Metaheurísticas (2016-2017)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 1.A: Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Asignación Cuadrática

---

Germán González Almagro

2 de julio de 2017

DNI: 76593910T
Correo Electrónico: germang.almagro@correo.ugr.es
Subgrupo de prácticas 3. Horario: Lunes de 5:30 a 7:30

Algoritmos Considerados: Algoritmo Voraz, Búsqueda Local, Algoritmo Genético Generacional con operadores de cruce de Posición y OX, Algoritmo Genético Estacionario con operadores de cruce de Posición y OX, Algoritmo Memético de mejora completa, Algoritmo Memético de mejora parcial aleatoria, Algoritmo Memético de mejora elitista.

Índice

Índice de figuras

Índice de cuadros

1. Descripción del problema

El problema de la asignación cuadrática o QAP (Quadratic Assignement Problem) es un problema de optimización combinatoria que pertenece a la clase de problemas NP-Completos. Consiste en determinar la asignación óptima de n unidades funcionales a n localizaciones, conociendo el flujo que circula entre cada unidad funcional y la distancia entre las localizaciones.

Para representar este problema de forma que sea computable, consideraremos las matrices F y D, que almacenarán el flujo entre unidades y la distancia entre localizaciones respectivamente. De esta manera el flujo entre las unidades i y j es F_{ij} y la distancia entre las localizaciones k y l es D_{kl}; el coste de asignar la unidad i a la localización k y la unidad j a la localización l es $F_{ij} \times D_{kl}$.

Conociendo la representación del problema, podemos plantearlo como:

$$QAP = min\left(\sum_{i=1}^{n}\sum_{j=1}^{n} F_{ij} \times D_{S(i)S(j)}\right) \; t.q. \; S \in \prod_{N}$$

2. Métodos de resolución del problema

Dado que es computacionalmente costoso resolver este problema de forma óptima mediante algoritmos puramente deterministas, emplearemos diversas técnicas heurísticas para obtener una solución que, aunque no es óptima, es de calidad. Las técnicas que emplearemos para ello serán técnicas basadas en poblaciones que evolucionarán de distinta manera, a saber: Algoritmo Genético Generacional con cruce basado en Posición, Algoritmo Genético Generacional con cruce OX, Algoritmo Genético Estacionario con cruce basado en Posición, Algoritmo Genético Estacionario con cruce basado en OX, Algoritmo Memético de mejora completa, Algoritmo Memético de mejora parcial aleatoria, Algoritmo Memético de mejora parcial elitista.

Abordaremos también el problema con técnicas más simples, como pueden ser la Búsqueda Local o la resolución mediante algoritmo Greedy.

Aunque las diferentes técnicas de resolución presentan diferencias significativas, también presentan similitudes; una de ellas es la representación del problema. Las soluciones al problema estarán representadas por una estructura que contendrá un vector que almacena la permutación propuesta como solución, en el que los índices representan las unidades funcionales y el contenido las localizaciones asociadas, además contendrá un dato de tipo `int` que representa el valor asociado a la permutación propuesta. Entenderemos por dimensión del problema del número de unidades funcionales o localizaciones que lo conforman.
De igual forma que las diferentes técnicas de resolución comparte el modelo de representación, los algoritmos que las implementan utilizan procesos comunes que pueden ser definidos de forma modular como sigue:

2.1. Función objetivo

La función objetivo es la encargada de dar valor al campo de tipo `int` asociado a una solución, para ello, recorre la matriz de distancias y de flujo realizando las operaciones pertinentes sobre un acumulador.

Input: Dada una solcuión S
function $CalcularSolucionNumerica(\&S)$ **begin**
  $SNumerica \leftarrow 0$
  for $i \leftarrow 0$ **to** $Dimension$ **do**
    for $j \leftarrow 0$ **to** $Dimension$ **do**
      $SNumerica \leftarrow SNumerica + MatrizFlujo_{ij} \times$
      $MatrizDistancias_{S_i,S_j}$
    end
  end
  $S.SolucionNumerica \leftarrow SNumerica$
end

2.2. Factorización de la función objetivo

Es posible reducir el esfuerzo computacional realizado al evaluar un individuo de forma que, conociendo el estado del mismo, a saber, permutación que almacena y valor numérico de la misma, factorizamos el cálculo del coste de forma que el orden de complejidad del algoritmo que lo calcula se reduce de $\mathcal{O}(n^2)$ a $\mathcal{O}(n)$.

Input: Dados s y r las posiciones a intercambiar y π la solución
function $ChequearMovimiento(s,\ r,\ \pi)$ **begin**
  $Incremento \leftarrow 0$
  for $k \leftarrow 0$ **to** $Dimension$ **do**
    if $k \neq s$ && $k \neq r$ **then**
      $Incremento \leftarrow Incremento + F_{rk}(D_{\pi(s)\pi(k)} - D_{\pi(r)\pi(k)}) +$
      $F_{sk}(D_{\pi(r)\pi(k)} - D_{\pi(s)\pi(k)}) +$
      $F_{kr}(D_{\pi(k)\pi(s)} - D_{\pi(k)\pi(r)}) +$
      $F_{ks}(D_{\pi(k)\pi(r)} - D_{\pi(k)\pi(s)})$
    end
  end
  return $Incremento$
end

Dado que, tanto el orden de complejidad como el espacio de trabajo de la función objetivo y su versión factorizada no es el mismo, es lógico pensar que no deben contabilizar de igual forma para la obtención del número total de llamadas a la función objetivo que realiza un algoritmo. Como ya hemos visto, mientras que le orden de complejidad de la función objetivo es $\mathcal{O}(n^2)$, el de su versión factorizada es de orden $\mathcal{O}(n)$, concretamente $\mathcal{O}(4n)$, por tanto para completar una llamada a la función objetivo empleando únicamente llamadas a la función factorizada será necesario realizar $\frac{N}{4}$ llamadas a la misma.

