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 cambio, si se utiliza una representación binaria, el pseudocódigo 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

4.1. Generador de soluciones aleatorias

Algorithm 3 Generador de soluciones aleatorias en representación binaria

```
procedure generateRandomSolution
    individual ← [ ]
    for i ∈ 1..m do
        individual = individual + [1]        ▷ Insertamos un 1 al final del vector
    end for
    for i ∈ 1..n − m do
        individual = individual + [0]
    end for
    shuffle(individual)
    return individual
end procedure
```

4.2. Mecanismo de selección del algoritmo genético generacional

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.

```
procedure GENERATIONALSELECTIONOPERATOR(popul, populDisp)
    parentsPopul ← [ ]
    parentsPopulDisp ← [ ]
    for i ∈ 1.,50 do
        firstRandElem ← rand(1, 50)
        secondRandElem ← rand(1, 50)          ▷ Debe de ser diferente a firstRandElem

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

        parentsPopul+ = {popul[bestRandElem]}
        parentsPopulDisp+ = {populDisp[bestRandElem]}
        return parentsPopul, parentsPopulDisp
    end for
end procedure
```

4.3. Mecanismo de selección del algoritmo genético estacionario

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 ← random(1, 50)
    secondRandElem ← random(1, 50)          ▷ Debe de ser diferente a firstRandElem

    parent1Index ← firstRandElem
    if populDisp[parent1Index] > populDisp[secondRandElem] then
        parent1Index ← secondRandElem
    end if

    firstRandElem ← random(1, 50)
    secondRandElem ← random(1, 50)          ▷ Debe de ser diferente a firstRandElem

    parent2Index ← firstRandElem
    if populDisp[parent2Index] > populDisp[secondRandElem] then
        parent2Index ← secondRandElem
    end if
    return parent1Index, parent2Index
end procedure
```

4.4. Operador de mutación

Algorithm 6 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)
end procedure
```

Parte III

Pseudocódigo de los algoritmos

5. Algoritmo Greedy

El primer algoritmo que implementamos para resolver el problema utiliza una estrategia Greedy, donde partiendo de una solución incompleta S con sólo dos vértices $v_1, v_2 \in V$ elegidos aleatoriamente, llegamos a una solución completa añadiendo un nuevo vértice en cada iteración. En concreto, se añade el vértice que minimiza la dispersión con respecto a los ya existentes.

Algorithm 7 Algoritmo Greedy

```
1: s_1, s_2 ← Rand(V)                ▷ Rand() devuelve dos vértices aleatorios de V
2: S ← {s_1, s_2}
3: U ← V \ {s_1, s_2}
4:
5: sum ← [ ]                         ▷ Array que contiene la distancia acumulada,
6: for u ∈ U do                      ▷ necesario para la factorización de la función objetivo
7:     for s ∈ S do
8:         sum[u] += d_{u,s}
9:     end for
10: end for
11: for s ∈ S do
12:     for s2 ∈ S do
13:         sum[s] += d_{s,s2}
14:     end for
15: end for
```

16: **while** $|S| < m$ **do** ▷ Donde m es el tamaño que debe tener la solución
17: $g_{min} \leftarrow \infty$
18: $u\_g_{min} \leftarrow -1$
19:
20: **for** $u \in U$ **do**
21: $\delta(v)_{max} \leftarrow -\infty$
22: $\delta(v)_{min} \leftarrow \infty$
23:
24: **for** $v \in S$ **do**
25: $\delta(v) \leftarrow sum[v] + d_{u,v}$
26: **if** $\delta(v)_{max} < \delta(v)$ **then**
27: $\delta(v)_{max} \leftarrow \delta(v)$
28: **end if**
29: **if** $\delta(v)_{min} > \delta(v)$ **then**
30: $\delta(v)_{min} \leftarrow \delta(v)$
31: **end if**
32:
33: $\delta(u)_{max} \leftarrow max(sum[u], \delta(v)_{max})$
34: $\delta(u)_{min} \leftarrow min(sum[u], \delta(v)_{min})$
35: $g = \delta(u)_{max} - \delta(u)_{min}$
36: **if** $g_{min} > g$ **then**
37: $g_{min} \leftarrow g$
38: $u\_g_{min} \leftarrow -1$
39: **end if**
40: **end for**
41:
42: $U \leftarrow U \setminus \{u\_g_{min}\}$
43: $S \leftarrow S + \{u\_g_{min}\}$
44: **end for**
45:
46: **for** $v \in V$ **do**
47: $sum[v]$ += $d_{v,u\_g_{min}}$
48: **end for**
49: **end while**
50:
51: **return** S

