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

Práctica 2.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: Greedy, búsqueda local (BL), búsqueda multiarranque básica (BMB), enfriamiento simulado (ES), GRASP, búsqueda local reiterada (ILS), híbrido ILS-ES.

Índice

Índice de cuadros

1. Descripción del problema

El problema de la asignación cuadrática o QAP (Quadratic Assignment 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. En este caso emplearemos métodos basados en trayectorias como pueden ser el Enfriamiento Simulado (ES), Búsqueda Multiarranque Básica (BMB), Búsqueda Local Reiterada (ILS), GRASP y un híbrido que incluye ES e ILS. Además consideraremos algunos de los métodos de resolución ya desarrollados en la práctica anterior, 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 comparten 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. Factorizaciones 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 la 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.

Por otra parte, dada una solución parcial, es decir, una solución en la que no todos los elementos de la permutación han sido asignados, podemos calcular el incremento en el coste de la solución que supone asignar una nueva unidad a una nueva localización. Para ello es necesario conocer la unidad y la localización a añadir, así como los elementos de la

permutación que deben ser considerados. A continuación se muestra el pseudocódigo que describe esta idea:

Input: Dados u la unidad a asignar, l la localización a asignar, π la solución parcial y $asig$ la posiciones a tener en cuenta en π

function $ChequearAdicion(u,\ l,\ \pi,\ asig)$ **begin**
  $Incremento \leftarrow 0$
  for $i \leftarrow 0$ **to** $asig.tamanio()$ **do**
    $Incremento \leftarrow Incremento + F_{asig[i],u} \times D_{\pi(asig[i]),l}$
    $Incremento \leftarrow Incremento + F_{u,asig[i]} \times D_{l,\pi(asig[i])}$
  end
  return $Incremento$
end

2.3. Generación de soluciones aleatorias

Los algoritmos no constructivos (algoritmos de mejora) necesitan una solución inicial aleatoria sobre la que comenzar a trabajar, para generar las soluciones aleatorias implementamos el siguiente método:

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

3. 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 más 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
```

4. 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
```

5. Algoritmo de Enfriamiento simulado

Enfriamiento simulado es una técnica de exploración del espacio de soluciones basada en trayectorias. Introduce el concepto de plan de enfriamiento, que imita el proceso de enfriamiento descrito por los modelos físicos para considerar soluciones peores a la actual que le puedan llevar a un mejor óptimo que el que alcanza la búsqueda local básica.

Es necesario entonces describir un plan de enfriamiento para el funcionamiento de este algoritmo, en este caso emplearemos el esquema de enfriamiento de Cauchy modificado:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k} \quad \beta = \frac{T_0 - T_f}{M T_0 T_f} \quad T_0 = \frac{\mu C(S_0)}{-ln(\phi)}$$

Donde M es el número de enfriamientos a realizar, T_0 es la temperatura inicial y T_f es la temperatura final, que tendrá un valor cercano a 0. A continuación se describe en pseudocódigo la implementación de esta idea:

```
function EnfiramientoSimulado(...) begin
    #Inicalizamos la solución y el vector de intercambios
    Solucion ← PermutacionAleatoria
    Intercambiador ← PermutacionAleatoria
    #Inicalizamos los parámetros del enfriamiento
    T = (μ * C(Solucion))/(−log(φ))
    β = (T − T_fin)/(M * T * T_fin)

    while T > T_fin && successes > 0 && K <= M do
        incremento ← generados ← 0
        Barajar(Intercambiador)
        for i ← 0 to Dimension do
            for j ← i + 1 to Dimension do
                #Generamos un vecino
                a ← intercambiador[i]; b ← intercambiador[j]
                incremento ← ChaquearMovimiento(a, b)
                generados ← generados + 1
                #Aceptamos el vecino si es mejor o si lo indica el
                proceso de enfriamiento
                if incremento < 0 || Rand() ≤ e^(−incremento/T) then
                    AplicarMovimiento(a , b, incremento)
                    exitos ← exitos + 1
                end
            end
        end
        #Actualizamos la temperatura
        T ← T/(1+βT)
    end
end
```

Debemos tener en cuenta que los dos bucles internos al `while` también se detendrán si se alcanza el número máximo de vecinos permitidos, contabilizados por la variable *generados* o el número máximo de actualizaciones de solución, contabilizados por la variable *exitos*.

