Metaheurísticas: Práctica 1
Búsqueda Local y Algoritmos Greedy para el Problema de la Máxima Diversidad

David Cabezas Berrido

20079906D

Grupo 2: Viernes

dxabezas@correo.ugr.es

23 de marzo de 2021

Índice

1. Descripción y formulación del problema

Nos enfrentamos al **Problema de la Máxima Diversidad** (**Maximum Diversity Problem, MDP**). El problema consiste en seleccionar un subconjunto m elementos de un conjunto de $n > m$ elementos de forma que se **maximice** la *diversidad* entre los elementos escogidos.

Disponemos de una matriz $D = (d_{ij})$ de dimensión $n \times n$ que contiene las distancias entre los elementos, la entrada (i, j) contiene el valor d_{ij}, que corresponde a la distancia entre el elemento i-ésimo y el j-ésimo. Obviamente, la matriz D es simétrica y con diagonal nula.

Existen distintas formas de medir la diversidad, que originan distintas variantes del problema. En nuestro caso, la diversidad será la suma de las distancias entre cada par de elementos seleccionados.

De manera formal, se puede formular el problema de la siguiente forma:

Maximizar

$$f(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j \tag{1}$$

sujeto a

$$\sum_{i=1}^{n} x_i = m$$
$$x_i = \{0, 1\}, \quad \forall i = 1, \ldots, n.$$

Una solución al problema es un vector binario x que indica qué elementos son seleccionados, seleccionamos el elemento i-ésimo si $x_i = 1$.

Sin embargo, esta formulación es poco eficiente y para la mayoría de algoritmos proporcionaremos otra equivalente pero más eficiente.

El problema es **NP-completo** y el tamaño del espacio de soluciones es $\binom{n}{m}$, de modo que es conveniente recurrir al uso de metaheurísticas para atacarlo.

2. Aplicación de los algoritmos

Los algoritmos para resolver este problema tendrán como entradas la matriz D $(n \times n)$ y el valor m. La salida será un contenedor (vector, conjunto, ...) con los índices de los elementos seleccionados, y no un vector binario como el que utilizamos para la formulación. En nuestro caso utilizaremos conjuntos para representar soluciones.

Para representar las soluciones, usaremos conjuntos de enteros con los elementos seleccionados. La evaluación de la calidad de una solución se hará sumando la contribución de cada uno de los elementos, y dividiremos la evaluaciónen dos funciones. En lugar de calcular la función evaluación como en (1), lo haremos así:

$$f(x) = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} d(i,j) = \frac{1}{2}\sum_{i=1}^{m} \text{contrib}(i) \tag{2}$$

La diferencia es que contamos la distancia entre cada dos elementos i, j dos veces, distancia del elemento i-ésimo al j-ésimo y del j-ésimo al i-ésimo. Esto es obviamente más lento que con $j > i$ en la sumatoria, pero nos permite factorizar la evaluación de la solución como suma de las contribuciones de los elementos, lo cuál será útil para reaprovechar cálculos al evaluar soluciones para la Búsqueda Local. Además, representar la solución como un vector de m índices y no un vector binario de longitud n presenta una clara ventaja: las sumatorias van hasta m en lugar de n. No tenemos que computar distancias para luego multiplicarlas por cero como sugería la formulación en (1).

Presentamos el pseudocódigo de la función para calcular la contribución de un elemento x_i.

Algorithm 1: CONTRIB calcula la contribución de un elemento en una solución.

Input: Un conjunto de índices S.
Input: La matriz de distancias D.
Input: Un entero e correspondiente al índice del elemento.
Output: La contribución del elemento e, como se describe en (2).

```
contrib ← 0
for s in S do
    contrib ← contrib + D[e, s]        // Sumo las distancias del elemento e a cada elemento de S
return contrib
```

Nótese que el elemento e no tiene que pertenecer al conjunto S. Esto obviamente no ocurrirá cuando se vaya a evaluar una solución al completo invocando esta función con la que describiremos a continuación. Pero, de esta forma, permite conocer cómo influirá en la evaluación el añadir un nuevo elemento sin necesidad de añadirlo realmente.

Ahora presentamos el pseudocódigo de la función para evaluar una solución completa.

Algorithm 2: FITNESS calcula la evaluación de una solución.