6. Algoritmo Búsqueda Local

En esta segunda implementación utilizamos una búsqueda local. Para ello, se genera una solución inicial aleatoria (y válida) y se va explorando su entorno. Cuando se encuentra un vecino que reduce la dispersión, se actualiza como nueva solución. Así se procede hasta haber recorrido todo el vecindario sin encontrar una solución mejor o hasta llegar a las 100000 evaluaciones de la función objetivo. Cada vez que se reinicia la exploración del vecindario, se barajan el vector que contiene la solución y el que contiene los vértices que no pertenecen a esta, para asegurar que el orden en el que se visitan los nodos no es determinístico.

Algorithm 8 Algoritmo Búsqueda Local primero el mejor

1: $U \leftarrow V$
2: $U \leftarrow$ Shuffle(U)
3: $S \leftarrow [\,]$
4: $sum \leftarrow [\,]$
5:
6: **for** $i = 0$ **to** $m - 1$ **do**
7: element $\leftarrow U.last$
8: $U \leftarrow U - \{element\}$
9: $S \leftarrow S + \{element\}$
10: **end for**
11:
12: $S_{best} \leftarrow S$
13: $current_cost \leftarrow$dispersion(S)
14: $best\_cost \leftarrow current\_cost$
15:
16: eval $\leftarrow 0$
17: $better\_solution \leftarrow true$
18:
19: **while** eval < 100000 **and** $better\_solution$ **do**
20: $better\_solution \leftarrow false$
21:
22: **for** $u \in S$ **and while** !$better\_solution$ **and** $eval < 100000$ **do**
23: **for** $v \in U$ **and while** !$better\_solution$ **and** $eval < 100000$ **do**
24: $eval \leftarrow eval + 1$
25: $\delta \leftarrow [\,]$ ▷ Array inicializado a 0
26: $\delta(w)_{max} \leftarrow -\infty$
27: $\delta(w)_{min} \leftarrow \infty$
28:

```
29:             for w ∈ S do
30:                 if w! = u then
31:                     δ[w] ← sum[w] − d_{w,u} + d_{w,v}
32:                     δ[v] += d_{w,v}
33:
34:                     if δ[w] > δ(w)_max then
35:                         δ(w)_max ← δ[w]
36:                     end if
37:                     if δ[w] < δ(w)_min then
38:                         δ(w)_min ← δ[w]
39:                     end if
40:                 end if
41:             end for
42:
43:             δ_max ← max(δ[v], δ(w)_max)
44:             δ_min ← min(δ[v], δ(w)_min)
45:             new_cost ← δ_max − δ_min
46:             if new_cost < current_cost then
47:                 best_cost ← new_cost
48:                 current_cost ← new_cost
49:
50:                 swap ← u                          ▷ intercambio u y v en S y U
51:                 u ← v
52:                 v ← swap
53:                 better_solution = true
54:                 best_solution = S
55:             end if
56:
57:         end for
58:     end for
59:
60:     shuffle(S)
61:     shuffle(U)
62: end while
63:
64: return S
```

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
│   └── practica1.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:

$./bin/practica1$ $<semilla>$ $<algoritmo>$ $<fichero\_datos>$

Donde $<algoritmo>$ puede tomar como valor g (Greedy) o b (Búsqueda Local).