6. Algoritmo de Búsqueda Multiarranque Básica

El algoritmo de búsqueda multiarranque básica consiste en lanzar el algoritmo de búsqueda local sobre un número N de soluciones generadas de forma aleatoria, de esta forma es posible encontrar mejores soluciones que con una única búsqueda local. A continuación se muestra el pseudocódigo que implementa esta idea (la función C es la función de coste):

```
function BMB(MaxIters, MaxLlamadasLS) begin
    #Inicializamos la mejor solución de forma aleatoria
    MejorSolucion ← GenerarSolucionAleatoria()

    for i ← 0 to MaxIters do
        #Generamos una nueva solución de forma aleatoria
        SlcnAleatoria ← GenerarSolucionAleatoria()
        #Lanzamos BL sobre la nueva solución aleatoria
        NuevaSlcn ← BusquedaLocal(SlcnAleatoria, MaxLlamadasLS)
        #Actualizamos la mejor solución si fuese necesario
        if C(NuevaSlcn) < C(MejorSolucion) then
            MejorSolucion ← NuevaSlcn
        end
    end
    return MejorSolucion
end
```

El proceso de generación de soluciones aleatorias ha sido descrito en anteriores secciones de este documento.

7. Algoritmo GRASP

El procedimiento GRASP consiste en lanzar el algoritmo de Búsqueda Local sobre soluciones generadas mediante un procedimiento greedy aleatorizado que introduce diversidad en la exploración del espacio de soluciones. Este procedimiento de generación de soluciones greedy aleatorizadas consta de dos etapas, en la primera se asignan dos unidades a dos localizaciones conjuntamente, estas serán seleccionadas aleatoriamente de entre la lista de mejores candidatos correspondientes a cada una, en la etapa dos se asigna una unidad a una localización seleccionadas aleatoriamente de entre la lista de aquellas asignaciones factibles que provocan el menor incremento en el coste de la solución.