2.3. Operador de mutación y generación de vecinos

Dado que el vecindario de una solución está compuesto por todas aquellas soluciones que presenten diferencia únicamente en dos de los índices asociados a la permutación que las conforman, es sencillo definir el operador que genera un vecino a partir de una solución. Basta con hacer uso de la función objetivo factorizada, para obtener el incremento o decremento del valor numérico de la solución, para poder así sumarlo al valor numérico del vecino generado, e intercambiar las dos posiciones que dan lugar a esta modificación del valor.

function $GenerarVecino/Mutar(i,\ j,\ \&Solucion)$ **begin**
  $Incremento \leftarrow CheckMove(i,\ j,\ Solucion)$
  $Solucion.IntercambiarIndices(i,\ j)$
  $Solucion.Valor \leftarrow Solucion.Valor + Incremento$
end

2.4. Generación de soluciones aleatorias

Para trabajar con algoritmos basados en poblaciones es necesario contar con una población inicial que, en este caso, estará formada por soluciones factibles aleatorias generadas siguiendo el proceso descrito en el pseudocódigo.

function $GenerarSolucionAleatoria(\&Solucion)$ **begin**
  $Solucion \leftarrow \emptyset$
  for $i \leftarrow 0$ **to** $Dimension$ **do**
    $Solucion[i] \leftarrow i$
  end
  `#Barajamos el vector que contiene la permutación solución`
  for $i \leftarrow 0$ **to** $Dimension$ **do**
    $Rand \leftarrow AleatorioEntre(i,\ Dimension)$
    $Solucion.Intercambiar(Rand,\ i)$
  end
  $CalcularSolucionNumerica(Solucion)$
end

2.5. Operador de mutación

El operador de mutación es el encargado de introducir diversidad en la población, basta con seleccionar de forma aleatoria cromosomas de la población así como los dos genes que se intercambiarán para hacer efectiva la mutación. Cabe destacar que, el número de mutaciones se calcula como: $NumMutaciones = ProbabilidadMutacion \times NumeroCromosomas \times GenesPorCromosoma$, de esta forma obtenemos el número de genes que mutan, y, puesto que en cada individuo mutan dos genes, el número de individuos que mutan es: $NumeroMutaciones/2$.
Cabe destacar que, como veremos mas adelante, el proceso de mutación es el mismo en los algoritmos genéticos estacionarios y generacionales, aunque no la forma en la que esta se aplica.

```
function AplicarMutacion(&Poblacion, NumMutaciones) begin
    for i ← 0 to NumeroMutaciones/2 do
        Crma ← Poblacion[AleatorioEntre(0, TamPoblacion − 1)]
        #Gen1 y Gen2 deben ser diferentes
        Gen1 ← AleatorioEntre(0, Dimension − 1)
        Gen2 ← AleatorioEntre(0, Dimension − 1)
        Mutar(Gen1, Gen2, Crma)
    end
end
```

2.6. Operador de selección en los Algoritmos Genéticos

El mecanismo de selección empleado es el torneo binario, es decir, se seleccionan dos parejas de individuos de la población en las que cada individuo compite con el otro para ser seleccionado como padre. El motivo del uso de este mecanismo de selección es evitar seleccionar soluciones pobres como padres, ya que, para que se seleccionasen como padres dos soluciones pobres debe darse el caso de que las 4 soluciones seleccionadas sean malas soluciones.
(Para el correcto funcionamiento de este algoritmo es necesario mantener ordenada la población, así como mantener en memoria un vector que contenga todas las posibles parejas para seleccionar los competidores del torneo; los mecanismos empleados para ello serán detallados más adelante.)

```
begin
    .....
    for i ← 0 to NumeroCruces do
        ParejaAleatoria ← VectorParejas(Rand)
        Candidato1 ← ParejaAleatoria.first
        Candidato2 ← ParejaAleatoria.second
        Padre1 ← Candidato1 < Candidato2?Candidato1 : Candidato2
        ParejaAleatoria ← VectorParejas(Rand)
        Candidato1 ← ParejaAleatoria.first
        Candidato2 ← ParejaAleatoria.second
        Padre2 ← Candidato1 < Candidato2?Candidato1 : Candidato2
        Hijo ← OperadorCruce(Padre1, Padre2)
        NuevosIndividuos ← NuevosIndividuos ∪ Hijo
    end
    .....
end
```

2.7. Operadores de cruce

2.7.1. Operador de cruce basado en posiciones comunes

El funcionamiento del operador de cruce basado en posiciones comunes es simple; consiste en asignar a la permutación del hijo los elementos que coincidan en ambos padres y asignar de manera aleatoria el resto en aquellas posiciones de la permutación del hijo no ocupadas ya por elementos comunes.

```
function OperadorDeCrucePorPosicion(Padre1, Padre2) begin
    Hijo ← ∅   NoComunes ← ∅
    for i ← 0 to Dimension do
        if Coinciden Padre1 y Padre2 then
            Hijo[i] ← Padre1[i]
        else
            Hijo[i] ← ValorEspecial
            NoComunes.Add(Padre1[i])
        end
    end
    BarajarVector(NoComunes)
    for i ← 0 to Dimension do
        if Hijo[i] contiene Valor especial then
            Hijo[i] ← NoComunes[Elemento]
        end
    end
    CalcularSolucionNumerica(Hijo)
    return Hijo
end
```

2.7.2. Operador de cruce OX

Consiste en seleccionar un segmento de alguno de los padres que será copiado en el hijo sin modificación, todos aquellos elementos de la permutación del padre seleccionado como portador del segmento que hayan quedado fuera del mismo serán copiados en el hijo, en las posiciones vacías, atendiendo al orden en el que aparecen en el segundo padre.

```
function OperadorDeCruceOX(Padre1, Padre2) begin
    Segmento ← Dimension × 0,45
    Inicio ← Dimension/2 − Segmento/2
    Fin ← Inicio + Segmento  Hijo ← ∅   NoSegmento ← ∅
    for i ← 0 to Dimension do
        if i ∈ [Inicio, Fin) then
            Hijo[i] ← Padre1[i]
        else
            NoSegmento.Add(Padre1[i])
        end
    end
    #Ordenamos los elementos del vector atendiendo al orden en el que
    aparecen en Padre2
    OrdenarSegunPadre2(NoSegmento)
    for i ← 0 to Dimension do
        if i ∉ [Inicio, Fin) then
            Hijo[i] ← NoSegmento[Elemento]
            Elemento + +
    end
    CalcularSolucionNumerica(Hijo)
    return Hijo
end
```

3. Algoritmo de Búsqueda Local

En esta ocasión, emplearemos, para la implementación de la búsqueda local, un vector que nos permitirá "guiar" el proceso de búsqueda de forma que evitamos explorar ramas que sabemos que no conducen a una buena solución, el vector Don't look bits (DLB). El esquema general consiste en generar vecinos de una solución de manera aleatoria, para no dar prioridad a unos sobre otros, de forma que la solución actual será reemplazada por el primer vecino que proporcione una mejora a la solución. A continuación se muestra el pseudocódigo del método descrito. (Suponer que las operaciones se realizan sobre una interna a la clase).

```
function BusquedaLocal(MaxLlamadas) begin
    Intercamb ← Solucion ← PermutacionAleatoria
    while Se ha producido mejora && Llamadas < MaxLlamada do
        BarajarVector(Intercamb)
        DLB ← {0}
        for (i ← 0; i < Dimension && NoMejora; i++) do
            if DLB[i] == 0 then
                for (i ← 0; i < Dimension && NoMejora; i++) do
                    Incremento ←
                    ChequearMovimiento(Intercamb[i], Intercamb[j])
                    Llamadas ← Llamadas + 1/(Dimension/4)
                    if Incremento < 0 then
                        Individuo ← Vecino
                        Llamadas ← Llamadas + 1/(Dimension/4)
                        DLB[Intercamb[i]] ← DLB[Intercamb[j]] ← 0
                    else
                        DLB[Intercamb[i]] ← 1
                    end
                end
        end
    end
end
```

4. Algoritmos Genéticos Generacionales

Los algoritmos genéticos generacionales son métodos de exploración de espacios de soluciones basados en poblaciones en los que, en cada iteración, se genera una población de tamaño igual a la anterior de forma que la nueva población reemplaza por completo a la siguiente, a excepción del mejor elemento de la población anterior, en caso de que este sea mejor que el nuevo mejor individuo generado.

