Metaheurísticas (Curso 2021-2022)

Grado en Ingeniería Informática
Universidad de Granada

UNIVERSIDAD DE GRANADA

Práctica 2: Técnicas de Búsqueda basadas en Poblaciones

Problema a: Mínima Dispersión Diferencial

Pedro Bedmar López - 75935296Z
pedrobedmar@correo.ugr.es

Grupo de prácticas 3 - Martes 17:30-19:30

Índice

Parte I

Formulación del problema

Sea $G = (V, E)$ un grafo completo no dirigido donde V, de tamaño n, es el conjunto de vértices que lo forman y E es el conjunto de las aristas que unen estos vértices. Este grafo es un grafo ponderado, ya que cada una de las aristas $e_{u,v} \in E$ lleva asociada un peso que representa la distancia $d_{u,v}$ entre dos vértices $u, v \in V$.

La dispersión es una medida que se puede aplicar en este dominio, donde dado un subconjunto $S \subset V$ de tamaño m se mide cómo de homogéneas son las distancias entre los vértices que forman S. Una de las aplicaciones más importantes de las Ciencias de la Computación consiste en optimizar valores como éste, maximizando o minimizando el resultado que devuelve una **función objetivo**.

En esta práctica queremos minimizar su valor, obteniendo la mínima dispersión. Este problema tiene un gran paralelismo con problemas reales, como puede ser la organización del género en almacenes, donde minimizar la dispersión de la mercancía reduce los costes. Por tanto, si resolvemos este problema de forma teórica es trivial aplicar la solución en estos casos.

Anteriormente he definido la dispersión de una forma muy genérica, sin entrar en su formalización. Y es que se puede definir de diferentes formas, teniendo en cuenta la dispersión media de los elementos del conjunto S o utilizando los valores extremos (máximos y mínimos) en éste. Esta segunda opción se define formalmente como:

$$diff(S) = max_{i \in S}\{\sum_{j \in S} d_{i,j}\} - min_{i \in S}\{\sum_{j \in S} d_{i,j}\}$$

Utilizando esta definición de dispersión como función objetivo obtenemos lo que se conoce como **Problema de la Mínima Dispersión Diferencial (MDD)**, es decir:

$$S^* = argmin_{S \subset V} diff(S)$$

Parte II

Descripción de la aplicación de los algoritmos

En esta segunda práctica de la asignatura implementamos y comparamos el rendimiento de algoritmos de búsqueda basados en poblaciones, en concreto algoritmos **Genéticos** y **Meméticos**. Antes de describirlos, vamos a comentar información común a ambos.

1. Datos utilizados

Los datos que necesitamos para analizar el comportamiento de los algoritmos en este problema no son muy complejos. En cada posible instancia se necesita conocer el valor n indicando el número de puntos que contiene el dataset, el valor $m < n$ indicando cuantos puntos se quieren escoger de forma que se minimice la dispersión en esos m puntos y la matriz d con tamaño $n \times n$, simétrica y con valor 0 en su diagonal, que contiene las distancias entre cada uno de los n puntos del dataset. En definitiva, se necesita conocer el grafo G.

En total, en los experimentos utilizamos 50 instancias diferentes con datos extraídos del dataset **GKD**. Las instancias toman valores $n \in \{25, 50, 100, 125, 150\}$ y $m \in [2, 45]$.

2. Representación de soluciones

El conjunto V descrito en la formulación del problema coincide con n en tamaño. Al contrario que en la práctica anterior, una posible solución S puede representarse de dos formas diferentes según el punto de ejecución en que se encuentre el algoritmo.

En la búsqueda local del algoritmo memético, la solución se representa de la misma forma que en la práctica anterior para poder reutilizar el código, siendo S es una solución válida si:

- $|S| = m$
- $S \subset V$

Y por tanto, $m < n$. Estamos ante una representación entera donde los índices de los vértices que forman la solución son recogidos por S. Por ejemplo $S = \{3, 7, 2\}$ representa que los vértices con índice 3, 7 y 2 forman una solución al problema.

En cambio, para la mayor parte del algoritmo memético y todo el genético, la representación que se utiliza es binaria. Para diferenciar la representación anterior de ésta, vamos a nombrarla S_b en vez de S. Cumple las siguientes condiciones:

- $|S_b| = n$
- count(1, S_b) = m, donde count(1, S_b) devuelve el número de 1s en S_b.

- $S_b = [x|x \in \{0, 1\}]$

El orden de los valores en S_b importa, cada uno representa un vértice del grafo, por lo que se implementará como un vector y no como un conjunto. Por ejemplo, $S_b = [1, 0, 0, 1, 0, 1]$ representa que los vértices 1, 4 y 6 del grafo forman parte de la solución pero el resto no.

3. Función objetivo

Como hemos comentado al describir el problema, la función objetivo a minimizar se define como:

$$diff(S) = max_{i \in S}\{\sum_{j \in S} d_{i,j}\} - min_{i \in S}\{\sum_{j \in S} d_{i,j}\}$$

En el caso de utilizar representación entera para las posibles soluciones, el pseudocódigo quedaría de la siguiente forma:

Algorithm 1 Función objetivo, implementación para la representación entera

```
max ← −∞
min ← ∞
for s ∈ S do
    distance ← Σ_{s2∈S} d_{s,s2}
    if distance > max then
        max ← distance
    end if
    if distance < min then
        min ← distance
    end if
end for
return max − min
```

En el algoritmo de Búsqueda Local, no se utiliza directamente esta implementación de la función objetivo. Esto se debe a que es costosa, en concreto tiene una complejidad computacional de $O(n^2)$. Utilizamos una versión factorizada de la función, que reutiliza cálculos de iteraciones previas para actualizar el valor de la dispersión. De esta forma, obtenemos una complejidad de $O(n)$.

En las situaciones en las que las soluciones se representen de forma binaria, el pseudocódigo de la función objetivo quedaría así:

Algorithm 2 Función objetivo, implementación para la representación binaria

```
max ← −∞
min ← ∞
for i ∈ S_b do                          ▷ Donde i indica la posición en el vector S_b
    if S_b[i] == 1 then
        distance ← 0

        for j ∈ S_b do
            if S_b[j] == 1 then         ▷ Donde j indica la posición en el vector S_b
                distance+ = d_{i,j}
            end if
        end for

        if distance > max then
            max ← distance
        end if
        if distance < min then
            min ← distance
        end if
    end if
end for
return max − min
```

4. Operadores comunes

En esta sección se describen los operadores comunes a los diferentes algoritmos.