A continuación se muestra el pseudocódigo correspondiente a la obtención de las listas de candidatos de la primera etapa, y el pseudocódigo correspondiente al procedimiento de generación de soluciones greedy aleatorizadas:

```
function ListasCostes(&Unidades, &Locs, α) begin
    #Inicializamos los límites
    UnidadMin ← LocMin ← ∞
    UnidadMax ← LocMax ← −∞
    #Buscamos los máximos y mínimos en las unidades y localizaciones
    for i ← 0 to Dimension do
        if Unidades[i] > UnidadMax then
            UnidadMax ← Unidades[i]
        end
        if Unidades[i] < UnidadMin then
            UnidadMin ← Unidades[i]
        end
        if Locs[i] > LocMax then
            LocMax ← Locs[i]
        end
        if Locs[i] < LocMin then
            LocMin ← Locs[i]
        end
    end
    #Calculamos los umbrales μ para las listas de candidatos
    μ_u = UnidadMax − α(UnidadMax − UnidadMin)
    μ_l = LocMin − α(LocMax − LocMin)
    #Inicilizamos las listas de mejores candidatos
    MejoresUnds ← MejoresLocs ← ∅
    #Añadimos unidades y localizaciones a las listas de mejores
    candidatos según los umbrales μ_u y μ_l
    for i ← 0 to Dimension do
        if Unidades[i] >= μ_u then
            MejoresUnds.Add(pareja(i, Unidades[i]))
        end
        if Localzcns[i] <= μ_l then
            MejoresLocs.Add(pareja(i, Locs[i]))
        end
    end
    return (MejoresUnds, MejoresLocs)
end
```

```
function CalcularGreedyAleatorizado(α) begin
    DistSumV ← FluxSumV ← ∅
    #Calculamos los vectores que almacenan la suma de igual forma que
    en el Greedy
    FSum ← SumaFlujos; DSum ← SumaDistancias
    #Inicalizamos el vector de posibles asignaciones (bucle doble)
    for i, j ← 0 to Dimension do
        AsigsFactibles.Add(pareja(i, j))
    end
    (MejoresUnds, MejoresLocs) ← ListasCostes(FSum, DSum, α)
    #Asignamos las dos primeras localizaciones a unidades de forma
    aleatoria (R1 ≠ R3 y R2 ≠ R4)
    Solucion[MejoresUnds[R1].first] ← MejoresLocs[R2].first
    Solucion[MejoresUnds[R3].first] ← MejoresLocs[R4].first
    Asigs.Add(MejoresUnds[R1].first); Asigs.Add(MejoresUnds[R3].first)
    #Tachamos de entre las asignaciones posibles aquellas relacionadas
    con las asignaciones realizadas
    for i ← 0 to AsigsFactibles.tamanio() do
        if Ya asignada AsigsFactibles[i] then
            AsigsFactibles[i] ← −1
        end
        ValorAsig.Add(∞)#Inicializamos el vector de costes de
        asignaciones
    end
    for k ← 0 to Dimension − 2 do
        for i ← 0 to AsigsFactibles.tamanio() do
            if No tachada AsigsFactibles[i] then
                ValorAsig[i] ←
                ChequearAdicion(AsigsFactibles[i] Solucion, Asigs)
                if ValorAsig[i] < min then
                    min ← ValorAsig[i]
                end
                if ValorAsig[i] > max then
                    max ← ValorAsig[i]
                end
            end
        end
        μ ← min + α ∗ (max − min)#Calculamos el umbral
        #Obtenemos la lista de candidatos
        Candidatos ← ObtenerCandidatos(ValorAsig, μ)
        (Unidad, Loc) ← AsigsFactibles[Candidatos[R1]]#R1 entero aleatorio
        #Asignamos la unidad y localización seleccionadas
        Solucion[Unidad] ← Loc; Asigs.Add(Unidad)
        #Tachamos las asignaciones que dejan de ser factibles
        for i ← 0 to AsigsFactibles.tamanio() do
            if Ya asignada AsigsFactibles[i] then
                AsigsFactibles[i] ← −1
            end
        end
    end
    CalcularCoste(Solucion); return Solucion
end
```

Finalmente se muestra el pseudocódigo correspondiente al procedimiento que emplea los métodos anteriormente descritos para obtener una solución (la función C es la función de coste):

```
function GRASP(MaxIters, MaxLlamadasLS, α) begin
    #Inicializamos la mejor solución de forma aleatoria
    MejorSolucion ← GenerarSolucionAleatoria()
    for i ← 0 to MaxIters do
        #Generamos una nueva solución greedy aleatorizada
        SlcnGreedy ← CalcularGreedyAleatorizado(α)
        #Lanzamos BL sobre la nueva solución obtenida
        NuevaSlcn ← BusquedaLocal(SlcnGreedy, MaxLlamadasLS)
        #Actualizamos la mejor solución si fuese necesario
        if C(NuevaSlcn) < C(MejorSolucion) then
            MejorSolucion ← NuevaSlcn
        end
    end
    return MejorSolucion
end
```