Para el correcto funcionamiento de este algoritmo es necesario mantener ordenada la población, pero, dado que sería muy costoso computacionalmente ordenar un vector de vectores que pueden ser de gran tamaño, mantendremos ordenada una estructura auxiliar en la que almacenamos parejas formadas por: el índice asociado a cada individuo en el vector de población y el valor de la solución asociada al mismo. De esta forma no es necesario ordenar la población, bastará con ordenar la estructura descrita atendiendo al valor de cada solución. Una de las ventajas de este método es que el torneo binario es inmediato, puesto que bastará con seleccionar al azar dos individuos del vector ordenado y comparar sus índices, aquel cuyo índice sea menor será seleccionado.

Otro aspecto importante a tener en cuenta es que, el proceso de generación de números aleatorios consume tiempo de ejecución, por tanto, reducir el número de llamadas al generador de números aleatorios proporcionará una mejora en el tiempo de ejecución. Por este motivo mantendremos en memoria un vector de parejas que contendrá todas las posibles parejas (no repetidas) de números enteros comprendidos en el intervalo $[0, Dimension)$ donde la dimensión es el tamaño de la permutación. De esta forma basta con generar un único número aleatorio para llevar a cabo los torneos binarios, en lugar de los dos que serían necesarios en caso de no utilizar este mecanismo.

Dado que el número de individuos generados en cada iteración viene dado por una probabilidad de cruce, y es calculado como $NumNuevosIndividuos = TamanioPoblacion \times ProbabilidadCruce$, es posible que no se generen tantos individuos como tamaño tenga la población completa. Puesto que el tamaño de la población debe ser constante a lo largo del proceso de evolución, puede ser necesario conservar miembros de la $Iteracion_{t-1}$ en la iteración $Iteracion_t$, en este caso, se conservarán los mejores individuos.

La única diferencia entre el algoritmo genético generacional con operador de cruce por posición y con operador de cruce OX es la llamada la función que genera el hijo dados dos padres.

```
function AGG − Posicion/OX(Pc, Pm, MaxEvals, TamanioPoblacion) begin
    # Cáculo del número de cruces a realizar
    NumCruces ← TamanioPoblacion × Pc)
    # Cáculo del número de mutaciones a realizar
    NumMutaciones ← Pm × TamanioPoblacion × Dimension
    #Inicializamos una población aleatoria de tamaño TamanioPoblacion
    InicializarPoblacionAleatoria(TamanioPoblacion)
    #Inicializar y ordenar el vector que indica el orden de la
    población
    Inicializar(OrdenPoblacion)
    Ordenar(OrdenPoblacion)
    Evals ← 0
    while Evals < MaxEvals do
        NuevosIndividuos ← ∅
        for i ← 0 to NumCruces do
            ParejaAleatoria ← VectorParejas(Rand)
            Candidato1 ← ParejaAleatoria.first
            Candidato2 ← ParejaAleatoria.second
            Padre1 ← Candidato1 < Candidato2?Candidato1 : Candidato2
            ParejaAleatoria ← VectorParejas(Rand)
            Candidato1 ← ParejaAleatoria.first
            Candidato2 ← ParejaAleatoria.second
            Padre2 ← Candidato1 < Candidato2?Candidato1 : Candidato2
            Hijo ← OperadorCrucePosicion/OX(Padre1, Padre2)
            NuevosIndividuos ← NuevosIndividuos ∪ Hijo
        end
        AplicarMutacion(NuevosIndividuos, NumMutaciones)
        Evals ← Evals + NumCruces + NumMutaciones/(Dimension/4)
        #Buscamos el mejor y el peor individuo de entre los generados
        MejorIndividuoNuevo ← BuscarMejor(NuevosIndividuos)
        PeorIndividuoNuevo ← BuscarPeor(NuevosIndividuos)
        if MejorIndividuoPoblacion < MejorIndividuoNuevo then
            NuevosIndividuos.Eliminar(PeorIndividuoNuevo)
            NuevosIndividuos.Aniadir(MejorIndividuoPoblacion)
        end
        #Si es necesario añadir individuos, añadimos los mejores de la
        poblacion anterior
        if NuevosIndividuos.Size() < TamanioPoblacion then
            Restantes ← TamanioPoblacion − NuevosIndividuos.Size()
            for i ← 0 to Restantes do
                NuevosIndividuos.Aniadir(Poblacion[OrdenPoblacion[i]])
            end
        end
        Poblacion ← NuevosIndividuos
        Inicializar(OrdenPoblacion)
        Ordenar(OrdenPoblacion)
    end
end
```

5. Algoritmos Genéticos Estacionarios

Los algoritmos genéticos estacionarios, de igual forma que los generacionales, son métodos de exploración de espacios de soluciones basados en poblaciones en los que, a diferencia de como se procede en los generacionales, en cada iteración, se generan dos individuos que compiten con los dos últimos individuos de la población para pasar a formar parte de la misma. En caso de que sean mejores, los peores individuos de la población se verán reemplazados por los nuevos individuos.

De igual forma que en los algoritmos generacionales, haremos uso de un vector auxiliar para mantener ordenada la población sin un sobrecoste computacional, así como emplearemos el mismo vector de parejas para seleccionar los competidores para los torneos binarios.

Para determinar si los dos individuos generados reemplazan a los dos últimos de la población, realizamos un torneo que implica a los individuos generados y a los dos últimos individuos de la población. Los dos ganadores de este torneo serán los que reemplacen a los dos últimos individuos de la población. Para realizar el torneo basta con ordenar un vector que contenga a los 4 participantes.