4.1. Generador de soluciones aleatorias

Algorithm 3 Generador de soluciones aleatorias en representación binaria.

```
procedure GENERATERANDOMSOLUTION
    individual ← [ ]
    for i = 0 to m − 1 do
        individual+ = {true}            ▷ Insertamos un true al final del vector
    end for
    for i = 0 to n − m − 1 do
        individual+ = {false}
    end for
    shuffle(individual)
    return individual
end procedure
```

4.2. Mecanismos de selección

Algorithm 4 Mecanismo de selección del algoritmo genético generacional. Toma como argumentos un vector que contiene la población de individuos y otro vector que contiene la dispersión de cada individuo. Devuelve la población que contiene a los padres, de tamaño 50.

```
procedure GENERATIONALSELECTIONOPERATOR(popul, populDisp)
    parentsPopul ← [ ]
    parentsPopulDisp ← [ ]

    for i = 0 to 49 do
        firstRandElem ←rand(0, 49)
        secondRandElem ←rand(0, 49)          ▷ Debe de ser diferente a firstRandElem

        bestRandElem ← firstRandElem
        if populDisp[bestRandElem] > populDisp[secondRandElem] then
            bestRandElem ← secondRandElem
        end if

        parentsPopul+ = {popul[bestRandElem]}
        parentsPopulDisp+ = {populDisp[bestRandElem]}
    end for

    return parentsPopul, parentsPopulDisp
end procedure
```

Algorithm 5 Mecanismo de selección del algoritmo genético estacionario. Toma como argumentos un vector que contiene la población de individuos y otro vector que contiene la dispersión de cada individuo. Devuelve el índice en la población de los dos individuos elegidos como padres.

```
procedure STATIONARYSELECTIONOPERATOR(popul, populDisp)
    parents ← [ ]

    firstRandElem ← rand(0, 49)
    secondRandElem ← rand(0, 49)             ▷ Debe de ser diferente a firstRandElem
    parent1Index ← firstRandElem
    if populDisp[parent1Index] > populDisp[secondRandElem] then
        parent1Index ← secondRandElem
    end if

    firstRandElem ← rand(0, 49)
    secondRandElem ← rand(0, 49)             ▷ Debe de ser diferente a firstRandElem
    parent2Index ← firstRandElem
    if populDisp[parent2Index] > populDisp[secondRandElem] then
        parent2Index ← secondRandElem
    end if

    return parent1Index, parent2Index
end procedure
```

4.3. Operadores de cruce

Algorithm 6 Operador de cruce uniforme. Cruza dos individuos, generando un hijo que mantiene los genes comunes y asigna uno valor aleatorio en aquellos no comunes. Si el hijo no es factible, se repara. Devuelve este hijo.

```
procedure UNIFORMCROSSOVEROPERATOR(parent1, parent2)
    child ← parent1

    for i = 0 to n − 1 do
        if parent1[i] != parent2[i] then
            child[i] ← rand(0, 1)
        end if
    end for

    countNbTrues ← 0
    for i = 0 to n − 1 do
        if child[i] then
            countNbTrues = countNbTrues + 1
        end if
    end for

    while countNbTrues > m do
        accDistance ← [ ]
        for i = 0 to n − 1 do
            sum ← 0
            for j = 0 to n − 1 do
                if child[i] and child[j] then
                    sum+ = Distance(i, j)    ▷ Distance(x,y) devuelve la distancia entre x e y
                end if
                accDistance+ = {sum}
            end for
        end for

        avg ← 0
        avg ← forall i in accDistance do avg + i
        avg ← avg/len(accDistance)

        max ← −∞
        maxPosition ← −1
        for i = 0 to n − 1 do
            if child[i] then
                sum ← 0

                for j = 0 to n − 1 do
                    sum+ = Distance(i, j)
                end for
```

```
            if abs(sum − avg) > max then
                max ← abs(sum − avg)              ▷ abs(x) devuelve el valor absoluto de x
                maxPosition ← i
            end if
        end if
    end for

    child[maxPosition] = false
    countNbTrues = countNbTrues − 1
end while

while countNbTrues < m do
    accDistance ← []
    for i = 0 to n − 1 do
        sum ← 0
        for j = 0 to n − 1 do
            if child[i] and child[j] then
                sum+ = Distance(i, j)    ▷ Distance(x,y) devuelve la distancia entre x e y
            end if
            accDistance+ = {sum}
        end for
    end for

    avg ← 0
    avg ← forall i in accDistance do avg + i
    avg ← avg/len(accDistance)

    min ← ∞
    minPosition ← −1
    for i = 0 to n − 1 do
        if !child[i] then
            sum ← 0

            for j = 0 to n − 1 do
                sum+ = Distance(i, j)
            end for

            if abs(sum − avg) < min then
                min ← abs(sum − avg)              ▷ abs(x) devuelve el valor absoluto de x
                minPosition ← i
            end if
        end if
    end for

    child[minPosition] ← true
    countNbTrues ← countNbTrues + 1
end while
return child
end procedure
```

Algorithm 7 Operador de cruce basado en posición. Cruza dos individuos, generando un hijo que mantiene los genes comunes y asigna asigna los de un padre cualquiera en aquellos no comunes, en un orden aleatorio. Devuelve este hijo.

```
procedure POSITIONBASEDCROSSOVEROPERATOR(parent1, parent2)
    child ← parent1

    notEqualIndexes ← [ ]
    for i = 0 to n − 1 do
        if parent1[i]! = parent2[i] then
            notEqualIndexes+ = i
        end if
    end for

    shuffle(notEqualIndexes)
    pos ← 0
    for i = 0 to n − 1 do
        if parent1[i]! = parent2[i] then
            child[i]parent1[notEqualIndexes[pos]]
            pos ← pos + 1
        end if
    end for
    return child
end procedure
```

4.4. Operador de mutación

Algorithm 8 Operador de mutación de un individuo. Toma como argumento el individuo a mutar. Intercambia el valor de dos genes en el individuo, y devuelve el individuo modificado.

```
procedure MUTATIONOPERATOR(individual)
    position ← rand(0, n − 1)
    position2 ← rand(0, n − 1)
    value ← individual[position]
    value2 ← individual[position2]

    while position == position2||value == value2 do
        position2 = rand(0, n − 1)
        value2 = individual[position2]
    end while

    individual[position] ←!value
    individual[position2] ←!value2
    return individual
end procedure
```

Parte III

Pseudocódigo de los algoritmos

En los algoritmos basados en poblaciones, nos inspiramos en la naturaleza para simular la evolución de una especie. En ella, una población tiene sucesores, que heredarán sus características. Sólo sobrevivirán los individuos mas fuertes, mejor adaptados al medio, en nuestro caso equivaldría a aquellos que minimizen la función objetivo.