8. Algoritmo de Búsqueda Local Reiterada

El algoritmo de Búsqueda Local Reiterada consiste en lanzar búsqueda local sobre soluciones generadas aplicando un operador de mutación fuerte a la mejor solución encontrada por el algoritmo hasta el momento; de esta forma se evita la localidad en cierta medida. El operador de mutación consiste en seleccionar un fragmento fijo de la permutación que constituye la solución y desordenarlo de manera aleatoria, del tal manera que la solución resultante sea diferente a la original. A continuación se muestra el pseudocódigo que describe esta idea:

```
function OpMutSblstAleatoria(&Solucion, FraccionSublista) begin
    #Calculamos el tamaño de la sublista
    TamanioSublista ← Dimension/FraccionSublista
    Sublista ← ∅
    #Establecemos la posición de inicio de la sublista de forma
    aleatoria
    PosInicio ← AleatorioEntre(0, Dimension)
    #Extraemos la sublista
    for i ← 0 to TamanioSublista do
        Sublista[i] ← Solucion[(i + PosInicio) %Dimension]
    end
    #Desordenamos la sublista
    BarajarVector(&Sublista)
    #Reinsertamos la sublista
    for i ← 0 to TamanioSublista do
        Solucion[(i + PosInicio) %Dimension] ← Sublista[i]
    end
    #Recalculamos el coste de la solución
    CalcularCoste(&Solucion)
end
```

A continuación se muestra el código que describe el proceso de búsqueda local reiterada (la función C es la función de coste):

```
function ILS(MaxIters, MaxLlamadasLS, FraccSublista) begin
    #Inicializamos la mejor solución de forma aleatoria
    MejorSolucion ← GenerarSolucionAleatoria()

    for i ← 0 to MaxIters do
        #Generamos una nueva solución de forma aleatoria
        SlcnMutada ← OpMutSblstAleatoria(&MejorSolucion, FraccSublista)
        #Lanzamos BL sobre la nueva solución obtenida con el operador
        de mutación
        NuevaSlcn ← BusquedaLocal(SlcnMutada, MaxLlamadasLS)
        #Actualizamos la mejor solución si fuese necesario
        if C(NuevaSlcn) < C(MejorSolucion) then
            MejorSolucion ← NuevaSlcn
        end
    end
    return MejorSolucion
end
```

9. Hibridación ILS-ES

Muy similar a ILS, con la diferencia de que, en lugar de utilizar Búsqueda Local para mejorar las soluciones obtenidas mediante el operador de mutación, emplearemos el algoritmo de Enfriamiento Simulado con este fin. A continuación se muestra el pseudocódigo correspondiente a esta idea (la función C es la función de coste):

```
function ILS-ES(MaxIters, MaxLlamadasLS, FraccSublista ...) begin
    #Inicializamos la mejor solución de forma aleatoria
    MejorSolucion ← GenerarSolucionAleatoria()
    for i ← 0 to MaxIters do
        #Generamos una nueva solución de forma aleatoria
        SlcnMutada ← OpMutSblstAleatoria(&MejorSolucion, FraccSublista)
        #Lanzamos BL sobre la nueva solución obtenida con el operador
        de mutación
        EsSlcn ← EnfriamientoSimulado(SlcnMutada, MaxLlamadasLS ...)
        #Actualizamos la mejor solución si fuese necesario
        if C(NuevaSlcn) < C(MejorSolucion) then
            MejorSolucion ← NuevaSlcn
        end
    end
    return MejorSolucion
end
```

10. 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, así como la de la documentación del lenguaje `C++`.

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

11. Experimentos y Análisis de resultados

11.1. 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 en el guión.

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.

11.2. 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.

11.3. 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.

11.4. Análisis de resultados del Enfriamiento Simulado

Algoritmo de Enfriamiento Simulado					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	42.2802	0.00152593	sko100a	1.78234	0.0138806
chr22a	14.3665	0.00197728	sko100b	2.04068	0.0144491
els19	34.624	0.00412297	sko100c	2.7247	0.0135785
esc32b	13.8095	0.00143775	sko100d	1.91849	0.0143158
kra30b	5.70116	0.00410259	sko100e	2.4	0.014177
lipa90b	22.3519	0.0055648	tai30	11.4635	0.000528993
nug25	3.19444	0.000275109	tai50	5.885	0.00193798
sko56	3.70654	0.00222522	tai60	5.33505	0.00202294
sko64	3.45334	0.00369136	tai256	5.35515	0.00195059
sko72	2.92502	0.00486041	tho150	3.5706	0.019035