Input: Un conjunto de índices S.
Input: La matriz de distancias D.
Output: El valor de la función objetivo sobre la solución compuesta por S, como se describe en (2).

```
fitness ← 0
for e in S do
    fitness ← fitness + contrib(S, D, e)    // Sumo la contribución de cada elemento de la solución
return fitness/2                            // Hemos contado cada distancia dos veces
```

Podemos definir la distancia de un elemento e a un conjunto S como:

$$d(e,S) = \sum_{s \in S} d(e,s) \tag{3}$$

Esta expresión nos será de utilidad para la implementación de los algoritmos.

Gracias a la existencia del Algoritmo 1, podemos obtener esta expresión como $\text{contrib}(S, D, e)$.

3. Descripción de los algoritmos

3.1. Búsqueda local

Procedemos con la descripción del algoritmo de Búsqueda Local que se nos ha presentado en el seminario. Este algoritmo utiliza la técnica del Primer Mejor, en la que se van generando soluciones en el entorno de la actual y se salta a la primera con mejor evaluación. Para la implementación del algoritmo, necesitamos distintos elementos.

El primer elemento, es una función para generar una solución aleatoria de partida. Simplemente se eligen m elementos diferentes del conjunto. Por comodidad, también calculamos el complementario.

Algorithm 3: RandomSol proporciona una solución válida aleatoria

Input: El entero m.
Input: El entero n.
Output: Una solución válida del MDP obtenida aleatoriamente.
Output: El complementario de la solución obtenida.
$E \leftarrow \{0, \ldots, n-1\}$ `// Conjunto con los elementos no seleccionados`
$S \leftarrow \emptyset$ `// La solución empieza vacía`
while $|S| < m$ **do**
  $e \leftarrow$ elemento aleatorio de E
  $E \leftarrow E \backslash \{e\}$
  $S \leftarrow S \cup \{e\}$
return S
return E `// El complementario`

Lo siguiente que necesitamos es un método para generar las soluciones del entorno. Estas soluciones se consiguen sustituyendo el menor contribuyente de la solución actual por otro candidato. Presentamos el código para obtener el menor contribuyente.

Algorithm 4: lowestContrib obtiene el elemento de S que menos contribuye en la valoración.

Input: Un conjunto de elementos S.
Input: La matriz de distancias D.
Output: El elemento de S que minimiza contrib(S, S, e) con $e \in S$.
Output: Su contribución, para la factorización de la función objetivo.
lowest $\leftarrow$ primer elemento de S
min_contrib $\leftarrow$ contrib$(S, D, lowest)$
for s ***in*** S **do**
  contrib $\leftarrow$ contrib(S, D, s)
  if *contrib* $<$ *min_contrib* **then**
    min_contrib $\leftarrow$ *contrib*
    lowest $\leftarrow s$ `// Si encuentro un candidato con menor contribución, actualizo`
return *lowest*
return *min_contrib*

En el caso de que S se represente como un conjunto, no sabemos cuál será el primer elemento (depende de la implementación del iterador). Pero esto no es relevante, ya que vale cualquier elemento de S.

Finalmente, proporcionamos el algoritmo de Búsqueda Local para actualizar la solución por otra del entorno iterativamente hasta encontrar un máximo local (una solución mejor que todas las de su entorno) ollegar a un límite de evaluaciones de la función objetivo: $LIMIT = 100000$. Las soluciones del entorno se generan aleatoriamente.

Algorithm 5: LocalSearch

Input: El entero m.
Input: La matriz de distancias D, $n \times n$.
Output: Una solución válida del MDP por el algoritmo de BS que hemos descrito, junto con su evaluación.

```
S ← randomSol(m, n)                          // Comenzamos con una solución aleatoria
E ← {0, ..., n − 1}\S                        // randomSol también devuelve el complementario
fitness ← fitness(S)                         // Diversidad de la solución
E ← vector(E)                                // No importa el orden, pero debe poder barajarse
carryon ← true
LIMIT ← 100000                               // Límite de llamadas a la función de evaluación
CALLS ← 0
while carryon do
    lowest = lowestContributor(S, D)
    min_contrib ← contrib(S, D, lowest)      // Se calcula dentro de lowestContributor
    S ← S\{lowest}
    E ← shuffle(E)
    for e in E do
        contrib ← contrib(S, D, e)
        CALLS ← CALLS + 1                    // He evaludado una posible solución
        if contrib > min_contrib then
            fitness ← fitness + contrib − min_contrib   // Diversidad de la nueva solución
            carryon ← true                   // Toca saltar, lo que completa la iteración
            S ← S ∪ {e}                      // Saltamos a la nueva solución
            E ← E\{e}
            E ← E ∪ {lowest}
        if carryon == true or CALLS ≥ LIMIT then
            break                            // Se cumple alguna de las condiciones de parada
if |S| < m then
    S ← S ∪ {lowest}     // Si salimos porque no encontramos una mejor, recuperamos la solución
return S
return fitness
```