Existen diferentes algoritmos basados en poblaciones, siendo dos de ellos los Genéticos y los Meméticos.

5. Algoritmos Genéticos

En ellos, se parte de una población inicial aleatoria de la que se seleccionan unos progenitores. Estos se cruzarán dando lugar a unos descendientes que heredarán sus genes. Durante el proceso también se podrán producir mutaciones de estos hijos, que finalmente reemplazarán la población inicial. El proceso se repite de forma iterativa hasta llegar a un número máximo de evaluaciones de la función objetivo.

5.1. Algoritmo Genético Generacional

En esta versión de algoritmo genético, se realizan tantas selecciones como individuos tiene la población. De esta selección, los progenitores se cruzan con una probabilidad determinada, generando un conjunto de descendientes. La población de los hijos sustituye a la población anterior, sólamente se mantiene el mejor individuo.

Algorithm 9 Estructura de búsqueda del Algoritmo Genético Generacional.

```
procedure GENERATIONALMODEL
    numEvaluations ← 0
    population ← [ ]
    populDisp ← [ ]
    for i = 0 to 49 do                    ▷ Se genera una población con individuos aleatorios
        individual ← generateRandomSolution()
        population+ = {individual}
        populDisp+ = {Dispersion(individual)}
        numEvaluations ← numEvaluations + 1
    end for

    lastBestSolDisp ← ∞
    for i = 0 to 49 do
        if lastBestSolDisp > populDisp[i] then
            lastBestSol ← population[i]
            lastBestSolDisp ← populDisp[i]
        end if
    end for

    while numEvaluations < 100000 do
        population, populDisp, numEvaluations ← generationalModelEvolution(population,
        populDisp, numEvaluations)

        population, populDisp, lastBestSol, lastBestSolDisp ←
        generationalModelReplacement(population, populDisp, lastBestSol, lastBestSolDisp)
    end while

    return lastBestSol
end procedure
```

Algorithm 10 Esquema de evolución del Algoritmo Genético Generacional. En él, crossoverOperator puede referirse al operador de cruce uniforme o al basado en posición.

```
procedure GENERATIONALMODELEVOLUTION(population, populDisp, numEvals)
    population, populDisp ← generationalSelectionOperator(population, populDisp)
    for i = 0 to 49 * 0,7 do
        child1 ← crossoverOperator(population[i], population[i + 1])
        child2 ← crossoverOperator(population[i], population[i + 1])
        population[i] = child1
        population[i + 1] = child2
    end for

    for i = 0 to 49 * 0,1 do
        population[i] ←mutationOperator(population[i])
    end for

    for i = 0 to 49 do
        population[i] ←Dispersion(population[i])
        numEvals + +
    end for

    return population, populDisp, numEvals
end procedure
```

Algorithm 11 Esquema de reemplazamiento del Algoritmo Genético Generacional

```
procedure GENERATIONALMODELREPLACEMENT(population, populDisp, lastBestSol,
lastBestSolDisp)
    higherDispersion ← −∞
    for i ← 0 to 49 do
        if higherDispersion < populDisp[i] then
            worstSolutionIndex ← i
            higherDispersion ← populDisp[i]
        end if
    end for
    population[worstSolutionIndex] ← lastBestSol
    populDisp[worstSolutionIndex] ← lastBestSolDisp

    lowerDispersion ← −∞
    for i ← 0 to 49 do
        if higherDispersion < populDisp[i] then
            bestSolutionIndex ← i
            lowerDispersion ← populDisp[i]
        end if
    end for
    lastBestSol ← population[worstSolutionIndex]
    lastBestSolDisp ← lowerDispersion

    return population, populDisp, lastBestSol, lastBestSolDisp
end procedure
```

5.2. Algoritmo Genético Estacionario

Algorithm 12 Estructura de búsqueda del Algoritmo Genético Estacionario.

```
procedure STATIONARYMODEL
    numEvaluations ← 0
    population ← [ ]
    populDisp ← [ ]
    for i = 0 to 49 do                    ▷ Se genera una población con individuos aleatorios
        individual ← generateRandomSolution()
        population+ = {individual}
        populDisp+ = {Dispersion(individual)}
        numEvaluations ← numEvaluations + 1
    end for

    while numEvaluations < 100000 do
        child1, child2 ← stationaryModelEvolution(population, populDisp)

        population, populDisp, numEvaluations ← stationaryModelReplacement(population,
        populDisp, child1, child2, numEvaluations)
    end while

    lowerDispersion ← ∞
    for i = 0 to 49 do
        if populDisp[i] < lowerDispersion then
            lowerDisp ← populDisp[i]
            bestIndividual ← popul[i]
        end if
    end for
    return bestIndividual
end procedure
```

Algorithm 13 Esquema de evolución del Algoritmo Genético Estacionario. En él, crossoverOperator puede referirse al operador de cruce uniforme o al basado en posición.

```
procedure STATIONARYMODELEVOLUTION(population, populDisp)
    parent1Index, parent2Index ← stationarySelectionOperator(population, populDisp)

    child1 ← crossoverOperator(population[parent1Index], population[parent2Index])
    child2 ← crossoverOperator(population[parent1Index], population[parent2Index])

    if Rand(0, 100 * 0,1 − 1) == 0 then          ▷ 10 % de probabilidad de mutar
        child1 ← mutationOperator(child1)
    end if

    if Rand(0, 100 * 0,1 − 1) == 0 then
        child2 ← mutationOperator(child2)
    end if
    return child1, child2
end procedure
```

Algorithm 14 Esquema de reemplazamiento del Algoritmo Genético Estacionario

procedure STATIONARYMODELREPLACEMENT(population, populDisp, child1, child2, numEval)

$higherDispersion \leftarrow -\infty$
$secondHigherDisp \leftarrow -\infty$

for $i \leftarrow 0$ **to** 49 **do**
  if $higherDispersion < populDisp[i]$ **then**
    $worstIndividualIndex \leftarrow i$
    $higherDispersion \leftarrow populDisp[i]$
  end if
end for
for $i \leftarrow 0$ **to** 49 **do**
  if $higherDispersion < populDisp[i]$ && $i! = worstIndividualIndex$ **then**
    $secondWorstIndividualIndex \leftarrow i$
    $secondHigherDisp \leftarrow populDisp[i]$
  end if
end for