Cuadro 3: Tabla que contiene los datos asociados al algoritmo de Enfriamiento Simulado

El algoritmo de enfriamiento simulado resulta eficiente respecto al tiempo, concretamente obtiene mejores resultados que la búsqueda local en lo que a esta medida se refiere. Por otra parte, parece obtener resultados que mejoran a la búsqueda local en los casos de menor dimensión, mientras que en los casos de mayor dimensión es la búsqueda local la que obtiene la ventaja. El algoritmo de ES parece no ser el adecuado para resolver los casos de mayor dimensión, aun así obtiene resultados excelentes respecto al tiempo.

11.5. Análisis de resultados de la Búsqueda Multiarranque Básica

Algoritmo de Búsqueda Multiarranque Básica					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	21.114	0.00336642	sko100a	0.973671	1.01539
chr22a	6.36777	0.00490697	sko100b	1.13276	0.957014
els19	3.85076	0.00393425	sko100c	1.15378	0.958829
esc32b	11.9048	0.0141487	sko100d	1.11756	0.952573
kra30b	2.028	0.0137506	sko100e	1.13711	0.979864
lipa90b	21.207	0.633258	tai30	1.23248	0.0239779
nug25	0.747863	0.00802006	tai50	1.37149	0.0992864
sko56	1.4998	0.12673	tai60	3.53464	0.159128
sko64	1.34603	0.213188	tai256	0.286678	13.5233
sko72	1.30162	0.306443	tho150	1.33283	4.24885

Cuadro 4: Tabla que contiene los datos asociados al algoritmo de Búsqueda Multiarranque Básica

La búsqueda multiarranque básica obtiene soluciones al problema en menos de un segundo excepto para los casos de mayor dimensión, por tanto podemos decir que resulta eficiente en lo que al tiempo se refiere. Por otra parte, como es de esperar, mejora a la búsqueda local respecto a la calidad de las soluciones obtenidas. Observamos una estrecha relación entre BL y BMB, ya que esta segunda consiste en lanzar varias veces la primera, por tanto el tiempo que BMB emplea en obtener la solución es proporcional al empleado por BL; es gracias a este incremento en el tiempo de ejecución que BMB obtiene mejores resultados de BL.

11.6. Análisis de resultados GRASP

Algoritmo GRASP					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	19.0775	0.00545148	sko100a	0.970514	1.54063
chr22a	6.66017	0.00811693	sko100b	1.03373	1.54228
els19	3.54704	0.0057894	sko100c	1.05152	1.57772
esc32b	12.8571	0.0235059	sko100d	1.10205	1.55468
kra30b	2.26427	0.0218917	sko100e	1.16071	1.564
lipa90b	21.131	1.03603	tai30	0.436801	0.025703
nug25	1.13248	0.0123982	tai50	1.05046	0.145528
sko56	1.27924	0.199471	tai60	3.55004	0.247251
sko64	1.40294	0.302667	tai256	0.312504	39.2974
sko72	1.2986	0.466713	tho150	1.3544	6.80024

Cuadro 5: Tabla que contiene los datos asociados al algoritmo GRASP

El algoritmo GRASP es capaz de obtener soluciones menos condicionadas a los óptimos locales que todos los considerados anteriormente en este documento. Esta mejora, claro está, implica un incremento en el tiempo de ejecución que se hace más notable a medida que incrementamos la dimensión del problema. Aún con todo, GRASP no parece ser el

método adecuado para la resolución del problema QAP, puesto que en la mayoría de las ocasiones las mejoras obtenidas respecto al resto de algoritmos es a costa de un sobrecoste en el tiempo que no es proporcional a la mejora obtenida.

