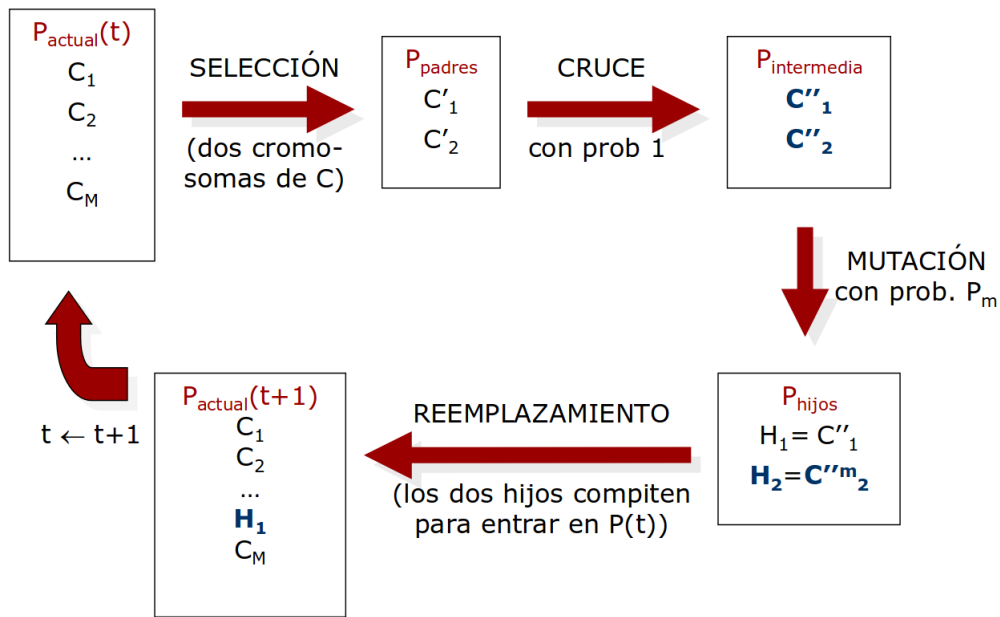
1 def Replace( $P, P'$ ):
2    $P'' = P'$ 
3    $peor\_coste = peor\_indice = 0$ 
4   if  $\exists \pi_{best}$  then
5     for  $i = 1$  to  $n$  do
6       if  $coste\ P''_i$  desconocido then
7         Calcula  $coste\ P''_i$ 
8          $evaluaciones++$ 
9       end
10      if  $peor\_coste < coste\ P''_i$  then
11         $peor\_coste = coste\ P''_i$ 
12         $peor\_indice = i$ 
13      end
14    end
15     $P''_{peor\_indice} = \pi_{bestAnterior}$ 
16  end
17  return  $P''$ 
18 end

```

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

5.2. Algoritmo 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 reemplazo. 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  $P_{r1} \leq P_{r2}$  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 rempazan 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  $peores == \emptyset$  or  $coste P''_i > coste peores_0$  then
10      |   if  $len(peores) \geq len(P')$  then
11      |   |   Elimina  $peores_0$  de  $peores$ 
12      |   end
13      |    $peores = peores \cup P''_i$ 
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. Algoritmos Meméticos

Partiremos de un AGG con operador de cruce basado en posición sobre el cual, pasado un determinado número de iteraciones (blN), ejecutaremos una búsqueda local sobre un número de elementos calculado a partir de una probabilidad P_{bl} y el tamaño de la población como:

$$blExecN = \lceil P_{bl} \cdot n \rceil$$

La búsqueda local se detiene cuando sobre un cromosoma no se ha encontrado un vecino mejor, cuando se explora todo el entorno sin encontrar una solución mejor, o bien, cuando se alcanza un número máximo de evaluaciones sobre vecinos.

Gracias a esta hibridación de técnicas, tenemos la ventaja de alcanzar rápidamente un óptimo local gracias a la búsqueda local, y la capacidad de aceptar soluciones peores como válidas a fin de evitar quedarnos atrapados en óptimos locales.

Además, podemos elegir si aplicar la búsqueda local sobre los mejores de la población.

La única función que cambia respecto a un algoritmo genético es el método de búsqueda, cuya implementación es la siguiente:

```

1 def Solve():
2    $P = \text{GenerateRandomPopulation}()$ 
3    $\pi = \text{Evaluate}(P)$ ;  $\pi_{best} = \pi$ ;  $generations = 0$ 
4   while  $evaluaciones \neq limite$  do
5      $P' = \text{Select}(P)$ 
6      $P' = \text{Cross}(P')$ 
7      $P' = \text{Mutate}(P')$ 
8      $P' = \text{Replace}(P, P')$ 
9     if  $generations == blN$  then
10       $generations = 0$ 
11      if aplicar en los mejores then
12         $P'' = P$ 
13        Ordena los cromosomas de  $P''$  con el coste de cada uno
14        for  $i = 0$  to  $blExecN$  do
15           $\pi' = \text{BusquedaLocalSobre}(P''_i)$ 
16          if  $\exists \pi'$  then  $P''_i = \pi'$  ;
17        end
18        Reemplaza los modificados en  $P''$  en  $P'$ 
19      end
20      else
21        for  $i = 0$  to  $blExecN$  do
22           $\pi' = \text{BusquedaLocalSobre}(P'_i)$ 
23          if  $\exists \pi'$  then  $P'_i = \pi'$  ;
24        end
25      end
26    end
27     $\pi = \text{Evaluate}(P)$ 
28    if  $\pi$  mejor que  $\pi_{best}$  then  $\pi_{best} = \pi$  ;
29  end
30  return  $\pi_{best}$ 
31 end

```

Algorithm 14: *memetic.cpp* - MemeticAlg::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
│   └── practical ..... 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 ..

```

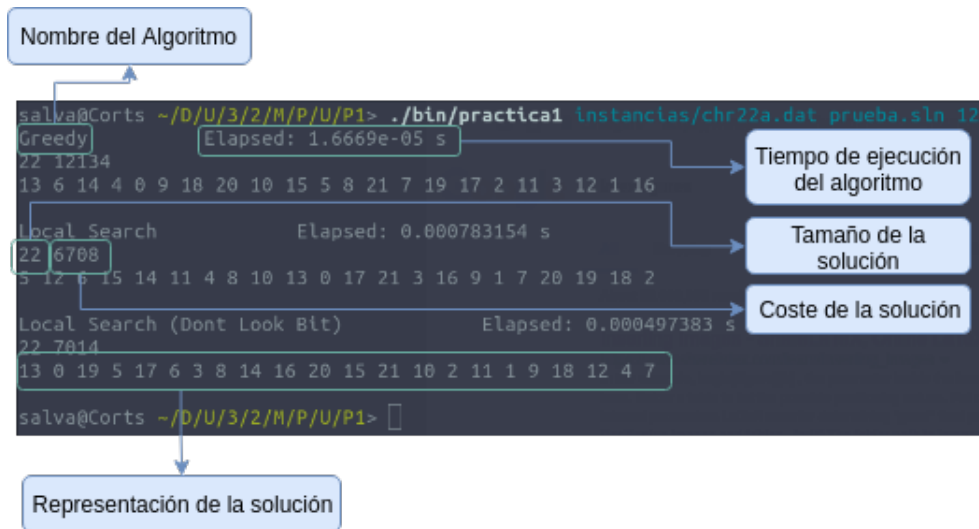
La sintaxis de ejecución es la siguiente¹:

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

Por ejemplo:

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

La salida del programa tiene la siguiente estructura:



¹El 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.

$$\frac{1}{|\text{casos}|} \sum_{i=1}^{\text{casos}} 100 \frac{\text{valoralgoritmo}_i - \text{mejorValor}_i}{\text{mejorValor}_i}$$

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 entre los algoritmos greedy y búsqueda local.*
- *Los tiempos para los algoritmos genéticos y meméticos se han tomado con compilación **Release**, por lo tanto, se comparará con el tiempo medio del greedy y búsqueda local compilados en **Release**.*

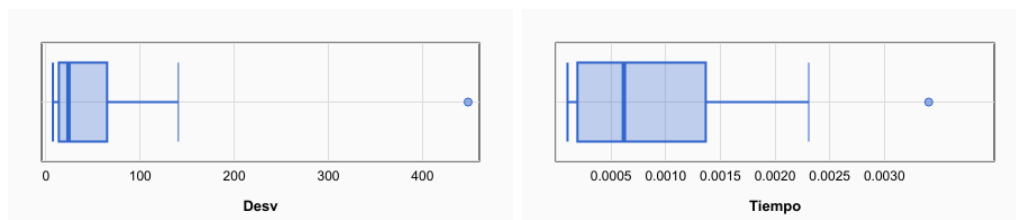
²<http://anjos.mgi.polymtl.ca/qaplib//inst.html>

8.1. Resultados Greedy

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.