$disp1 \leftarrow$ Dispersion($child1$)
$disp2 \leftarrow$ Dispersion($child2$)
$numEval \leftarrow numEval + 2$
if $disp1 < disp2$ **then**
  if $disp2 < higherDispersion$ **then**
    $population[worstIndividualIndex] = child2$
    $populDisp[worstIndividualIndex] = disp2$

    $population[secondWorstIndividualIndex] = child1$
    $populDisp[worstIndividualIndex] = disp1$

  else if $disp1 < higherDispersion$ **then**
    $population[secondWorstIndividualIndex] = child1$
    $populDisp[worstIndividualIndex] = disp1$
  end if
else
  if $disp1 < higherDispersion$ **then**
    $population[worstIndividualIndex] = child1$
    $populDisp[worstIndividualIndex] = disp1$

    $population[secondWorstIndividualIndex] = child2$
    $populDisp[worstIndividualIndex] = disp2$

  else if $disp1 < higherDispersion$ **then**
    $population[secondWorstIndividualIndex] = child2$
    $populDisp[worstIndividualIndex] = disp2$
  end if
end if

return $population, populDisp, numEval$
end procedure

6. Algoritmo Memético

6.1. Estructura de búsqueda del Algoritmo Memético

Algorithm 15 Estructura de búsqueda del Algoritmo Memético.

```
procedure STATIONARYMODEL
    numEvaluations ← 0
    population ← [ ]
    populDisp ← [ ]
    for i = 0 to 49 do                    ▷ Se genera una población con individuos aleatorios
        individual ← generateRandomSolution()
        population+ = {individual}
        populDisp+ = {Dispersion(individual)}
        numEvaluations ← numEvaluations + 1
    end for

    while numEvaluations < 100000 do
        child1, child2 ← stationaryModelEvolution(population, populDisp)

        population, populDisp, numEvaluations ← stationaryModelReplacement(population,
        populDisp, child1, child2, numEvaluations)
    end while

    if numEvaluations %10 == 0 then
        if memeticType == AM1.0 then
            for i = 0 to 49 do
                population[i], populDisp[i], numEvaluations ←
                localSearchExecution(population[i], populDisp[i], numEvaluations)
            end for

        else if memeticType == AM0.1 then
            idx ← RandIndiv(5)            ▷ Devuelve 5 índices de individuos de la población
            for i = 0 to 5 do
                population[idx[i]], populDisp[idx[i]], numEvaluations ←
                localSearchExecution(population[idx[i]], populDisp[idx[i]], numEvaluations)
            end for

        else if memeticType == AM0.1mej then
            idx ← BestIndiv(5)            ▷ Índices de los 5 mejores individuos de la población
            for i = 0 to 5 do
                population[idx[i]], populDisp[idx[i]], numEvaluations ←
                localSearchExecution(population[idx[i]], populDisp[idx[i]], numEvaluations)
            end for
        end if
    end if
```

```
    lowerDispersion ← ∞
    for i = 0 to 49 do
        if populDisp[i] < lowerDispersion then
            lowerDisp ← populDisp[i]
            bestIndividual ← popul[i]
        end if
    end for
    return bestIndividual
end procedure
```

6.2. Enlace entre el algoritmo memético y la BL

Algorithm 16 Enlace entre el algoritmo memético y la BL

```
procedure LOCALSEARCHEXECUTION(individual, dispersion, numEvaluations)
    solution ← [ ]
    unselectedItems ← [ ]

    for i = 0 to 49 do
        if individual[i] then
            solution+ = {i}
        else
            unselectedItems+ = {i}
        end if
    end for

    unselectedItems, solution, dispersion, numEvaluations ←
    localSearch(unselectedItems, solution, dispersion, numEvaluations)
    individual < −[ ]                                  ▷ Inicializado a false
    for i = 0 to 49 do
        individual[solution[j]] = true
    end for
end procedure
```

6.3. Búsqueda Local

Algorithm 17 Algoritmo Búsqueda Local primero el mejor

```
1: procedure localSearch(U, S, Sdispersion, eval)
2:     sum ← [ ]
3:
4:     for i = 0 to m − 1 do
5:         element ← U.last
6:         U ← U − {element}
7:         S ← S + {element}
8:     end for
9:
10:    S_best ← S
11:    current_cost ←Dispersion(S)
12:    best_cost ← current_cost
13:
14:    better_solution ← true
15:    STOP = eval + 10000
16:
17:    while eval < 400 and better_solution and eval < STOP do
18:        better_solution ← false
19:
20:        for u ∈ S and while !better_solution and eval < 400 and eval < STOP do
21:            for v ∈ U and while !better_solution and eval < 400 and eval < STOP do
22:                eval ← eval + 1
23:                δ ← [ ]                                  ▷ Array inicializado a 0
24:                δ(w)_max ← −∞
25:                δ(w)_min ← ∞
26:
27:                for w ∈ S do
28:                    if w! = u then
29:                        δ[w] ← sum[w] − d_{w,u} + d_{w,v}
30:                        δ[v] += d_{w,v}
31:
32:                        if δ[w] > δ(w)_max then
33:                            δ(w)_max ← δ[w]
34:                        end if
35:                        if δ[w] < δ(w)_min then
36:                            δ(w)_min ← δ[w]
37:                        end if
38:                    end if
39:                end for
```

```
40:
41:                 δ_max ← max(δ[v], δ(w)_max)
42:                 δ_min ← min(δ[v], δ(w)_min)
43:                 new_cost ← δ_max − δ_min
44:                 if new_cost < current_cost then
45:                     best_cost ← new_cost
46:                     current_cost ← new_cost
47:
48:                     swap ← u                          ▷ intercambio u y v en S y U
49:                     u ← v
50:                     v ← swap
51:                     better_solution = true
52:                     best_solution = S
53:                 end if
54:
55:             end for
56:         end for
57:
58:         shuffle(S)
59:         shuffle(U)
60:     end while
61:
62:     return U, S, best_cost,
63: end procedure
```