Ejemplo: ./bin/practica1 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, y cada caso 5 veces para promediar los resultados de tiempo de ejecución y coste. La semilla se volverá a fijar al ejecutar cada caso, pero no entre ejecuciones sobre el mismo conjunto de datos.

7. Algoritmo Greedy

Con este algoritmo se observan unos tiempos de ejecución bajos, ya que la solución se construye progresivamente, y una vez elegido un elemento no se vuelve atrás. Los costes obtenidos no son demasiado cercanos a los óptimos y por tanto su desviación es bastante alta.

También se puede observar cómo el peor y mejor coste entre ejecuciones varía en gran medida. Esto se debe a que el algoritmo se encuentra influenciado por los dos elementos aleatorios elegidos al inicio de la ejecución.

Como desviación y tiempo de ejecución medio entre todos los casos encontramos:

Media Desv:	**66.71**
Media Tiempo:	**6.84E-05**

Lo cuál es un valor de desviación bastante elevado. Como ventajas de este algoritmo, podemos destacar su bajo tiempo de ejecución. Aún así, el criterio heurístico que utiliza para elegir los candidatos no es bastante bueno en este problema.

Algoritmo Greedy						
Caso	**Coste medio obtenido**	**Desv**	**Tiempo**	**Peor coste obtenido**	**Mejor coste obtenido**	**Coste óptimo**
GKD-b_1_n25_m2	0.0000	0.00	7.00E-06	0.0000	0.0000	0
GKD-b_2_n25_m2	0.0000	0.00	6.00E-06	0.0000	0.0000	0
GKD-b_3_n25_m2	0.0000	0.00	4.00E-06	0.0000	0.0000	0
GKD-b_4_n25_m2	0.0000	0.00	7.00E-06	0.0000	0.0000	0
GKD-b_5_n25_m2	0.0000	0.00	6.00E-06	0.0000	0.0000	0
GKD-b_6_n25_m7	66.8184	80.97	1.00E-05	42.8698	91.3110	13
GKD-b_7_n25_m7	61.5395	77.09	1.00E-05	29.2765	106.8920	14
GKD-b_8_n25_m7	47.7101	64.87	1.10E-05	24.2035	61.4048	17
GKD-b_9_n25_m7	56.4502	69.76	1.00E-05	28.6604	78.9821	17
GKD-b_10_n25_m7	72.0830	67.72	1.10E-05	44.5938	121.6421	23
GKD-b_11_n50_m5	22.4261	91.41	1.40E-05	10.7082	27.4908	2
GKD-b_12_n50_m5	35.0795	93.95	1.00E-05	16.4399	46.1466	2
GKD-b_13_n50_m5	25.4011	90.70	8.00E-06	7.3063	32.7557	2
GKD-b_14_n50_m5	27.2301	93.89	7.00E-06	12.8679	53.1448	2
GKD-b_15_n50_m5	34.9795	91.84	8.00E-06	13.4118	60.1254	3
GKD-b_16_n50_m15	186.2275	77.05	1.60E-05	139.2353	246.0371	43
GKD-b_17_n50_m15	168.9353	71.52	1.60E-05	148.0619	195.2262	48
GKD-b_18_n50_m15	133.2367	67.58	1.50E-05	93.0530	196.0516	43
GKD-b_19_n50_m15	154.6660	69.99	1.50E-05	117.3899	188.4828	46
GKD-b_20_n50_m15	195.3694	75.58	1.50E-05	130.9698	277.4382	48
GKD-b_21_n100_m10	60.9593	77.31	2.10E-05	45.7342	81.9708	14
GKD-b_22_n100_m10	70.8286	80.71	3.80E-05	47.9586	106.2160	14
GKD-b_23_n100_m10	64.6286	76.26	3.60E-05	42.1474	88.5631	15
GKD-b_24_n100_m10	53.7252	83.92	2.30E-05	44.4337	76.3345	9
GKD-b_25_n100_m10	66.5061	74.14	2.20E-05	49.4304	98.4419	17
GKD-b_26_n100_m30	405.3022	58.37	8.00E-05	298.4035	571.4217	169