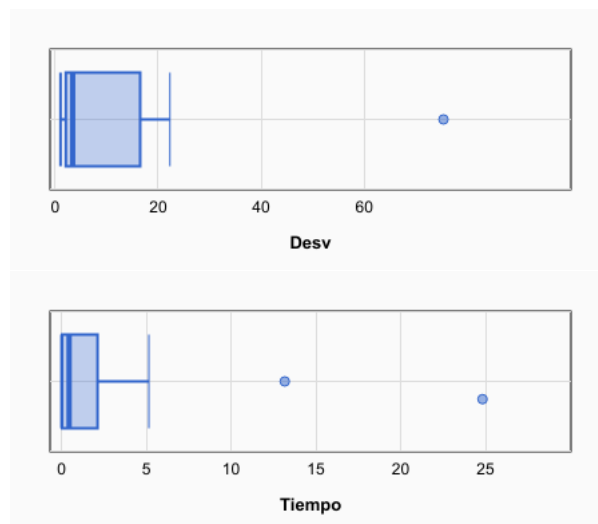


8.2. Resultados Búsqueda Local

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.

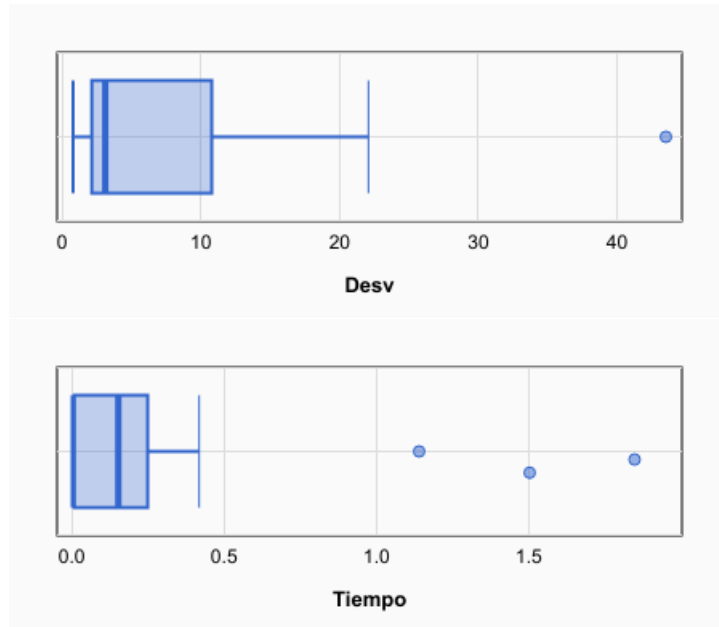
Sin embargo, en combinación con otros algoritmos como los Genéticos podemos obtener resultados aún mejores y en un tiempo mas aceptable dado que restringimos el espacio de búsqueda para este algoritmo.



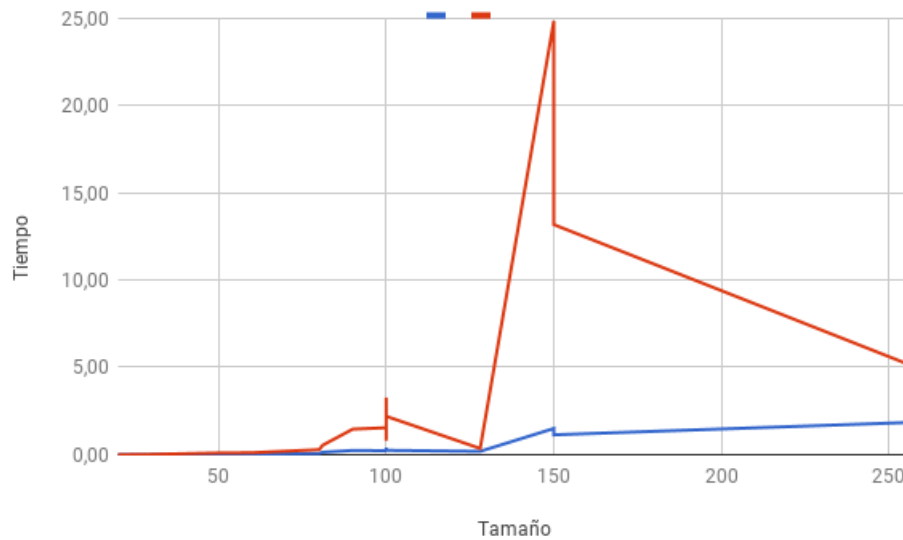
8.3. Resultados Búsqueda Local con *Don't Look Bits*