La diferencia más notable se encuentra en el método de mutación, en este caso no es posible aplicar a cada nueva población de individuos un número fijo de mutaciones calculadas en base a un porcentaje, debido al tamaño de la misma, a saber, 2. En su lugar calcularemos la frecuencia esperada con la que un gen muta, a saber, $FrecuenciaMutacion = ((1/ProbabilidadMutacion)/TamPoblacion)$, y contaremos el número de nuevos individuos generados hasta alcanzar dicha frecuencia, en cuyo caso mutaremos uno de los individuos de la población.

```
function AGE − Posicion/OX(Pm, MaxEvals, TamPoblacion) begin
    NumCruces ← 2)  NumMutaciones ← 2  ContMut ← 0
    FrecuenciaMutacion ← ((1/ProbabilidadMutacion)/TamPoblacion)
    #Inicializamos una población aleatoria de tamaño TamanioPoblacion
    InicializarPoblacionAleatoria(TamPoblacion)
    #Inicializar y ordenar el vector que indica el orden de la
    población
    Inicializar(OrdenPoblacion)
    Ordenar(OrdenPoblacion)
    Evals ← 0
    while Evals < MaxEvals do
        NuevosIndividuos ← ∅
        for i ← 0 to NumCruces do
            ParejaAleatoria ← VectorParejas(Rand)
            Candidato1 ← ParejaAleatoria.first
            Candidato2 ← ParejaAleatoria.second
            Padre1 ← Candidato1 < Candidato2?Candidato1 : Candidato2
            ParejaAleatoria ← VectorParejas(Rand)
            Candidato1 ← ParejaAleatoria.first
            Candidato2 ← ParejaAleatoria.second
            Padre2 ← Candidato1 < Candidato2?Candidato1 : Candidato2
            Hijo ← OperadorCrucePosicion/OX(Padre1, Padre2)
            NuevosIndividuos ← NuevosIndividuos ∪ Hijo
        end
        ContMut ← ContMut + NumCruces if ContMut ≥ FrecuenciaMutacion
        then
            AplicarMutacion(NuevosIndividuos, NumMutaciones)
            Evals ← Evals + NumCruces + NumMutaciones/(Dimension/4)
        end
        else
            Evals ← Evals + NumCruces
        end
        #Añadimos los dos últimos individuos de la población a los
        generados para realizar el torneo
        NuevosIndividuos.Add(Poblacion[OrdenPoblacion[TamPoblacion − 1]])
        NuevosIndividuos.Add(Poblacion[OrdenPoblacion[TamPoblacion − 2]])
        Ordenar(NuevosIndividuos)
        #Sustituimos los dos últimos elementos de la población por los
        ganadores
        Poblacion[OrdenPoblacion[TamPoblacion − 2]] ← NuevosIndividuos[0]
        Poblacion[OrdenPoblacion[TamPoblacion − 1]] ← NuevosIndividuos[1]
        Inicializar(OrdenPoblacion)
        Ordenar(OrdenPoblacion)
    end
end
```

6. Algoritmos Meméticos

Los algoritmos meméticos son algoritmos basados en evolución de poblaciones que utilizan todo el conocimiento disponible sobre el problema para realizar búsqueda heurística. En este caso basaremos los algoritmos meméticos en los algoritmos genéticos generacionales con cruce basado en posición, con la diferencia de que lanzaremos una búsqueda local sobre toda la población o un subconjunto de la misma con cierta frecuencia, antes de generar los nuevos individuos.

Debemos tener en consideración que la búsqueda local realiza llamadas a la función objetivo y a su factorización, por tanto estas llamadas deben computar para el calculo del contador que controla la condición de parada. Además, al aplicar una búsqueda local sobre los individuos de la población es posible que cambie el orden de la misma, por lo que deberemos reordenarla tras aplicar la búsqueda local a la población.

6.1. Algoritmo Memético de mejora completa (AM - (10, 1.0))

En este caso, cada 10 iteraciones, lanzaremos una búsqueda local sobre todos los miembros de la población. Para ello basta con recorrer en un bucle la población y llamar al algoritmo $LocalSearch(\&Solucion)$ sobre cada uno de ellos.

El siguiente pseudocódigo pone de manifiesto las diferencias entre el algoritmo memético y el algoritmo genético generacional:

```
function AMFull(Pc, Pm, MaxEvals, TamanioPoblacion) begin
    Declaración de variables...
    FrecuenciaLS ← 10  EvalsLS ← 0  ContadorGeneraciones ← 0
    Inicializar y ordenar OrdenPoblacion...
    while Evals < MaxEvals do
        NuevosIndividuos ← ∅
        if ContadorGeneraciones ≥ FrecuenciaLS then
            ContadorGeneraciones ← 0
            for i ← 0 to TamanioPoblacion do
                EvalsLS ← EvalsLS + LocalSearch(Poblacion[i])
            end
            Reiniciar y ordenar OrdenPoblacion...
        end
        Generación de nuevos individuos...
        Aplicar mutación...
        Evals ←
        Evals + EvalsLS + NumCruces + NumMutaciones/(Dimension/4)
        Conservar Elitismo...
        Mantener constante el tamaño de la población...
        Poblacion ← NuevosIndividuos
        Reiniciar y ordenar OrdenPoblacion...
        ContadorGeneraciones ← ContadorGeneraciones + 1
    end
end
```

6.2. Algoritmo Memético de mejora parcial aleatoria (AM - (10, 0.1))

En este caso, cada 10 iteraciones, lanzaremos una búsqueda local sobre un subconjunto aleatorio de los miembros de la población. Para no aplicar dos veces la búsqueda local sobre el mismo individuo, mantendremos en memoria un vector que contendrá todos los índices asociados a la población una única vez. Para seleccionar miembros aleatorios de la población barajaremos este vector cada vez que sea necesario.

El siguiente pseudocódigo pone de manifiesto las diferencias entre el algoritmo memético y el algoritmo genético generacional:

```
function AMSub(Pc, Pm, MaxEvals, TamanioPoblacion) begin
    Declaración de variables...
    FrecuenciaLS ← 10  EvalsLS ← 0  ContadorGeneraciones ← 0
    #Declaracion del vector de indices
    VA ← {0...TamanioPoblacion − 1}
    Barajar(VectorAleatorio)
    IndiceVA ← 0
    Inicializar y ordenar OrdenPoblacion...
    while Evals < MaxEvals do
        NuevosIndividuos ← ∅
        if ContadorGeneraciones ≥ FrecuenciaLS then
            ContadorGeneraciones ← 0
            for i ← 0 to TamanioPoblacion × 0,1 do
                EvalsLS ← EvalsLS + LocalSearch(Poblacion[VA[IndiceVA]])
                IndiceVA ← IndiceVA + 1
            end
            if IndiceVA ≥ VA.Size() − 1 then
                Barajar(VectorAleatorio)
                IndiceVA ← 0
            end
            Reiniciar y ordenar OrdenPoblacion...
        end
        Generación de nuevos individuos...
        Aplicar mutación...
        Evals ←
        Evals + EvalsLS + NumCruces + NumMutaciones/(Dimension/4)
        Conservar Elitismo...
        Mantener constante el tamaño de la población...
        Poblacion ← NuevosIndividuos
        Reiniciar y ordenar OrdenPoblacion...
        ContadorGeneraciones ← ContadorGeneraciones + 1
    end
end
```

