Problema de Máxima Diversidad (MDP)

Técnicas de búsqueda local y algoritmos greedy

Metaheurísticas: Práctica 1, Grupo 1

José Antonio Álvarez Ocete - 77553417Q
joseantonioao@correo.ugr.es

30 de marzo de 2019

Índice

1. El problema

1.1. Descripción del problema

El **problema de la máxima diversidad** (en inglés, *maximum diversity problem*, MDP) es un problema de optimización combinatoria que consiste en seleccionar un subconjunto M de m elementos ($|M| = m$) de un conjunto inicial N de n elementos (con $n > m$) de forma que se maximice la diversidad entre los elementos escogidos. El MDP se puede formular como:

$$\text{Maximizar } z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} \cdot x_i \cdot x_j$$

$$\text{Sujeto a } \sum_{i=1}^{n} x_i = m$$

$$x_i \in \{0,1\}, \forall i \in \{1,\ldots,n\}$$

Donde:

- x es una solución al problema que consiste en un vector binario que indica los m elementos seleccionados.
- d_{ij} es la distancia existente entre los elementos i y j.

1.2. Casos considerados

Se utilizarán 30 casos seleccionados de varios de los conjuntos de instancias disponibles en la *MDPLIB* (`http://www.optsicom.es/mdp/`), 10 pertenecientes al grupo **GKD** con distancias Euclideas, $n = 500$ y $m = 50$ ($GKD - c\_i\_n500\_m50$ para $i \in \{1,\ldots,10\}$), 10 del grupo **SOM** con distancias enteras entre 0 y 999, $n \in \{300, 400, 500\}$ y $m =\in \{40,\ldots,200\}$ ($SOM-b\_11\_n300\_m90$ a $SOM-b\_20\_n500\_m200$) y 10 del grupo **MDG** con distancias enteras entre 0 y 10, $n = 2000$ y $m = 200$ ($MDG - a\_i\_n2000\_m200$ para $i \in \{21,\ldots,30\}$.

Puesto que la numeración utilizada es unívoca se hará referencia a estas entradas simplemente como **MDP-a_i** con $i \in \{1,\ldots,10\}$, **SOM-b_i** con $i \in \{11,\ldots,20\}$ y **GKD-c_i** con $i \in \{21,\ldots,30\}$.

2. Descripción de la aplicación de los algoritmos

En esta sección describiremos las consideraciones comunes a los distintos algoritmos. Este incluye la representación de las soluciones, la función objetivo y los operadores comunes a los distintos algoritmos. Ya que los únicos puntos en común de la búsqueda local y la técnica greedy son la función objetivo y la representación de las soluciones no estudiaremos ningún operador común. Tampoco se han incluido los detalles específicos de un algoritmo a pesar de que sean comunes a varios algoritmos finales, ya que estos son pequeñas variaciones unos y otros.

El lenguaje utilizado para la implementación de la práctica ha sido $C++$.

2.1. Representación de la soluciones

El esquema de representación de una solución es el siguiente:

```
struct solution {
        vector<int> v;
        double fitness;
};
```

Donde el vector contiene números entre 1 y n no repetidos que componen la solución ($|v| = m$). Aunque el orden de estos elementos no es relevante se utilizará determinado ordenamiento sobre este mismo vector en algunas de las soluciones planteadas.

Cabe destacar que los datos proporcionados únicamente representan las distancias punto a punto, para todos los puntos. Sin embargo se desconoce la posición exacta de cada elemento. Es por esto por lo que no se ha podido implementar la técnica Greedy planteada inicialmente ya que se centraba en el concepto de *centroide* o *baricentro* de un conjunto y no podíamos calcularlo sin estimar primero la posición de los puntos.

Estos datos se han almacenado en una matriz simétrica de tamaño n x n que de aquí en adelante denotaremos por MAT.

2.2. Función objetivo

Para la función objetivo se ha dividido la implementación en dos funciones ya que algunos algoritmos utilizarán unicamente una de las dos y otros, ambas. La primera calcula la contribución del elemento i a la solución actual. La segunda calcula el $fitness$ total utilizando la función anterior.

Algorithm 1: singleContribution

```
Data: solution : sol , i : int
Result: contribution : double
begin
    contribution ← 0
    foreach j ∈ sol.v do
        contribution ← contribution + MAT[ i ][ j ]
    end
end
```

Algorithm 2: evaluateSolution

```
Data: solution : sol
Result: fitness : double
begin
    fitness ← 0
    foreach i ∈ sol.v do
        fitness ← fitness + SingleContribution ( sol, i )
    end
    fitness ← fitness / 2
end
```

3. Algoritmos

Para esta práctica se han implementado un total de 4 algoritmos que a continuación describiremos en profundidad. Son los siguientes:

- Greedy
- Búsqueda local

- Búsqueda local determinista
- Búsqueda local con greedy

3.1. Greedy

Como ya se ha comentado, la concepción inicial de la práctica era utilizar la idea del *baricentro*. Para ello en cada iteración se calcularía el baricentro de los elementos de la solución y se escogería el punto más alejado a este entre los aún no escogidos. Intentando simular esta estrategia utilizaremos la suma de las distancias al resto de elementos de la solución para cada punto aún no he escogido.