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).

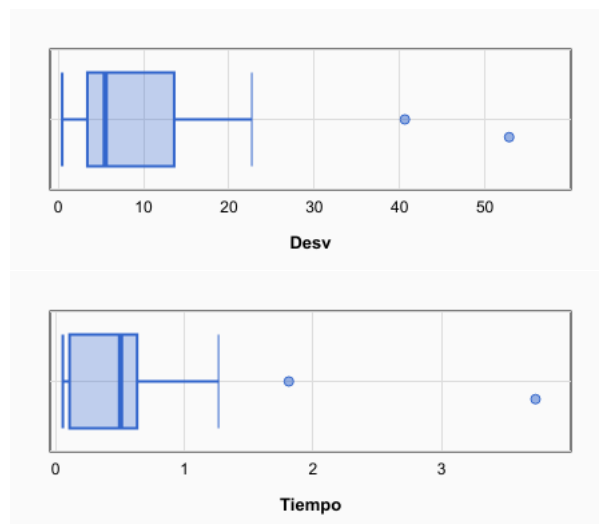


Se puede ver como el tamaño del problema no es lo único que afecta al tiempo de ejecución; La dificultad del problema en si (que es mucho mas difícil de medir) influye también. Por ejemplo, para la búsqueda local básica, se tarda más en *Tai150b* de tamaño 150 (24,83 segundos) que en *Tai256c* de tamaño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

8.4. Resultados AGG con cruce basado en posición

Algoritmo AGG-Posición					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	11,86	0,06	Sko100a	3,62	0,61
Chr22b	14,37	0,06	Sko100f	3,36	0,63
Chr25a	52,85	0,08	Tai100a	5,09	0,60
Esc128	40,63	0,95	Tai100b	10,52	0,63
Had20	0,46	0,06	Tai150b	12,81	1,81
Lipa60b	20,94	0,24	Tai256c	3,63	3,73
Lipa80b	22,68	0,42	Tho40	2,75	0,14
Nug28	5,92	0,08	Tho150	7,22	1,27
Sko81	4,25	0,40	Wil50	2,31	0,20
Sko90	3,47	0,64	Wil100	2,30	0,59

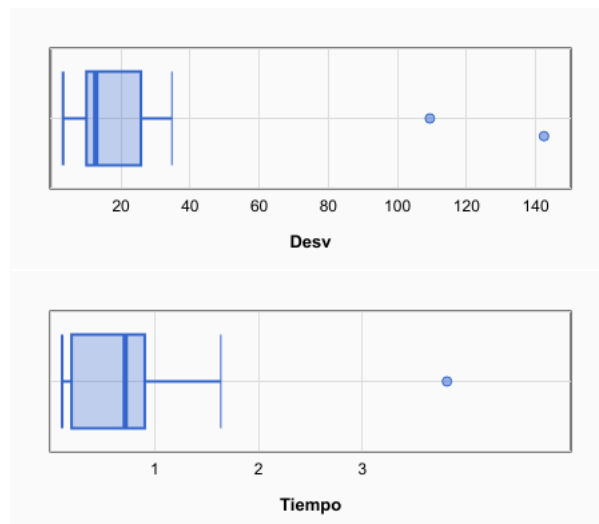
Rápidamente obtenemos soluciones bastante buenas a nuestro problema. Aunque en las primeras iteraciones, este tipo de cruce genera hijos que se diferencian mas de los padres que los generados por el cruce OX, poco a poco, al ser un modelo elitista, la población converge hasta el punto de que muchos padres son iguales y por lo tanto, debido al funcionamiento de este tipo de cruce, los hijos son iguales. Todo esto hace que para pocas evaluaciones, este cruce actúe mejor que OX, pero un número lo suficientemente elevado de evaluaciones, el cruce basado en posición hará que la población se estanque y OX funcione mejor.