6.3. Algoritmo Memético de mejora elitista (AM - (10, 0.1 mej))

En este caso, cada 10 iteraciones, lanzaremos una búsqueda local sobre los n mejores miembros, donde $n = TamanioPoblacion \times 0{,}1$.
El siguiente pseudocódigo pone de manifiesto las diferencias entre el algoritmo memético y el algoritmo genético generacional:

```
function AMElite(Pc, Pm, MaxEvals, TamanioPoblacion) begin
    Declaración de variables...
    FrecuenciaLS ← 10  EvalsLS ← 0  ContadorGeneraciones ← 0
    Inicializar y ordenar OrdenPoblacion...
    while Evals < MaxEvals do
        NuevosIndividuos ← ∅
        if ContadorGeneraciones ≥ FrecuenciaLS then
            ContadorGeneraciones ← 0
            for i ← 0 to TamanioPoblacion × 0,1 do
                EvalsLS ← EvalsLS + LocalSearch(Poblacion[OrdenPoblacion[i]])
            end
            Reiniciar y ordenar OrdenPoblacion...
        end
        Generación de nuevos individuos...
        Aplicar mutación...
        Evals ←
        Evals + EvalsLS + NumCruces + NumMutaciones/(Dimension/4)
        Conservar Elitismo...
        Mantener constante el tamaño de la población...
        Poblacion ← NuevosIndividuos
        Reiniciar y ordenar OrdenPoblacion...
        ContadorGeneraciones ← ContadorGeneraciones + 1
    end
end
```

7. Algoritmo de comparación: Algoritmo Greedy

Los algoritmos voraces proporcionan soluciones a problemas en tiempo mínimo a costa de una pérdida de calidad en la solución. En este caso el algoritmo voraz consiste en seleccionar iterativamente las unidades funcionales que maximicen el flujo y situarlas en las localizaciones mas céntricas, es decir, en aquellas localizaciones que minimicen la distancia. Para ello basta con almacenar en dos vectores la suma por filas de las matrices de flujo y distancia respectivamente, y seleccionar de entre ellos de la forma descrita.

```
function Greedy() begin
    DistSumV ← ∅ FluxSumV ← ∅
    #Calcular los vectores que almacenan la suma
    for i ← 0 to Dimension do
        FSum ← 0  DSum ← 0
        for j ← 0 to Dimension do
            FSum ← FSum + MatrizFlujo[i][j]
            DSum ← DSum + MatrizDistancia[i][j]
        end
        FluxSumV[i] ← FSum
        DistSumV[i] ← DSum
    end
    #Asignar unidades a localizaciones
    for i ← 0 to Dimension do
        MaxFluxVal ← −1
        MinDistVal ← ∞
        for j ← 0 to Dimension do
            #Actualizar mejor flujo y unidad
            if Encontradoflujomayor then
                MaxFluxVal ← FluxSumV[i]
                MaxFluxValInd ← i
            end
            #Actualizar mejor distancia y localización
            if Encontradadistancimenor then
                MinDistVal ← DistSumV[i]
                MinDistValInd ← i
            end
        end
        #Asignar localización a unidad
        Solucion[MaxFluxValInd] ← MinDistValInd
        #Marcar unidad y localización como procesados
        FluxSumV[MaxFluxValInd] ← −1
        DistSumV[MinDistValInd] ← ∞
    end
end
```

8. Procedimiento considerado para el desarrollo de la práctica

El código empleado para el desarrollo de esta obra ha sido enteramente escrito por el autor de la misma sin más ayuda que la proporcionada por el guión y los seminarios impartidos en clase.

El estilo de programación es orientado a objetos; cabe destacar el uso de bibliotecas implementadas para `C++` como la `STL`, la biblioteca `Algorithm` o la biblioteca `Chrono`, que proporciona un reloj de alta resolución.

Para la obtención de las gráficas de convergencia se ha hecho uso de la herramienta `Gnuplot`, que permite, de manera sencilla, obtener gráficas a partir de ficheros de datos.

9. Experimentos y Análisis de resultados

9.1. Descripción de parámetros y casos considerados

Para la obtención de resultados se han considerado todas las instancias proporcionadas para el desarrollo de la práctica, así como los parámetros recomendados. Para la búsqueda local se ha fijado el número máximo de evaluaciones a 50000, así como para los algoritmos genéticos y meméticos; para estos dos últimos se ha establecido la probabilidad de cruce a 0.6, la de mutación a 0.001 y el tamaño de la población a 50, en el caso de los algoritmos meméticos se establece el máximo de evaluaciones realizadas por la búsqueda local a 400.

Para tomar las mediciones se ha lanzado cada algoritmo para cada instancia 5 veces con 5 semillas diferentes, a saber: 5, 17, 281, 881 y 6673.

Cabe destacar que todo el código ha sido compilado con optimización automática de compilador 2.

9.2. Análisis de resultados del algoritmo Greedy

Los resultados obtenidos con el algoritmo Greedy para cada instancia son los siguientes:

Algoritmo Greedy					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	365.796	8.1752e-06	sko100a	13.2327	5.26428e-05
chr22a	119.916	8.6274e-06	sko100b	13.4902	6.00204e-05
els19	124.416	6.999e-06	sko100c	14.5271	5.23372e-05
esc32b	90.4762	9.674e-06	sko100d	12.5287	5.39372e-05
kra30b	29.6106	6.907e-06	sko100e	13.2511	5.32086e-05
lipa90b	29.0592	4.52854e-05	tai30	117.729	6.6426e-06
nug25	18.5363	5.0162e-06	tai50	71.8325	1.636e-05
sko56	19.2931	1.885e-05	tai60	15.8156	2.1938e-05
sko64	17.6255	2.3584e-05	tai256	120.481	0.000295434
sko72	15.6424	2.96366e-05	tho150	17.14	0.00011395

Cuadro 1: Tabla que contiene los datos asociados al algoritmo Greedy

Teniendo en cuenta el reducido orden de complejidad del algoritmo Greedy, así como la simplicidad de las operaciones que realiza, es de esperar que, tal y como sucede, el algoritmo Greedy proporcione resultados excelentes en cuanto al tiempo se refiere; esta mejora en tiempo es a costa de una pérdida notable de calidad en las soluciones. En ningún caso la desviación típica proporcionada por este algoritmo es menor que 10, por tanto, aunque el algoritmo Greedy proporciona un buen marco de comparación, no es el adecuado para resolver este problema si lo que queremos es obtener una solución lo más cercana a la óptima posible.