Parte IV

Procedimiento considerado para desarrollar la práctica

La implementación de los algoritmos ha sido realizada en C++, concretamente en su versión de 2017. Para ello, hemos creado un proyecto con la siguiente estructura:

```
/
├── BIN ..... archivos ejecutables
│   └── practica1
├── data ..... ficheros .txt con los datos de entrada
│   ├── data_index.txt ..... índice con los nombres de los archivos de datos
│   └── ...
├── doc
├── FUENTES
│   ├── DataLoader.cpp ..... clase encargada de cargar los datos de los ficheros
│   ├── DataLoader.h
│   ├── functions.cpp ..... funciones auxiliares
│   ├── functions.h
│   ├── GreedyAlgorithm.cpp ..... implementación del algoritmo Greedy
│   ├── GreedyAlgorithm.h
│   ├── LocalSearchAlgorithm.cpp.....implementación del algoritmo BL
│   ├── LocalSearchAlgorithm.h
│   ├── GeneticAlgorithm.cpp.....implementación del algoritmo Genético
│   ├── GeneticAlgorithm.h
│   ├── MemeticAlgorithm.cpp.....implementación del algoritmo Memético
│   ├── MemeticAlgorithm.h
│   └── practica2.cpp.....archivo desde donde se inicia la ejecución
├── obj ..... ficheros objeto
├── makefile
└── LEEME
```

Se ha partido desde cero, sin utilizar ningún framework de metaheurísticas ni librería adicional a las que vienen incluídas en el propio C++. Para la generación de números aleatorios, se utiliza la librería $< random >$ incluida en el lenguaje. La semilla utilizada en los experimentos es el número 1. El equipo donde se han realizado las pruebas es un MacBook Pro de 15 pulgadas del año 2015, con CPU Intel Core i7 2.5 GHz I7-4870HQ y 16 GB de RAM. Utiliza el sistema operativo macOS Big Sur 11.6.1.

Para ejecutar el código, nos situamos en la raíz del proyecto y ejecutamos *make* en la terminal. A continuación, ejecutamos:

*./bin/practica*2 $<semilla>$ $<algoritmo>$ $<fichero\_datos>$

Donde $<algoritmo>$ puede tomar como valor g (Genético) o m (Memético). Se ejecutan por defecto la versión del genético estacionario con operador uniforme o la del memético AM-(10,0.1).

Ejemplo: ./bin/practica2 1 g data/GKD-b_50_n150_m45.txt

Parte V

Experimentos y análisis de resultados

Para comprobar el funcionamiento de los algoritmos, realizamos experimentos de ejecución. Nuestros algoritmos son probabilísticos, ya que la aleatoriedad está presente en ellos. Por tanto, para que los resultados sean reproducibles es necesario fijar una semilla. Como mencionamos en el apartado anterior, fijamos su valor en 1.

Vamos a ejecutar el algoritmo con los 50 casos que tenemos.

7. Algoritmos Genéticos

Encontramos 4 versiones diferentes de este algoritmo. Estas versiones vienen dadas de combinar dos tipos de modelos evolutivos (generacional y estacionario) con dos tipos de operadores de cruce (uniforme y basado en posición).

La versión que mejores resultados devuelve es la que utiliza el modelo estacionario con el operador de cruce uniforme. Esto es así ya que el operador uniforme genera hijos con mayor parecido a los padres que el basado en posición, y por tanto las soluciones buenas se heredan con mayor facilidad. También, el estacionario puede tener ventaja sobre el generacional ya que no se actualiza toda la población, sino sólo dos elementos. En el generacional, al ser elitista sólo mantiene el mejor individuo, perdiendo el segundo, tercero, cuarto... mejores.

Con los tiempos se puede observar que el operador uniforme es más costoso que el de posición, ya que tiene que reparar las soluciones que no son factibles. El modelo generacional también es un poco más lento, ya que selecciona y cruza toda la población.

Como desviación y tiempo de ejecución medio entre todos los casos encontramos lo siguiente para cada tipo de algoritmo genético:

AGG - uniforme	
Media Desv:	**51.71**
Media Tiempo:	**0.674**
AGG - posición	
Media Desv:	**51.62**
Media Tiempo:	**0.37**
AGE - uniforme	
Media Desv:	**49.14**
Media Tiempo:	**0.36**
AGE - posición	
Media Desv:	**54.51**
Media Tiempo:	**0.24**

A continuación, desglosamos los resultados para cada caso:

AGG - uniforme			AGG - posición		
Coste medio obtenido	**Desv**	**Tiempo**	**Coste medio obtenido**	**Desv**	**Tiempo**
0.0000	0.00	0.2502	0.00	0.00	0.11
0.0000	0.00	0.2490	0.00	0.00	0.15
0.0000	0.00	0.3279	0.00	0.00	0.16
0.0000	0.00	0.3396	0.00	0.00	0.18
0.0000	0.00	0.2877	0.00	0.00	0.17
43.9470	71.06	0.1548	33.25	61.75	0.11
29.7923	52.68	0.1566	20.96	32.72	0.09
36.2664	53.78	0.1460	28.45	41.09	0.10
26.7920	36.29	0.1265	35.98	52.57	0.12
43.7322	46.80	0.1255	51.30	54.65	0.09
8.3003	76.79	0.2352	13.47	85.70	0.11
13.4187	84.19	0.1985	9.95	78.68	0.14
13.6651	82.71	0.1751	10.85	78.22	0.15
13.3118	87.51	0.2063	14.39	88.44	0.15
3.3944	15.95	0.1670	14.45	80.26	0.12
175.8146	75.69	0.3039	74.68	42.76	0.21
147.2324	67.33	0.2791	60.04	19.87	0.23
109.8350	60.67	0.2872	76.78	43.74	0.21
103.7483	55.26	0.3544	77.22	39.90	0.20
125.8610	62.09	0.3512	81.51	41.46	0.21
36.7919	62.40	0.5436	32.94	58.01	0.25
30.1768	54.72	0.5287	33.92	59.72	0.24
40.9040	62.48	0.4944	37.41	58.98	0.25
41.7840	79.32	0.4765	39.38	78.06	0.25
46.5589	63.06	0.5334	31.66	45.67	0.25
305.4437	44.76	1.1550	270.79	37.69	0.54
250.2183	49.21	1.0812	282.60	55.03	0.54
175.8143	39.49	0.8751	288.82	63.17	0.52
303.1423	54.66	0.9472	333.77	58.82	0.52
233.1540	45.32	1.1646	258.69	50.72	0.47
33.1199	64.54	0.6471	27.42	57.17	0.29
42.4134	55.70	0.6402	39.04	51.87	0.32
34.1503	45.74	0.6609	60.10	69.17	0.31
51.5231	62.18	0.6205	51.96	62.49	0.31
33.1647	45.39	0.5096	65.11	72.18	0.32
418.8508	62.89	0.9900	353.30	56.00	0.71
389.8804	48.99	1.0288	562.06	64.61	0.64
306.4452	38.66	1.0654	453.29	58.53	0.67
535.2395	68.50	1.1277	369.14	54.33	0.72
376.3967	52.66	1.2665	431.29	58.68	0.68
64.6029	63.86	0.9410	52.35	55.40	0.44
69.4484	61.43	0.9388	56.80	52.83	0.40
97.5662	72.58	1.1850	64.64	58.61	0.43
48.3420	46.35	0.9940	84.39	69.27	0.44
61.9831	55.19	0.7309	53.19	47.78	0.37
466.8619	51.22	1.5968	532.56	57.24	0.90
615.8520	62.88	1.3057	511.01	55.26	0.96
376.4580	39.77	1.3344	506.25	55.21	0.90
464.3460	51.24	1.9691	612.89	63.06	0.93
513.4804	51.54	1.6356	537.48	53.70	0.98