8.5. Resultados AGG con cruce OX

Algoritmo AGG-OX					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	34,73	0,11	Sko100a	10,21	0,90
Chr22b	34,42	0,11	Sko100f	11,52	0,89
Chr25a	142,41	0,13	Tai100a	9,81	0,88
Esc128	109,38	1,42	Tai100b	25,56	0,91
Had20	3,24	0,13	Tai150b	19,43	1,64
Lipa60b	25,54	0,48	Tai256c	4,85	3,81
Lipa80b	25,98	0,66	Tho40	16,97	0,25
Nug28	11,73	0,15	Tho150	13,65	1,63
Sko81	10,78	0,68	Wil50	7,32	0,33
Sko90	11,14	0,76	Wil100	6,50	0,89

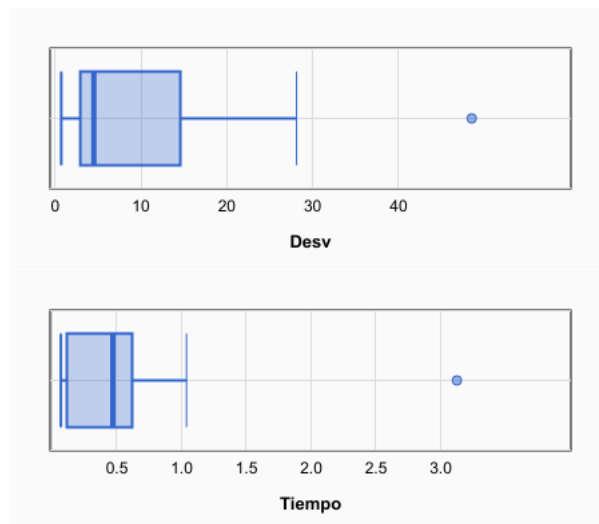
Como hemos comentado antes, para un número de evaluaciones máximas pequeño, OX funciona peor que el cruce basado en posición. Sin embargo, tarda más en converger.



8.6. Resultados AGE con cruce basado en posición

Algoritmo AGE-Posición					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	14,78	0,09	Sko100a	3,07	0,63
Chr22b	14,40	0,07	Sko100f	3,63	0,60
Chr25a	48,52	0,08	Tai100a	4,57	0,51
Esc128	28,13	0,86	Tai100b	10,50	0,51
Had20	0,72	0,08	Tai150b	6,42	1,02
Lipa60b	20,38	0,28	Tai256c	1,98	3,13
Lipa80b	21,90	0,46	Tho40	5,08	0,13
Nug28	3,87	0,10	Tho150	4,48	1,04
Sko81	3,28	0,37	Wil50	2,46	0,18
Sko90	2,81	0,48	Wil100	1,23	0,49

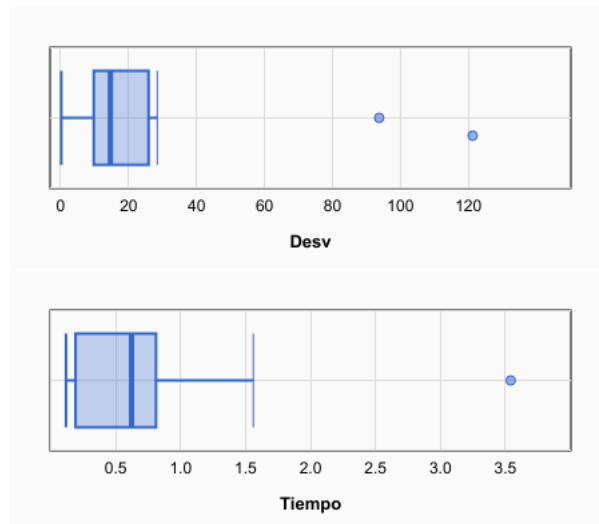
Obtenemos mejores resultados y tiempos de media que AGG con cruce basado en posición. Debido a que escogemos dos individuos de la población original al azar, los cruzamos, los mutamos y en caso de que sean mejores que los peores de la población original los remplazamos; mejoramos sustancialmente la población en cada iteración o la dejamos como estaba; a diferencia de AGG que al remplazar la población original por la nueva, puede empeorar fácilmente la población original.