11.7. Análisis de resultados de la Búsqueda Local Reiterada

Algoritmo de Búsqueda Local Reiterada					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	17.5457	0.0026577	sko100a	0.491046	0.85334
chr22a	6.62118	0.00423482	sko100b	0.579895	0.825253
els19	6.14337	0.0029313	sko100c	0.76639	0.768706
esc32b	11.4286	0.0100749	sko100d	0.868856	0.837173
kra30b	1.48108	0.0104997	sko100e	0.762186	0.78429
lipa90b	21.104	0.60355	tai30	1.49411	0.0141493
nug25	0.459402	0.00622479	tai50	1.08169	0.0831006
sko56	1.16664	0.110512	tai60	3.11625	0.136273
sko64	0.780238	0.17724	tai256	0.290796	7.93192
sko72	0.827095	0.267107	tho150	0.891662	3.59569

Cuadro 6: Tabla que contiene los datos asociados al algoritmo de Búsqueda Local Reiterada

El algoritmo de búsqueda local reiterada es el que obtiene mejores resultados, en lo que a calidad de las soluciones se refiere, de entre todos los considerados en este documento. En cuanto al tiempo vemos que, en general, mejora a BMB, que era el método que hasta ahora obtenía mejores resultados. Por tanto parece ser éste el método adecuado para resolver el problema QAP en cuanto lo que desviación respecto al óptimo se refiere.

11.8. Análisis de resultados del híbrido ILS-LS

Algoritmo híbrido ILS-ES					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
chr20b	17.5979	0.0663293	sko100a	0.74578	0.285966
chr22a	6.13385	0.0377989	sko100b	0.753005	0.281156
els19	15.0306	0.0929369	sko100c	0.822929	0.278866
esc32b	0	0.0586142	sko100d	0.913516	0.277938
kra30b	1.21418	0.125549	sko100e	0.950989	0.294719
lipa90b	21.5952	0.142827	tai30	2.59364	0.00947199
nug25	0.512821	0.00915123	tai50	1.54013	0.0432994
sko56	1.18521	0.0569492	tai60	4.15979	0.0468222
sko64	0.848695	0.0792185	tai256	2.35326	0.051929
sko72	0.919464	0.112909	tho150	0.965382	0.794868

Cuadro 7: Tabla que contiene los datos asociados al algoritmo híbrido ILS-ES

El híbrido ILS-ES consiste en aplicar ES como método de mejora para las soluciones obtenidas mediante el operador de mutación de ILS, por tanto, es lógico pensar que las diferencias entre ES y BL se verán reflejadas en este algoritmo. Vemos que, como esperábamos, este algoritmo mejora en tiempo a ILS, no es así respecto a la calidad de las

soluciones que, aunque peores que las obtenidas con ILS, siguen siendo soluciones cercanas a la óptima.

11.9. 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
ES	9,4444055	0,0065521532
BMB	4,2320321	1,1932433035
GRASP	4,13365345	2,865633258
ILS	3,8950093	0,824777754
ILS-ES	4,04181705	0,1601281345

Cuadro 8: Tabla que contiene los datos asociados al análisis de resultados medios generales

Tras el análisis detallado de cada uno de los algoritmos considerados en este documento, analizamos la media de los resultados obtenidos por cada uno de ellos. Tal y como hemos visto en el análisis de ILS, es este algoritmo el que obtiene los mejores resultados respecto a la calidad de las soluciones, por tanto, parece ser el método para evitar la localidad que implementa este algoritmo el más adecuado para el problema de la asignación cuadrática

Respecto al tiempo, podemos comparar BL y ES dada la similaridad en las soluciones obtenidas; vemos que ES obtiene soluciones de calidad similar a BL en tiempo menor, por tanto si es esta última característica la que consideramos como más relevante deberá ser ES el que utilicemos para resolver el problema. Lo mismo sucede al comparar ILS-ES e ILS, siendo esta última la que obtiene mejores resultados en lo que a desviación respecto al óptimo se refiere.

12. 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 binario `mainGeneral` para cada una de las instancias del problema y almacenará el resultado en el directorio `results` bajo el nombre de `<nombre_instancia>.rslt`.