En primer lugar se ha implementado una función que, dados dos conjuntos de puntos, *seleccionados* y *no_seleccionados*, devuelve el elemento de *no_seleccionados* cuya suma de distancias a los elementos de *seleccionados* es máxima. Para ello utilizamos la función *SingleContribution* explicado en el apartado anterior, que computa la suma de distancias de un punto a otro conjunto dado.

Algorithm 3: farthestToSel

```
Data: selected : set < int > , non_selected : set < int >
Result: farthest : i
begin
    fitness ← 0
    max_sum_dist ← 0
    foreach i ∈ non_selected do
        current_sum_dist ← SingleContribution ( selected , i )
        if current_sum_dist > max_sum_dist then
            max_sum_dist ← current_sum_dist
            farthest ← i
        end
    end
end
```

De cara al algoritmo greedy final necesitaremos inicializar el conjunto de elementos seleccionados con al menos un elemento. Este será el que esté más alejado de todos los demás. Para calcularlo utilizamos una abstracción de la función anterior, donde *selected* y *non_selected* serán ambos el conjunto de todos los puntos posibles: $\{0, 1, \ldots, n\}$.

Algorithm 4: farthestToAll

```
Data: none
Result: farthest : i
begin
    all_elements ← {0, 1, ..., n}
    farthest ← farthestToSel( all_elements, all_elements )
end
```

Finalmente presentamos el algoritmo greedy al completo, haciendo uso de las funciones anteriores.

Algorithm 5: greedy

```
Data: none
Result: selected : set < int >
begin
    non_selected ← {0, 1, ..., n}
    selected ← { farthestToAll() }
    while |selected| < m do
        farthest ← farthestToSel(selected, non_selected)
        non_selected ← non_selected ⋃ {farthest}
        selected ← selected - {farthest}
    end
end
```

3.2. Búsqueda local

Desgranemos la búsqueda local paso por paso. En primer lugar generamos una solución aleatoria que será el punto de partida. En cada iteración exploramos el vecindario hasta encontrar una solución mejor y sustituimos la actual por la encontrada. Repetimos este proceso hasta que la exploración del vecindario no encuentre una solución mejor.

Algorithm 6: localSearch

```
Data: none
Result: sol : solution
begin
    sol ← randomSolution()
    stop ← false
    while !stop do
        stop , sol ← stepInNeighbourhood(sol)
    end
end
```

La exploración del vecindario realizada en la función $stepInNeighbourhood$ tiene dos detalles relevantes a explicar. Por un lado se ha explicado la factorización del movimiento en el vecindario. Esto es, para estudiar si una solución es mejor en vez de calcular la fitness de la nueva solución y compararla con la actual, estudiamos si el intercambio del elemento $i \in sol$ y el elemento j de los no seleccionados tiene una repercusión positiva en la fitness de la solución. Para ello hacemos uso de la función $SingleContribution$ comparando las contribuciones de cada elemento. Si el intercambio merece la pena ($SingleContribution(i, sol) < SingleContribution(j, sol)$) reajustamos la fitness sumandole la diferencia entre ambas.

Por otro lado, a la hora de escoger que elementos i, j comparar, seleccionamos un elemento j que aún no esté en la solución de forma aleatoria y comparamos con todos los posibles $i \in sol$. Una pequeña mejora consiste en ordenar los elementos de la solución en función a la contribución que realizan a esta para intentar intercambiar primero los que menos contribuyan. Veamos como está implementada esta ordenación. Por un lado definimos un operador de comparación que trabaje con parejas $< int, double >$. Esto nos permitirá ordenar el vector solución en orden de contribución creciente.

A continuación reordenamos los elementos de la solución.