AGE - uniforme			AGE - posición		
Coste medio obtenido	**Desv**	**Tiempo**	**Coste medio obtenido**	**Desv**	**Tiempo**
0.0000	0.00	0.3901	0.0000	0.00	0.0863
0.0000	0.00	0.3801	0.0000	0.00	0.0856
0.0000	0.00	0.3888	0.0000	0.00	0.0907
0.0000	0.00	0.3838	0.0000	0.00	0.0912
0.0000	0.00	0.4100	0.0000	0.00	0.0848
44.5784	71.47	0.0727	31.2907	59.36	0.0504
26.5340	46.87	0.0701	33.9302	58.45	0.0517
29.1174	42.44	0.0714	21.2857	21.26	0.0494
26.7920	36.29	0.0847	38.4549	55.61	0.0486
44.5938	47.83	0.0697	30.9835	24.91	0.0498
14.3161	86.55	0.1126	13.1999	85.41	0.0719
13.8060	84.64	0.1027	12.7428	83.35	0.0671
10.0954	76.60	0.0964	20.4631	88.46	0.0654
12.7417	86.95	0.1034	9.9325	83.26	0.0713
8.1204	64.86	0.0953	15.4737	81.56	0.0649
91.4226	53.24	0.1676	126.2039	66.13	0.1107
64.6603	25.60	0.1744	93.9266	48.78	0.1132
104.0179	58.47	0.1757	90.8071	52.43	0.1089
119.7778	61.25	0.1627	144.1467	67.80	0.1098
88.4112	46.03	0.1699	101.1324	52.82	0.1097
30.1364	54.10	0.2599	42.1397	67.18	0.1491
21.6322	36.83	0.2518	33.2559	58.91	0.1552
33.3415	53.98	0.2672	35.2224	56.43	0.1434
42.7956	79.81	0.2510	28.4698	69.65	0.1499
38.2887	55.08	0.2310	31.0423	44.59	0.1447
306.7639	45.00	0.5047	377.6223	55.32	0.3928
236.6369	46.29	0.5255	420.6969	69.79	0.3484
184.7908	42.43	0.4538	278.3298	61.78	0.3247
288.1100	52.29	0.4688	288.2358	52.31	0.3196
251.8894	49.39	0.4581	291.0297	56.20	0.3177
30.3507	61.30	0.3078	44.3542	73.52	0.2112
48.3087	61.11	0.2892	56.1443	66.53	0.2055
33.7140	45.03	0.3210	68.3885	72.90	0.2036
45.5251	57.19	0.3000	43.1184	54.80	0.2051
45.9593	60.59	0.3148	61.5057	70.55	0.2067
383.2845	59.45	0.6426	385.7619	59.71	0.4679
512.8305	61.22	0.6364	549.4385	63.80	0.4829
338.3028	44.44	0.5993	503.8477	62.69	0.4729
396.4004	57.47	0.5925	489.5181	65.56	0.4661
415.1999	57.08	0.5771	340.1703	47.62	0.4793
43.9779	46.91	0.4038	61.9284	62.30	0.2884
61.2252	56.24	0.3799	70.7984	62.16	0.2776
70.6145	62.11	0.3647	89.3426	70.05	0.2764
55.5491	53.31	0.3959	55.4733	53.25	0.2879
55.0563	49.56	0.3819	63.6193	56.34	0.2846
309.1880	26.34	0.7970	545.3935	58.24	0.6392
578.8731	60.51	0.7819	480.4918	52.42	0.6457
353.4848	35.85	0.7983	393.9588	42.44	0.6292
413.1516	45.20	0.8025	470.8445	51.91	0.6317
517.1354	51.88	0.7914	580.7566	57.15	0.6524

8. Algoritmos Meméticos

Estos algoritmos surgen de la combinación de los algoritmos genéticos con la búsqueda local. En nuestro caso, utilizamos el algoritmo genético que mejor resultado dio para implementarla (AGE - uniforme). Como se puede observar, en los resultados medios de este problema, los resultados son mejores que con los genéticos en todos los casos.

Especialmente buenos son el algoritmo memético que aplica búsqueda local a toda la población cada 10 iteraciones (AM,1.0) y el que la aplica de forma aleatoria a un 10 % de los individuos que la forman (AM,0.1). Esto puede deberse a que la aleatoriedad y pérdida de buenas soluciones introducida por el algotitmo genético es contrarrestada por la búsqueda local, que analiza el entorno para buscar óptimos locales.

Los tiempos de ejecución también se reducen, ya que la búsqueda local es rápida.

