Metaheurísticas (Curso 2021-2022)

Grado en Ingeniería Informática
Universidad de Granada

UNIVERSIDAD DE GRANADA

Práctica 1: Técnicas de Búsqueda Local y Algoritmos Greedy

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 primera práctica de la asignatura implementamos y comparamos el rendimiento de dos familias de algoritmos, **Greedy** y **Búsqueda Local primero el mejor**. 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, 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. S es una solución válida del problema si:

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

Y por tanto, $m < n$.

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 pseudocódigo quedaría de la siguiente forma:

Algorithm 1 Función objetivo

```
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 los algoritmos que aparecen en esta práctica, no se utiliza directamente esta implementación de la función objetivo (excepto para inicializar la Búsqueda Local). Esto se debe a que es costosa, en concreto tiene una complejidad computacional de $O(n^2)$. Utilizamos versiones factorizadas de la función, que reutilizan cálculos previos de iteraciones anteriores para actualizar el valor de la dispersión. De esta forma, obtenemos una complejidad de $O(n)$.

Parte III

Pseudocódigo de los algoritmos

4. 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 2 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 ← ∞
18:     u_g_min ← −1
19:
20:     for u ∈ U do
21:         δ(v)_max ← −∞
22:         δ(v)_min ← ∞
23:
24:         for v ∈ S do
25:             δ(v) ← sum[v] + d_{u,v}
26:             if δ(v)_max < δ(v) then
27:                 δ(v)_max ← δ(v)
28:             end if
29:             if δ(v)_min > δ(v) then
30:                 δ(v)_min ← δ(v)
31:             end if
32:
33:             δ(u)_max ← max(sum[u], δ(v)_max)
34:             δ(u)_min ← min(sum[u], δ(v)_min)
35:             g = δ(u)_max − δ(u)_min
36:             if g_min > g then
37:                 g_min ← g
38:                 u_g_min ← −1
39:             end if
40:         end for
41:
42:         U ← U \ {u_g_min}
43:         S ← S + {u_g_min}
44:     end for
45:
46:     for v ∈ V do
47:         sum[v] += d_{v,u_g_min}
48:     end for
49: end while
50:
51: return S
```

5. 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 3 Algoritmo Búsqueda Local primero el mejor

```
1:  U ← V
2:  U ← Shuffle(U)
3:  S ← [ ]
4:  sum ← [ ]
5:
6:  for i = 0 to m − 1 do
7:      element ← U.last
8:      U ← U − {element}
9:      S ← S + {element}
10: end for
11:
12: S_best ← S
13: current_cost ←dispersion(S)
14: best_cost ← current_cost
15:
16: eval ← 0
17: better_solution ← true
18:
19: while eval < 100000 and better_solution do
20:     better_solution ← false
21:
22:     for u ∈ S and while !better_solution and eval < 100000 do
23:         for v ∈ U and while !better_solution and eval < 100000 do
24:             eval ← eval + 1
25:             δ ← [ ]                                  ▷ Array inicializado a 0
26:             δ(w)_max ← −∞
27:             δ(w)_min ← ∞
28:
```

```
            for w ∈ S do
                if w! = u then
                    δ[w] ← sum[w] − d_{w,u} + d_{w,v}
                    δ[v] += d_{w,v}

                    if δ[w] > δ(w)_max then
                        δ(w)_max ← δ[w]
                    end if
                    if δ[w] < δ(w)_min then
                        δ(w)_min ← δ[w]
                    end if
                end if
            end for

            δ_max ← max(δ[v], δ(w)_max)
            δ_min ← min(δ[v], δ(w)_min)
            new_cost ← δ_max − δ_min
            if new_cost < current_cost then
                best_cost ← new_cost
                current_cost ← new_cost

                swap ← u                          ▷ intercambio u y v en S y U
                u ← v
                v ← swap
                better_solution = true
                best_solution = S
            end if

        end for
    end for

    shuffle(S)
    shuffle(U)
end while

return S
```

Parte IV

Pseudocódigo de los algoritmos

Referencias

[1] https://arstech.net/phoronix-test-suite/, consultado el 19 de diciembre de 2021.

[2] https://stackoverflow.com/questions/68936024/how-can-i-install-the-file-libc-a-using-dnf-install-in-centos8, consultado el 19 de diciembre de 2021.

[3] https://ubunlog.com/phoronix-test-suite-una-herramienta-para-benchmark-multiplata consultado el 20 de diciembre de 2021.

[4] https://github.com/davidPalomar-ugr/iseP4JMeter, consultado el 21 de diciembre de 2021.

[5] https://jmeter.apache.org/download_jmeter.cgi, consultado el 21 de diciembre de 2021.

[6] https://jmeter.apache.org/usermanual/index.html, consultado el 22 de diciembre de 2021.

[7] https://docs.docker.com/desktop/mac/install/, consultado el 23 de diciembre de 2021.

[8] https://www.phoronix.com/scan.php?page=article&item=docker-phoronix-pts&num=1, consultado el 23 de diciembre de 2021.