GKD-b_27_n100_m30	538.8505	76.41	1.16E-04	367.8878	719.2656	127
GKD-b_28_n100_m30	434.2580	75.50	1.18E-04	288.0312	641.6096	106
GKD-b_29_n100_m30	372.3980	63.09	1.00E-04	331.6536	406.6306	137
GKD-b_30_n100_m30	398.3456	68.00	8.90E-05	298.7756	546.2969	127
GKD-b_31_n125_m12	90.9331	87.08	5.00E-05	61.7402	120.4408	12
GKD-b_32_n125_m12	115.6933	83.76	4.30E-05	72.9959	186.8720	19
GKD-b_33_n125_m12	94.5057	80.39	5.60E-05	69.9862	143.3528	19
GKD-b_34_n125_m12	110.8166	82.41	5.70E-05	64.7737	209.1679	19
GKD-b_35_n125_m12	77.6650	76.68	4.40E-05	56.3537	109.9123	18
GKD-b_36_n125_m37	571.5901	72.81	1.41E-04	431.7039	784.9058	155
GKD-b_37_n125_m37	630.5293	68.46	1.66E-04	403.5947	853.7735	199
GKD-b_38_n125_m37	492.7449	61.85	1.54E-04	350.9502	578.2679	188
GKD-b_39_n125_m37	407.9256	58.67	1.41E-04	355.7441	487.7512	169
GKD-b_40_n125_m37	454.0109	60.75	1.78E-04	408.5156	479.8117	178
GKD-b_41_n150_m15	103.7056	77.49	5.10E-05	73.8288	175.8578	23
GKD-b_42_n150_m15	141.9278	81.12	5.00E-05	135.4995	148.8573	27
GKD-b_43_n150_m15	139.2934	80.79	4.90E-05	106.5678	174.6647	27
GKD-b_44_n150_m15	140.8560	81.59	4.70E-05	77.1125	223.6438	26
GKD-b_45_n150_m15	125.5488	77.88	5.10E-05	80.6669	173.7174	28
GKD-b_46_n150_m45	650.1872	64.97	2.72E-04	416.0328	962.6625	228
GKD-b_47_n150_m45	512.8350	55.42	2.70E-04	431.9668	668.6509	229
GKD-b_48_n150_m45	632.2425	64.14	2.42E-04	503.0323	766.2350	227
GKD-b_49_n150_m45	426.1721	46.87	2.55E-04	378.0599	478.7449	226
GKD-b_50_n150_m45	551.5460	54.88	2.54E-04	388.4676	705.1301	249

8. Algoritmo Búsqueda Local

Utilizando esta técnica, los tiempos de ejecución aumentan con respecto a Greedy, aunque se siguen manteniendo en valores bajos. Los costes mejoran, aunque no en demasiada medida.

Ocurre la misma situación que en el otro algoritmo: los peores y mejores costes presentan gran variabilidad. En este caso, el no determinismo es incluso mayor que en Greedy, ya que se realiza un shuffle() cada vez que se encuentra una solución que mejora la actual y no se recorre todo el vecindario (ya que nos encontramos ante una búsqueda primero el mejor).

Como desviación y tiempo de ejecución medio entre todos los casos encontramos:

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

Ninguno de los dos algoritmos que hemos estudiado se acerca demasiado a la solución óptima, ya que sus criterios heurísticos no son demasiado fuertes. En el caso de Greedy, una vez elegido un elemento no se puede volver atrás, y en el caso de la búsqueda local, sólo tenemos en cuenta los posibles cambios con el vecindario y no el resto.

Aún así, el hecho de poder utilizar factorización hace que los algoritmos presenten una baja complejidad y tiempos de ejecución, que con técnicas más avanzadas no se conseguirían.

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