Como desviación y tiempo de ejecución medio entre todos los casos encontramos lo siguiente para cada tipo de algoritmo memético:

AM - (10,1.0)	
Media Desv:	**42.73**
Media Tiempo:	**0.674**
AM - (10,0.1)	
Media Desv:	**42.10**
Media Tiempo:	**0.11**
AM - (10,0.1mej)	
Media Desv:	**47.61**
Media Tiempo:	**0.07**

También mostramos los resultados desglosados:

Caso	AM - (10,1.0)			AM - (10,0.1)			AM - (10,0.1mej)			
	Coste medio obtenido	Desv	Tiempo	Coste medio obtenido	Desv	Tiempo	Coste medio obtenido	Desv	Tiempo	Coste óptimo
GKD-b_1_n25_m2	0.0000	0.00	0.2502	0.00	0.00	0.04	0.0000	0.00	0.0296	0
GKD-b_2_n25_m2	0.0000	0.00	0.2490	0.00	0.00	0.05	0.0000	0.00	0.0308	0
GKD-b_3_n25_m2	0.0000	0.00	0.3279	0.00	0.00	0.04	0.0000	0.00	0.0298	0
GKD-b_4_n25_m2	0.0000	0.00	0.3396	0.00	0.00	0.04	0.0000	0.00	0.0281	0
GKD-b_5_n25_m2	0.0000	0.00	0.2877	0.00	0.00	0.05	0.0000	0.00	0.0283	0
GKD-b_6_n25_m7	12.7180	0.00	0.1548	26.12	51.31	0.18	26.2850	51.62	0.0259	13
GKD-b_7_n25_m7	14.0988	0.00	0.1566	24.10	41.49	0.17	16.7430	15.79	0.1240	14
GKD-b_8_n25_m7	16.7612	0.00	0.1460	32.34	48.17	0.17	29.1174	42.44	0.1170	17
GKD-b_9_n25_m7	25.0145	31.76	0.1265	28.50	40.11	0.16	25.0145	31.76	0.0264	17
GKD-b_10_n25_m7	26.7197	12.93	0.1255	26.72	12.93	0.15	30.2731	23.15	0.1270	23
GKD-b_11_n50_m5	5.3757	64.17	0.2352	10.31	81.31	0.21	9.5737	79.88	0.1516	2
GKD-b_12_n50_m5	6.3454	66.57	0.1985	11.66	81.81	0.20	10.9421	80.62	0.1712	2
GKD-b_13_n50_m5	6.0964	61.25	0.1751	6.50	63.64	0.21	4.5810	48.43	0.1603	2
GKD-b_14_n50_m5	6.0803	72.65	0.2063	4.87	65.85	0.19	8.8019	81.10	0.2072	2
GKD-b_15_n50_m5	7.1175	59.91	0.1670	7.39	61.37	0.19	8.3959	66.02	0.1502	3
GKD-b_16_n50_m15	56.1986	23.94	0.3039	61.97	31.02	0.05	86.0503	50.32	0.0431	43
GKD-b_17_n50_m15	57.4781	16.30	0.2791	69.42	30.70	0.06	74.7455	35.64	0.0472	48
GKD-b_18_n50_m15	59.0233	26.82	0.2872	57.69	25.12	0.06	57.3248	24.65	0.0515	43
GKD-b_19_n50_m15	62.7150	25.99	0.3544	49.24	5.75	0.07	67.5312	31.27	0.0423	46
GKD-b_20_n50_m15	47.7151	0.00	0.3512	53.56	10.92	0.07	79.8077	40.21	0.0428	48
GKD-b_21_n100_m10	24.7860	44.19	0.5436	25.65	46.08	0.08	32.2536	57.11	0.0418	14
GKD-b_22_n100_m10	25.5303	46.48	0.5287	27.51	50.33	0.07	37.2888	63.36	0.0549	14
GKD-b_23_n100_m10	29.9823	48.82	0.4944	20.62	25.58	0.07	38.4189	60.06	0.0547	15
GKD-b_24_n100_m10	27.2514	68.29	0.4765	31.79	72.82	0.06	36.2562	76.17	0.0407	9
GKD-b_25_n100_m10	29.7412	42.17	0.5334	19.17	10.25	0.06	41.9617	59.01	0.0431	17
GKD-b_26_n100_m30	342.0370	50.67	1.1550	265.41	36.43	0.11	248.1691	32.01	0.0606	169
GKD-b_27_n100_m30	348.3666	63.52	1.0812	269.76	52.88	0.09	248.8716	48.93	0.0720	127
GKD-b_28_n100_m30	327.7649	67.54	0.8751	198.05	46.29	0.10	222.3924	52.17	0.0637	106
GKD-b_29_n100_m30	243.0180	43.44	0.9472	207.65	33.80	0.09	284.6075	51.70	0.0625	137
GKD-b_30_n100_m30	252.2664	49.47	1.1646	221.50	42.45	0.10	210.6140	39.47	0.0744	127
GKD-b_31_n125_m12	43.2347	72.83	0.6471	24.60	52.26	0.06	32.2846	63.62	0.0487	12
GKD-b_32_n125_m12	32.0708	41.41	0.6402	39.72	52.69	0.09	40.4896	53.60	0.0477	19
GKD-b_33_n125_m12	31.2203	40.64	0.6609	41.68	55.54	0.07	44.9319	58.76	0.0544	19
GKD-b_34_n125_m12	40.5194	51.90	0.6205	47.32	58.81	0.08	35.9234	45.75	0.0516	19
GKD-b_35_n125_m12	44.7063	59.49	0.5096	36.81	50.80	0.09	35.4433	48.90	0.0500	18
GKD-b_36_n125_m37	357.1133	56.47	0.9900	335.92	53.73	0.15	429.7860	63.83	0.0749	155
GKD-b_37_n125_m37	445.7510	55.38	1.0288	391.35	49.18	0.15	420.9031	52.75	0.0923	199
GKD-b_38_n125_m37	553.4325	66.04	1.0654	293.06	35.86	0.12	539.8290	65.18	0.0680	188
GKD-b_39_n125_m37	428.8101	60.68	1.1277	326.69	48.39	0.15	376.8001	55.26	0.0847	169
GKD-b_40_n125_m37	415.0927	57.07	1.2665	419.79	57.55	0.14	498.9409	64.29	0.0767	178
GKD-b_41_n150_m15	50.3527	53.63	0.9410	53.62	56.46	0.10	65.3702	64.29	0.0503	23
GKD-b_42_n150_m15	57.0873	53.07	0.9388	64.39	58.39	0.08	61.2948	56.29	0.0547	27
GKD-b_43_n150_m15	64.1989	58.33	1.1850	58.92	54.59	0.10	82.6389	67.62	0.0615	27
GKD-b_44_n150_m15	63.9096	59.42	0.9940	49.83	47.95	0.08	58.8719	55.95	0.0517	26
GKD-b_45_n150_m15	55.5413	50.00	0.7309	54.57	49.11	0.07	56.1867	50.57	0.0552	28
GKD-b_46_n150_m45	703.9622	67.65	1.5968	467.65	51.30	0.14	414.3680	45.04	0.1047	228
GKD-b_47_n150_m45	603.4726	62.12	1.3057	457.16	49.99	0.12	485.9116	52.95	0.0876	229
GKD-b_48_n150_m45	502.0608	54.84	1.3344	394.88	42.58	0.14	413.1280	45.11	0.0889	227
GKD-b_49_n150_m45	655.5411	65.46	1.9691	506.29	55.28	0.12	614.9173	63.18	0.0928	226
GKD-b_50_n150_m45	675.2422	63.15	1.6356	565.86	56.02	0.13	700.9690	64.50	0.0916	249

9. Algoritmo Búsqueda Local

Como referencia, mantenemos los resultados medios de la búsqueda local:

Media Desv:	**55.62**
Media Tiempo:	**2.68E-02**

Los algoritmos genéticos y meméticos mejoran su resultado. Los segundos son especialmente destacables ya que incorporan las ventajas de los genéticos y de la búsqueda local, evitando que quedar atrapados en óptimos locales.