9.3. Análisis de resultados de la Búsqueda Local

Los resultados obtenidos con la Búsqueda Local para cada instancia son los siguientes:

Algoritmo de Búsqueda Local					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	49.121	0.000124205	sko100a	2.10892	0.0354537
chr22a	13.9766	0.000199907	sko100b	1.98271	0.0334663
els19	35.0829	0.000204955	sko100c	1.6345	0.0413691
esc32b	30	0.000607973	sko100d	1.78852	0.0342172
kra30b	5.88711	0.000605045	sko100e	1.81026	0.0305345
lipa90b	21.5846	0.0264895	tai30	18.0639	0.000691034
nug25	5.26709	0.00029219	tai50	7.09251	0.00378619
sko56	2.43891	0.00551328	tai60	4.3088	0.006036
sko64	1.99183	0.00905081	tai256	0.385985	0.490289
sko72	2.58573	0.0119419	tho150	1.92728	0.19621

Cuadro 2: Tabla que contiene los datos asociados a la Búsqueda Local

La búsqueda local es un método de resolución de problemas eficiente en cuanto a tiempo se refiere, pero siempre corre el riesgo de caer en óptimos locales y nunca alcanzar la solución óptima, es más, podría darse el caso de que el algoritmo cayera en un óptimo local muy alejado de la solución óptima, tal y como parece suceder en los casos de menor dimensión. Por otra parte vemos que la búsqueda local es capaz de proporcionar desviaciones respecto al óptimo inferiores incluso al 1 %. De esta forma podemos decir que la búsqueda local parece adecuada si tenemos información sobre los datos que nos permita concluir que la probabilidad de caer en un óptimo local es pequeña.

9.4. Análisis de resultados del Algoritmo Genético Generacional con cruce por posición

La tabla que recoge los datos obtenidos para el Algoritmo Genético Generacional con Cruce por Posición es la siguiente:

Algoritmo Genético Generacional con Cruce por Posición					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	49.5561	0.0376079	sko100a	2.82365	0.620938
chr22a	13.7557	0.0456947	sko100b	3.10559	0.621921
els19	15.8206	0.0346957	sko100c	3.37179	0.620217
esc32b	25.2381	0.0827874	sko100d	3.16388	0.635166
kra30b	6.68781	0.0695518	sko100e	3.42608	0.628701
lipa90b	22.7584	0.525889	tai30	8.28752	0.0671699
nug25	3.78205	0.0502308	tai50	5.71605	0.169913
sko56	3.61716	0.212546	tai60	5.29765	0.244666
sko64	3.27849	0.277926	tai256	1.29676	4.0576
sko72	3.29027	0.344342	tho150	4.03556	1.38257

Cuadro 3: Tabla que contiene los datos asociados al Algoritmo Genético Generacional con Cruce por Posición

Vemos que el algoritmo Genético Generacional con Cruce por Posición proporciona soluciones de calidad en tiempo reducido, sin embargo en algunas ocasiones las soluciones proporcionadas llegan a alejarse del óptimo en hasta casi un 50 %, esto puede deberse a una rápida convergencia de la población que haga que la evolución de la población sea un proceso más de explotación que de exploración, bastaría con aumentar la probabilidad para determinar si es este el caso.
Podemos representar en una gráfica el valor para la mejor y peor solución para obtener información sobre el comportamiento de la población a lo largo de la evolución de la misma; representaremos en la misma gráfica también el valor para la quinta peor solución, ya que, dado que las mutaciones introducen diversidad en la población, el valor para la peor solución podría oscilar lo suficiente como para situarse lejos de la mejor solución tanto en etapas tempranas como tardías.

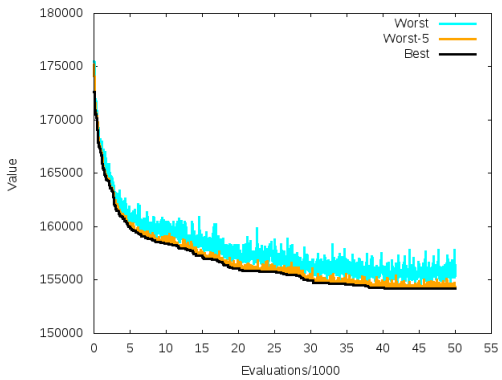

Figura 1: Gráfica que muestra la evolución de la población en el Algoritmo Genético Generacional con Cruce por Posición.

En la gráfica podemos observar que la población converge en etapas tempranas del algoritmo, de forma que es el operador de mutación el encargado de introducir diversidad en la población en etapas posteriores. Tal y como habíamos predicho, el valor de la peor solución oscila durante todo el desarrollo del algoritmo; la cuarta peor solución nos proporciona una mejor medida para determinar la convergencia de la población.

9.5. Análisis de resultados del Algoritmo Genético Generacional con cruce OX

En la siguiente tabla se encuentran recogidos los datos obtenidos para el Algoritmo Genético Generacional con Cruce OX:

Algoritmo Genético Generacional con Cruce OX					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	45.5527	0.0622186	sko100a	2.7326	0.896222
chr22a	15.2762	0.0682377	sko100b	2.6029	0.902475
els19	22.0427	0.0591677	sko100c	3.1175	0.913839
esc32b	28.0952	0.131873	sko100d	3.17752	0.910332
kra30b	6.90877	0.103364	sko100e	3.45156	0.899383
lipa90b	22.7863	0.751808	tai30	9.34051	0.103178
nug25	4.13462	0.079573	tai50	4.62364	0.270896
sko56	3.33275	0.290638	tai60	5.66483	0.355871
sko64	3.09621	0.392527	tai256	1.29891	5.45478
sko72	3.26793	0.528597	tho150	4.36575	1.85704

Cuadro 4: Tabla que contiene los datos asociados al AG Generacional con Cruce OX

De igual forma que en el caso anterior el Algoritmo Genético Generacional con Cruce OX proporciona, en la mayoría de las ocasiones, soluciones de calidad en un tiempo reducido, por tanto, podemos concluir que, salvo en algunas excepciones, se trata de un buen método de resolución para el problema de la asignación cuadrática, aunque, una vez más, encontramos casos en los que el método parece no comportarse como se espera de él, sobre todo en los casos de dimensión reducida.
Podemos también representar en una gráfica los mismos datos que representábamos en el caso anterior para comprobar que este algoritmo parece comportarse de manera muy similar al anterior, lo que es de esperar puesto que comparten la mayoría de operadores.

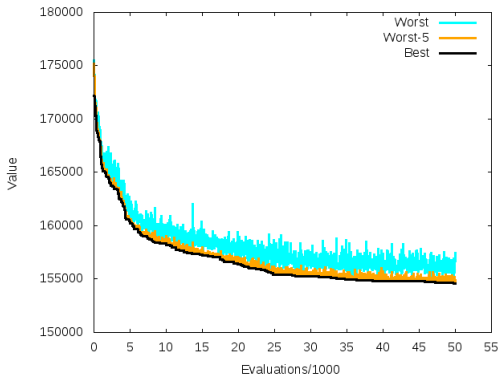

Figura 2: Gráfica que muestra la evolución de la población en el Algoritmo Genético Generacional con Cruce OX.