8.7. Resultados AGE con cruce OX

Algoritmo AGE-OX					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	25,41	0,12	Sko100a	12,82	0,75
Chr22b	16,73	0,12	Sko100f	12,51	0,73
Chr25a	121,29	0,12	Tai100a	11,77	0,80
Esc128	93,75	1,33	Tai100b	28,43	0,82
Had20	0,17	0,12	Tai150b	23,62	1,56
Lipa60b	26,33	0,41	Tai256c	3,76	3,54
Lipa80b	27,79	0,59	Tho40	22,94	0,25
Nug28	7,70	0,14	Tho150	15,74	1,38
Sko81	13,50	0,58	Wil50	7,70	0,31
Sko90	13,54	0,66	Wil100	7,35	0,77

De nuevo, por las mismas razones que en apartado anterior, obtenemos mejores resultados en mejor tiempo que AGG OX. No obstante, debido a que el número de iteraciones es bajo, el cruce OX se comporta peor que el operador de cruce basado en posición.

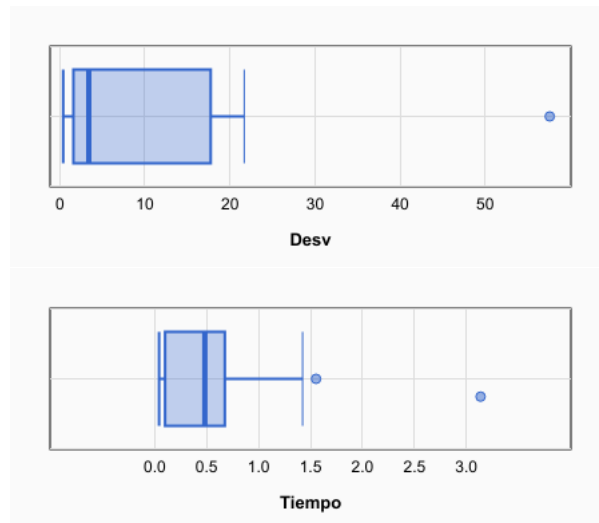


8.8. Resultados Memético (10, 1)

Algoritmo AM(10, 1)					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	16,73	0,07	Sko100a	2,24	0,60
Chr22b	19,66	0,06	Sko100f	1,60	0,62
Chr25a	57,59	0,07	Tai100a	3,63	0,58
Esc128	18,75	0,78	Tai100b	3,98	0,73
Had20	0,40	0,04	Tai150b	3,19	1,55
Lipa60b	20,49	0,25	Tai256c	0,64	3,14
Lipa80b	21,68	0,39	Tho40	3,78	0,13
Nug28	6,85	0,07	Tho150	2,96	1,42
Sko81	2,20	0,44	Wil50	0,79	0,18
Sko90	1,61	0,52	Wil100	0,72	0,58

Al aplicar una búsqueda local sobre toda la población, en pocas iteraciones hemos mejorado mucho la calidad de nuestra población. Esto hace que los resultados sean mucho mejores que en AGG.

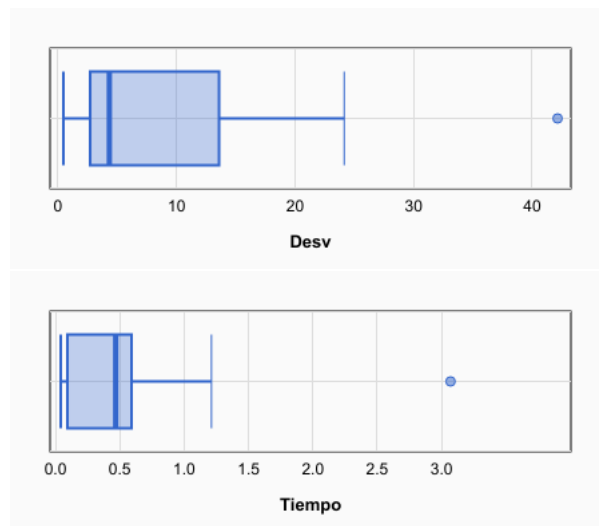
Aplicar una búsqueda local sobre cada cromosoma es muy costoso, por lo tanto, para un mismo tamaño de población, un algoritmo memético tardará mas en completar una iteración que un Genético. Sin embargo, el tamaño de la población en nuestro experimento es 10, frente a los 50 del Genético por lo que no podemos apreciar este detalle.