Cabe destacar que a diferencia del algoritmo Greedy (Algoritmo 7) en el que se evalúa la solución al final, en este algoritmo se calcula factorizando. Esto acelera mucho los cálculos, ya que hay que evaluar muchas soluciones diferentes.

4. Algoritmo de comparación: Greedy

Para comparar la eficacia de cada algoritmos, lo compararemos con el algoritmo **Greedy** descrito en el seminario.

El algoritmo consiste en empezar por el elemento más lejano al resto e ir añadiendo el elemento que más contribuya hasta completar una solución válida.

Como elemento más lejano al resto se toma el elemento cuya suma de las distancias al resto sea la mayor. Y en cada iteración se introduce el elemento cuya suma de las distancias a los seleccionados sea mayor. Es decir, utilizamos la definición de (3).

Para calcular ambos valores, usamos la siguiente función, que permite obtener el de entre un conjunto de candidatos más lejano (en el sentido que acabamos de comentar) a los elementos de un conjunto dado. El código para calcularlo es similar al del algoritmo 4.

Algorithm 6: FARTHEST obtiene el candidato más lejano a los elementos de S.

Input: Un conjunto de candidatos C.
Input: Un conjunto de elementos S.
Input: La matriz de distancias D.
Output: El candidato más lejano en el sentido de (3).

```
farthest ← primer elemento de C
max_contrib ← contrib(S, D, farthest)
for e in C do
    contrib ← contrib(S, D, e)
    if contrib > max_contrib then
        max_contrib ← contrib
        farthest ← e                    // Si encuentro un candidato con mayor contribución, actualizo
return farthest
```

En el caso de que C se represente como un conjunto, no sabemos cuál será el primer elemento (depende de la implementación del iterador). Pero esto no es relevante, ya que vale cualquier elemento de C.

Ya estamos en condiciones de proporcionar una descripción del algoritmo Greedy.

Algorithm 7: GREEDY

Input: La matriz de distancias D.
Input: El entero m.
Output: Una solución válida del MDP obtenida como hemos descrito anteriormente, y su diversidad.

```
C ← {0, ..., n − 1}                          // En principio los n elementos son candidatos
S ← ∅                                                    // La solución empieza vacía
farthest ← farthest(C, C, D)                          // Elemento más lejano al resto
C ← C\{farthest}
S ← S ∪ {farthest}
while |S| < m do
    farthest ← farthest(C, S, D)                  // Elemento más lejano a los seleccionados
    C ← C\{farthest}
    S ← S ∪ {farthest}
return S
return fitness(S)
```

5. Desarrollo de la práctica

La implementación de los algoritmos y la experimentación con los mismos se ha llevado acabo de C++, utilizando la librería STL. Para representar la soluciones hemos hecho uso del tipo `unordered_set`, ya que se realizan pocas operaciones de consulta y muchas de inserción y borrado.

Para medir los tiempos de ejecución se utiliza la función `clock` de la librería `time.h`.

A lo largo de la práctica se utilizan acciones aleatorias. Utilizamos la librería `stdlib.h` para la generación de números pseudoaleatorios con `rand` y fijamos la semilla con `srand`. También se baraja el vector de candidatos en la búsqueda local con la función `random_shuffle` de la librería `algorithm`.

Se almacena la matriz de distancias completa (no sólo un triángulo) por comodidad de los cálculos.

TODO ¿pequeño manual de usuario? ¿puede ser el LEEME?

6. Experimentación y análisis

Toda la experimentación se realiza en mi ordenador portátil personal, que tiene las siguientes especificaciones:

- OS: Ubuntu 20.04.2 LTS x86_64.
- RAM: 8GB, DDR4.
- CPU: Intel Core i7-6700HQ, 2.60Hz.

7. Trabajo voluntario

7.1. Comparación Búsqueda Local: Mejor vs Primer Mejor