Los algoritmos genéticos tienen una desventaja, y es que según la versión que utilicemos la población puede verse demasiado modificada entre iteración e iteración. Además, la calidad del operador de cruce también afecta seriamente a la de los sucesores.

Algoritmo Búsqueda Local						
Caso	**Coste medio obtenido**	**Desv**	**Tiempo**	**Peor coste obtenido**	**Mejor coste obtenido**	**Coste óptimo**
GKD-b_1_n25_m2	0.0000	0.00	1.58E-02	0.0000	0.0000	0
GKD-b_2_n25_m2	0.0000	0.00	1.62E-02	0.0000	0.0000	0
GKD-b_3_n25_m2	0.0000	0.00	1.58E-02	0.0000	0.0000	0
GKD-b_4_n25_m2	0.0000	0.00	1.51E-02	0.0000	0.0000	0
GKD-b_5_n25_m2	0.0000	0.00	1.56E-02	0.0000	0.0000	0
GKD-b_6_n25_m7	26.4899	51.99	1.65E-02	15.2853	36.8636	13
GKD-b_7_n25_m7	29.5395	52.27	1.33E-02	24.0961	35.7991	14
GKD-b_8_n25_m7	34.3999	51.28	1.36E-02	24.9729	45.7449	17
GKD-b_9_n25_m7	39.7469	57.06	1.35E-02	25.5400	51.9409	17
GKD-b_10_n25_m7	35.6030	34.65	1.63E-02	30.2731	44.5938	23
GKD-b_11_n50_m5	16.0448	88.00	2.22E-02	12.7062	19.2730	2
GKD-b_12_n50_m5	14.0267	84.88	2.48E-02	7.0539	18.2810	2
GKD-b_13_n50_m5	16.1494	85.37	2.18E-02	13.2728	20.7124	2
GKD-b_14_n50_m5	11.0517	84.95	3.00E-02	7.9148	15.1537	2
GKD-b_15_n50_m5	16.2757	82.47	2.87E-02	12.1132	22.7177	3
GKD-b_16_n50_m15	121.9782	64.96	4.24E-02	87.0772	159.1846	43
GKD-b_17_n50_m15	109.6692	56.13	3.53E-02	83.2302	131.4499	48
GKD-b_18_n50_m15	68.8174	37.23	3.47E-02	57.3248	79.7628	43
GKD-b_19_n50_m15	143.7370	67.71	3.22E-02	92.7757	170.4828	46
GKD-b_20_n50_m15	96.4486	50.53	3.48E-02	85.1983	113.2540	48
GKD-b_21_n100_m10	46.0147	69.94	3.90E-02	30.2875	58.3638	14
GKD-b_22_n100_m10	42.0094	67.47	3.19E-02	34.5064	49.2292	14
GKD-b_23_n100_m10	36.0227	57.40	2.76E-02	25.7817	57.5326	15
GKD-b_24_n100_m10	37.7987	77.14	2.82E-02	27.5898	51.9625	9
GKD-b_25_n100_m10	44.9021	61.69	2.65E-02	27.6837	54.8968	17
GKD-b_26_n100_m30	387.6356	56.47	3.48E-02	280.4328	525.3839	169
GKD-b_27_n100_m30	376.6248	66.25	3.15E-02	204.7102	519.3854	127
GKD-b_28_n100_m30	379.9123	72.00	3.18E-02	292.3476	563.9683	106
GKD-b_29_n100_m30	344.0501	60.05	3.03E-02	261.6596	453.5187	137
GKD-b_30_n100_m30	304.0676	58.08	3.01E-02	225.7108	342.0634	127
GKD-b_31_n125_m12	48.7661	75.92	2.24E-02	27.6991	65.6209	12
GKD-b_32_n125_m12	49.8411	62.30	2.29E-02	45.3354	57.1193	19
GKD-b_33_n125_m12	68.7881	73.06	2.38E-02	50.3064	114.6981	19
GKD-b_34_n125_m12	51.6503	62.27	2.39E-02	34.1022	73.4153	19
GKD-b_35_n125_m12	61.8754	70.73	2.34E-02	42.8567	83.1226	18
GKD-b_36_n125_m37	403.2151	61.45	3.25E-02	347.9359	471.0077	155
GKD-b_37_n125_m37	418.0820	52.43	3.41E-02	284.9139	562.0131	199
GKD-b_38_n125_m37	446.9816	57.95	3.34E-02	310.1741	575.8495	188
GKD-b_39_n125_m37	411.6134	59.04	3.27E-02	310.7378	493.5993	169
GKD-b_40_n125_m37	386.2099	53.86	3.34E-02	315.3312	454.0314	178
GKD-b_41_n150_m15	69.7972	66.55	1.99E-02	51.7237	91.5401	23
GKD-b_42_n150_m15	65.2854	58.97	2.11E-02	46.2038	97.6911	27
GKD-b_43_n150_m15	65.6087	59.22	2.06E-02	50.8017	74.0788	27
GKD-b_44_n150_m15	61.1381	57.58	2.05E-02	50.7285	71.0724	26
GKD-b_45_n150_m15	56.5077	50.85	1.99E-02	49.2566	76.3017	28
GKD-b_46_n150_m45	575.4527	60.42	3.52E-02	477.4060	637.9589	228
GKD-b_47_n150_m45	399.5420	42.78	3.50E-02	310.0359	494.6895	229
GKD-b_48_n150_m45	520.6545	56.45	4.16E-02	366.7828	889.9092	227
GKD-b_49_n150_m45	457.3357	50.49	3.58E-02	358.8121	544.1437	226
GKD-b_50_n150_m45	528.6708	52.93	3.49E-02	408.7418	741.4361	249