8.9. Resultados Memético (10, 0.1)

Algoritmo AM(10, 0.1)					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	11,57	0,05	Sko100a	2,00	0,61
Chr22b	24,15	0,05	Sko100f	2,11	0,55
Chr25a	42,15	0,06	Tai100a	4,44	0,55
Esc128	15,63	0,77	Tai100b	4,99	0,58
Had20	3,09	0,04	Tai150b	4,24	1,21
Lipa60b	19,52	0,24	Tai256c	0,47	3,07
Lipa80b	22,46	0,39	Tho40	5,42	0,12
Nug28	7,47	0,06	Tho150	2,91	1,16
Sko81	3,30	0,42	Wil50	2,53	0,19
Sko90	3,32	0,52	Wil100	0,77	0,56

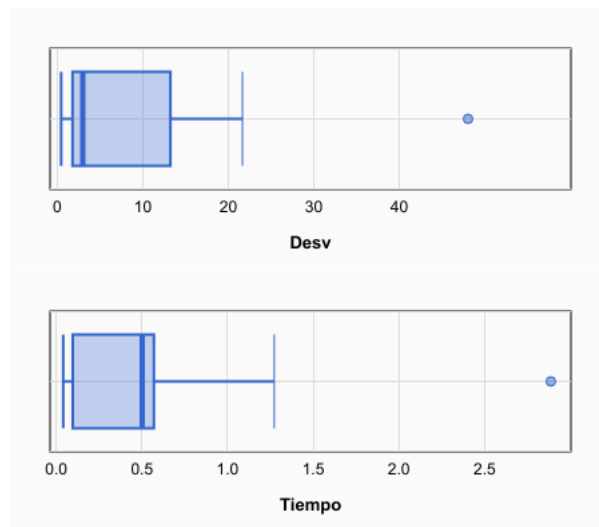
Al aplicar la búsqueda local sobre un porcentaje (en este caso 10 %) de la población mejoramos sustancialmente los resultados respecto a aplicar la búsqueda sobre todos los cromosomas. Esto se debe a que mantenemos mayor diversidad cuando ejecutamos la búsqueda local sobre solo un porcentaje.



8.10. Resultados Memético (10, 0.1 mejores)

Algoritmo AM(10, 0.1mej)					
Caso	Desv	Tiempo	Caso	Desv	Tiempo
Chr22a	18,84	0,05	Sko100a	1,52	0,58
Chr22b	13,98	0,05	Sko100f	2,11	0,57
Chr25a	48,05	0,05	Tai100a	4,16	0,55
Esc128	12,50	0,92	Tai100b	1,80	0,56
Had20	0,46	0,04	Tai150b	3,32	1,27
Lipa60b	20,45	0,27	Tai256c	0,72	2,89
Lipa80b	21,67	0,50	Tho40	4,18	0,11
Nug28	5,46	0,08	Tho150	2,66	1,16
Sko81	2,54	0,46	Wil50	2,19	0,18
Sko90	1,79	0,51	Wil100	1,01	0,52

De nuevo mejoramos aún mas los resultados obtenidos ya que mantenemos la diversidad pero mejoramos los mejores cromosomas aumentando rapidamente la calidad de nuestra población.