9.6. Análisis de resultados del Algoritmo Genético Estacionario con cruce por posición

La tabla que recoge los datos obtenidos para el Algoritmo Genético Generacional con Cruce por Posición es la siguiente:

Algoritmo Genético Estacionario con Cruce por Posición					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	41.9669	0.0532135	sko100a	3.32759	0.644159
chr22a	13.8012	0.0588647	sko100b	3.00214	0.644345
els19	17.0068	0.0516091	sko100c	3.62933	0.644071
esc32b	40.4762	0.102095	sko100d	3.38985	0.64236
kra30b	6.18902	0.0871975	sko100e	3.7361	0.6509
lipa90b	23.1209	0.554006	tai30	17.1822	0.083847
nug25	4.97863	0.0679414	tai50	7.50317	0.188142
sko56	3.96773	0.23033	tai60	6.04028	0.258917
sko64	4.27069	0.291898	tai256	1.27122	4.18861
sko72	3.86803	0.365735	tho150	4.73633	1.35778

Cuadro 5: Tabla que contiene los datos asociados al AG Estacionario con Cruce por Posición

A la vista de los resultados podemos concluir que, aunque el Algoritmo Genético Estacionario con Cruce por Posición proporciona soluciones de calidad en tiempo reducido, parece comportarse peor que la versión generacional del mismo, además de presentar un leve aumento en el tiempo de ejecución medio. Por otra parte podríamos decir que el algoritmo se comporta mejor que los anteriores en casos de dimensión reducida.

9.7. Análisis de resultados del Algoritmo Genético Estacionario con cruce OX

La tabla que recoge los datos obtenidos para el Algoritmo Genético Generacional con Cruce por Posición es la siguiente:

Algoritmo Genético Estacionario con Cruce OX					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	47.9373	0.0734059	sko100a	3.11206	0.919068
chr22a	13.3073	0.0814037	sko100b	3.18929	0.901415
els19	17.3905	0.0712997	sko100c	3.52383	0.900159
esc32b	37.1429	0.142177	sko100d	3.26603	0.896762
kra30b	5.28768	0.115615	sko100e	3.79511	0.906793
lipa90b	22.9082	0.760734	tai30	11.2028	0.116365
nug25	4.2735	0.0933909	tai50	7.84641	0.262443
sko56	4.53422	0.299718	tai60	5.76926	0.361858
sko64	4.37626	0.421914	tai256	1.0706	5.22044
sko72	3.3923	0.533318	tho150	4.46311	1.90374

Cuadro 6: Tabla que contiene los datos asociados al AG Estacionario con Cruce OX

Viendo los resultados podemos decir que, de igual forma que en el caso anterior, el Algoritmo Genético Estacionario con Cruce OX proporciona, salvo por algunas excepciones, soluciones de calidad para el problema que nos ocupa, aunque con un leve aumento en el tiempo respecto al caso anterior. Sin embargo, la desviación media sí parece mejorar respecto al caso anterior.

9.8. Análisis de los resultados de los Algoritmos Meméticos

9.8.1. Análisis de resultados

En comparación con los algoritmos genéticos, los algoritmos meméticos presentan unos resultados excelentes en cuanto a desviación respecto al óptimo se refiere, aunque con una leve pérdida en el tiempo con respecto a los mismos.
Como cabe esperar, el algoritmo memético que mejores resultados presenta es el de mejora completa de la población. Este algoritmo, aún empleando la mayoría de las evaluaciones disponibles en la búsqueda local, consigue mejorar los resultados proporcionados por los otros dos algoritmos meméticos, que emplean mas evaluaciones en la evolución de la población. Este resultado se debe a la calidad de las soluciones proporcionadas por la búsqueda local. Cabe destacar que, aunque este algoritmo proporciona mejores resultados en desviación, presenta también un leve aumento en el tiempo de ejecución respecto a sus dos competidores.
Seguido del algoritmo memético de mejora completa encontramos al de mejora parcial aleatoria, que supera al de mejora elitista, esto puede deberse a que, al contrario que el elitista, la mejora parcial aleatoria introduce diversidad en la población, lo que le permite explorar un espacio de soluciones más amplio.
Podemos representar en una gráfica, de igual forma que hacíamos con los algoritmos genéticos generaciones, los valores de la mejor, peor, y cuarta peor solución para obtener información sobre el comportamiento de la población en cada uno de los algoritmos meméticos.

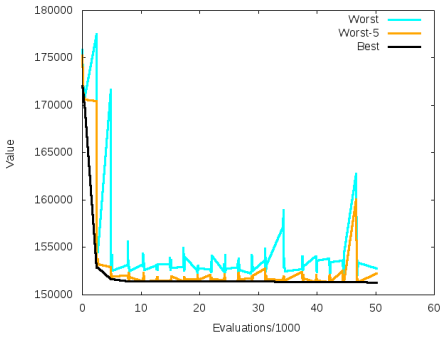

Figura 3: Gráfica que muestra la evolución de la población en el Algoritmo Memético de mejora parcial.

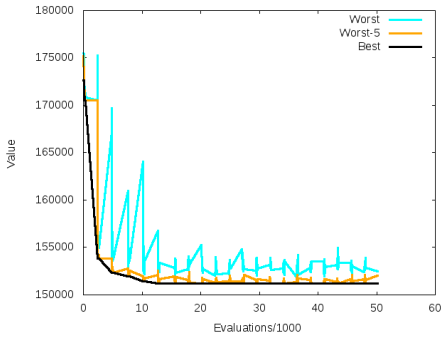

Figura 4: Gráfica que muestra la evolución de la población en el Algoritmo Memético de mejora elitista.

A la vista de las gráficas podemos decir que los algoritmos se comportan de forma similar. Las oscilaciones en las peores soluciones de deben a que, en las iteraciones en las que se mejora la población, en el caso de la mejora parcial cabe la posibilidad de que la peor solución sea mejorada y deje de ser la peor solución para dar paso a otra solución peor que la anterior; mientras que, en el caso de la mejora elitista vemos que la peor solución tiende a estabilizarse y acercarse a la mejor, esto se debe a que no es posible que la

peor solución deje de serlo debido a búsquedas locales, ya que estas solo se aplican a los mejores individuos de la población, de esta forma la peor solución solo puede mejorar al ser reemplazada por otra durante el proceso de reemplazo o mutación.