8.11. 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 *Don'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 rápida³ 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:

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

³ $2,80/0,33 = 8.48$

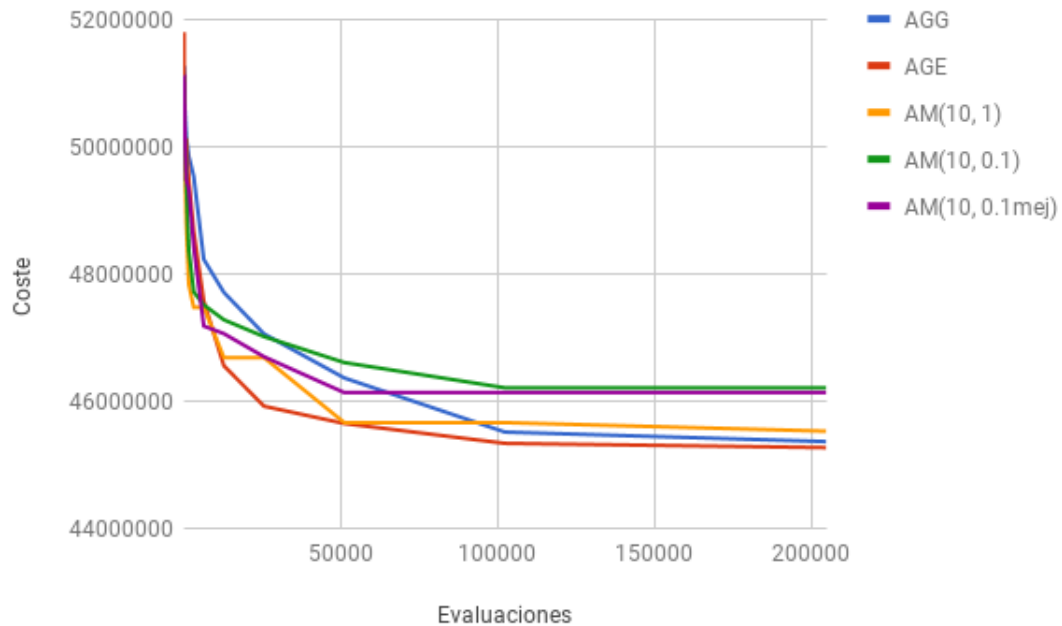
Algoritmo	Desv	Tiempo
Greedy	61.08	0,0000609593
BL	10.66	0.24
BL. DLB	8.26	0,03
AGG. Posición	11.55	0,66
AGG. OX	26,76	0,84
AGE. Posición	10,11	0,56
AGE. OX	24,64	0,76
AM (10, 1)	9,48	0,61
AM (10, 0.1)	9.13	0,56
AM (10, 0.1 mejores)	8,47	0,57

Como podemos ver, aunque obtenemos resultados bastante buenos, ninguno mejora la búsqueda local. Si bien los algoritmos genéticos y los meméticos son planteamientos mucho más sofisticados que la búsqueda local, necesitan ser ajustados correctamente para su correcto funcionamiento:

- En el caso de los genéticos, podemos jugar con el tamaño de la población y el número máximo de evaluaciones.
- En el caso de los meméticos, podemos ajustar los mismo parámetros que en los genéticos y, además, cada cuanto ejecutar la búsqueda local y sobre que cromosomas.

Hay que tener muy en cuenta el tiempo a la hora de ajustar estos parámetros pues por ejemplo ejecutar una búsqueda local sobre todos los cromosomas de una población de tamaño 100 es muy costoso y podemos conseguir el mismo resultado calibrando el porcentaje sobre los que ejecutar la búsqueda y el número máximo de evaluaciones.

En el siguiente gráfico, podemos ver como evolucionan los algoritmos en función del número máximo de evaluaciones. Sobra decir que a mayor límite de evaluaciones mayor tiempo se necesita.



Como vemos a mayor número de evaluaciones máximas, mejores resultados. Con un número de evaluaciones máximas lo suficientemente grandes, los meméticos funcionan mejor que los genéticos pero se quedan estancados. Esto hace que para un número de evaluaciones aún mayor, los AGE obtengan (en un tiempo menor) mejores resultados que los AGG y parecidos al memético que aplica BL sobre toda la población.

Como conclusión final sobre los algoritmos genéticos y meméticos, cabe destacar que para el problema de QAP, los resultados que obtenemos con los genéticos al incrementar el número de evaluaciones máximas respecto del tiempo no son buenas. Esto se debe a que estos algoritmos no se comportan bien con codificaciones de orden. Aunque localizan zonas prometedoras, no son capaces de explorarlas bien; es por esto que los meméticos permiten obtener los mejores resultados en estos problemas. Un memético bien afinado, con un número suficiente de evaluaciones máximas y un buen equilibrio con la búsqueda local es una solución muy buena para este tipo de problemas.