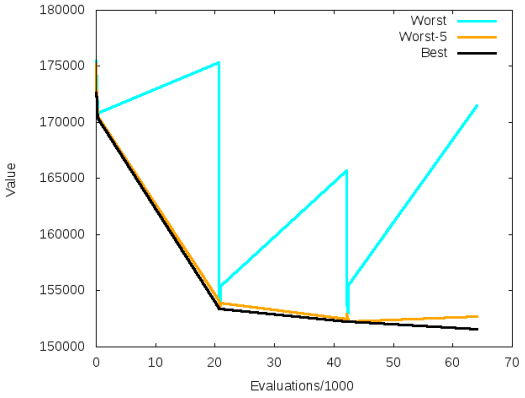

Figura 5: Gráfica que muestra la evolución de la población en el Algoritmo Memético de mejora completa.

Dado que en el caso del algoritmo memético de mejora completa se emplean muchas más iteraciones en la búsqueda local que en los otros dos, no es posible hacer una medición realmente significativa, ya que, al representar el valor frente al número de evaluaciones realizadas se da el caso de que la "distancia" entre mediciones es demasiado grande. Aún con esto podemos ver en la gráfica una clara oscilación de la peor solución debido a la pérdida de calidad de esta entre llamadas a la búsqueda local para la misma, mientras que la mejor solución mejora en etapas tempranas del algoritmo y se mantiene relativamente estable.

9.8.2. Tabla de resultados del Algoritmo Memético con mejora completa

La tabla que recoge los datos obtenidos para el Algoritmo Memético con mejora completa:

Algoritmo Memético con Mejora Completa					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	15.0566	0.0342958	sko100a	0.982092	1.01435
chr22a	4.73684	0.041258	sko100b	1.12782	1.02969
els19	5.41856	0.0313037	sko100c	1.06856	1.00132
esc32b	10	0.0842199	sko100d	1.26277	1.02495
kra30b	1.43951	0.0757131	sko100e	1.23419	0.999133
lipa90b	21.3346	0.89017	tai30	1.89744	0.0756545
nug25	0.224359	0.0529801	tai50	1.36411	0.236681
sko56	1.31058	0.2804	tai60	3.32016	0.330597
sko64	1.25448	0.382895	tai256	0.330001	11.4529
sko72	1.12594	0.510219	tho150	1.649	2.33096

Cuadro 7: Tabla que contiene los datos asociados al AM con mejora completa

9.8.3. Tabla de resultados del Algoritmo Memético con mejora parcial

La tabla que recoge los datos obtenidos para el Algoritmo Memético con mejora parcial:

Algoritmo Memético con Mejora Parcial					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	17.7894	0.0361432	sko100a	1.33604	0.781477
chr22a	6.78363	0.0410684	sko100b	1.14107	0.767343
els19	11.461	0.0369109	sko100c	1.43999	0.794057
esc32b	14.7619	0.0842439	sko100d	1.25742	0.791422
kra30b	2.08488	0.0722008	sko100e	1.40127	0.765639
lipa90b	21.2198	0.632343	tai30	4.91814	0.0713379
nug25	0.82265	0.0529358	tai50	3.27844	0.198651
sko56	1.46497	0.234458	tai60	3.5654	0.270645
sko64	1.32377	0.312339	tai256	0.367375	7.52126
sko72	1.27264	0.42065	tho150	1.96439	1.87782

Cuadro 8: Tabla que contiene los datos asociados al AM con mejora parcial

9.8.4. Tabla de resultados del Algoritmo Memético con mejora elitista

La tabla que recoge los datos obtenidos para el AM con mejora elitista:

Algoritmo Memético con mejora elitista					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	28.651	0.0356483	sko100a	1.37972	0.771712
chr22a	10.0975	0.0412956	sko100b	1.53304	0.768565
els19	9.97299	0.03414	sko100c	1.06992	0.782791
esc32b	12.381	0.0814054	sko100d	1.32535	0.782545
kra30b	3.58565	0.0786399	sko100e	1.86336	0.762292
lipa90b	21.6455	0.626443	tai30	3.3962	0.0762582
nug25	1.78419	0.0507658	tai50	3.07605	0.189336
sko56	2.07557	0.236508	tai60	3.33128	0.269062
sko64	1.9341	0.307968	tai256	0.371783	7.39254
sko72	1.47609	0.398982	tho150	1.79966	1.86555

Cuadro 9: Tabla que contiene los datos asociados al AM con mejora elitista

9.9. Análisis de resultados generales

La siguiente tabla recoge los resultados medios obtenidos para cada algoritmo:

Algoritmo	Desv	Tiempo
Greedy	62,01996	4,481808E-05
BL	10,45195775	0,0467831012
AGG - Posicion	9,4154605	0,53842018
AGG - OX	9,743455	0,74514024
AGE - Posicion	10,6732155	0,554948265
AGE - OX	10,389433	0,75297976
AM - (10, 1.0)	3,8068806	1,09386312
AM - (10, 0.1)	4,98270875	0,78891195
AM - (10, 0.1 mej)	5,63749765	0,78767248

Cuadro 10: Tabla que contiene los datos asociados al análisis de resultados medios generales

La tabla de resultados generales nos permite tener una visión mas general del comportamiento de los algoritmos. Podemos ver que los algoritmos que mejor se comportan respecto a la desviación son, con diferencia, los algoritmos meméticos; esto se debe a que utilizan más información para resolver el problema que los algoritmos genéticos. Por otra parte, estos últimos proporcionan soluciones de cierta calidad, concretamente los generacionales superan a la búsqueda local, aunque no en tiempo, si en desviación.

Respecto a los diferentes operadores de cruce podemos ver que, en el caso de los generaciones se comporta mejor el cruce por posición, mientras que en el caso de los estacionarios es el OX el que mejor actúa, podemos concluir entonces que el mejor comportamiento de uno u otro depende, no solo de la técnica con la que lo combinemos, también de la instancia concreta del problema a la que lo apliquemos

Por último, los métodos no basados en poblaciones. El algoritmo voraz proporciona soluciones en tiempo inmejorable, aunque de baja calidad, mientras que la búsqueda local proporciona soluciones que no tiene que envidiar a las proporcionadas por los genéticos en algunos casos y además en un tiempo menor.

10. Manual de usuario

Junto al código fuente utilizado para el desarrollo de la práctica se incluye un archivo tipo makefile que automatiza el proceso de compilación; este archivo genera un fichero ejecutable para cada uno de los algoritmos, así como un ejecutable general llamado `mainGeneral` que lanza todos los algoritmos con los parámetros especificados en el guión, es este último el utilizado para tomar las mediciones pertinentes. Además, se incorpora un archivo `run.sh` que ejecutará el archivo `mainGeneral` para cada una de las instancias del problema y almacenará el resultado en el directorio `results` bajo el nombre de `<nombre_instancia>.rslt`.
En la estructura de directorios también se encuentra el directorio `graphs`, que contiene los datos tomados para la obtención de las gráficas, así como estas mismas.