Algorithm 7: operator<

Data: p1 : $pair < int, double >$, p2 : $pair < int, double >$
Result: comp : bool
begin
  comp $\leftarrow p1.second < p2.second$
end

Algorithm 8: orderSolutionByContribution

Data: sol : solution
Result: sol : solution
begin
  pairs : $vector < pair < int, double >>$
  foreach $i \in sol.v$ **do**
    pairs[i].first $\leftarrow$ i
    pairs[i].second $\leftarrow$ singleContribution(sol.v, i);
  end
  pairs $\leftarrow$ sort(pairs, *operator* <)
  foreach $i \in sol.v$ **do**
    sol.v[i] $\leftarrow$ pairs[i].first
  end
end

Para acabar presentamos la exploración del vecindario, donde $rand(a, b)$ devuelve un número entero aleatorio en $[a, b)$. Se ordena el vector solución utilizando $orderSolutionByContribution$ y se toman $i \in sol.v$ en orden creciente y $j \notin sol.v$ para el intercambio. Este j se tomará de forma aleatoria y con el objetivo de explotar plenamente la ordenación utilizada generaremos múltiples j's para cada i.

Según el guión de prácticas hemos de generar hasta $MAX = 50{,}000$ vecinos (parejas (i, j) en nuestro caso) antes de parar la búsqueda local. Tras varias pruebas he decidido que merece la pena centrarse en el 10 % más prometedor de la solución. Llamemos a este porcentaje *percentage_studied*.

Por lo tanto estudiaremos $max_i = percentage\_studied \cdot |sol|$ elementos de la solución y para cada uno generaremos $max\_random = MAX/max\_i$ valores aleatorios distintos, obteniendo un total de $max\_i \cdot max\_random = max\_i \cdot (MAX/max\_i) = MAX$ elementos en total.

Algorithm 9: stepInNeighbourhood

```
Data: sol : solution
Result: stop : bool , sol : solution
begin
    sol ← orderSolutionByContribution(sol)
    max_i ← percentage_studied · |sol|
    max_randoms ← MAX/max_i
    stop ← true
    tries ← 0
    i ← 0
    while i < max_i do
        element_out ← sol.v[ i ]
        oldContribution ← singleContribution(sol.v, element_out)
        j ← rand(0, m)
        k ← 0
        while k < max_k do
            if j ∉ sol.v then
                newContribution ← singleContribution(sol.v, j) − MAT[j][element_out]
                if newContribution > oldContribution then
                    sol.v[ i ] ← j
                    sol.fitness ← sol.fitness + newContribution − oldContribution
                    pairs[ i ].first ← i
                    sol ← false
                    return
                end
                k ← k + 1
            end
            j ← rand(0, m)
        end
        i ← k + 1
    end
end
```

3.3. Búsqueda local determinista

Este algoritmo esta basado en la búsqueda local recién explicada pero con una pequeña mejora. A la hora de explorar el vecindario también ordenaremos los elementos no seleccionados en función de los más prometedores. Para ello utilizamos la función *obtainBestOrdering*.

Algorithm 10: obtainBestOrdering

```
Data: sol : solution
Result: best_ordering : vector¡int¿
begin
    pairs : vector < pair < int, double >>
    foreach i ∈ 0, ..., n do
        if i ∉ sol.v then
            pairs[ i ].first ← i
            pairs[ i ].second ← singleContribution(sol.v, i);
        end
    end
    pairs ← sort(pairs, operator <)
    for i from |pairs| − 1 to 0 do
        best_ordering[ i ] ← pairs[ i ].first
    end
end
```

Para el algoritmo en si, repetiremos un razonamiento análogo al anterior. Fijado el número de elementos del vecindario a explorar, MAX, exploramos un porcentaje p_i de la solución y un porcentaje p_k del ordenamiento generado a partir de la función $obtainBestOrdering$.

Recorreremos la solución hasta $max_i = |sol| \cdot p_i$ y los posibles intercambios hasta $max_k = n \cdot p_k$, teniendo en cuenta que:

$$max_k \cdot max_i = (n \cdot p_k) \cdot (|sol| \cdot p_i) = MAX$$

Si damos un valor a p_i podemos calcular max_i y max_k en función del resto de datos: $max_k = MAX/max_i$. Finalmente tenemos que tener en cuenta la posibilidad de que max_k sea mayor que el tamaño de $best_ordering$ es por esto que tomamos $max_k = min(MAX/max_i, |best_ordering|)$. En ese caso hemos de actualizar max_i a $min(MAX/max_k, |sol|)$ para asegurarnos de que hacemos la MAX exploraciones.

Este movimiento esta implementado en la función $stepInNeighbourhoodDet$ que a su vez es llamado por el algoritmo final, $localSearchDet$.

Algorithm 11: stepInNeighbourhoodDet

```
Data: sol : solution
Result: stop : bool , sol : solution
begin
    sol ← orderSolutionByContribution(sol)
    best_ordering ← obtainBestOrdering(sol)
    percent_i ← 0,1 max_i ← percent_i · |sol|
    max_k ← min(MAX/max_i, |best_ordering|)
    if max_k == |best_ordering| then
        max_i ← min(MAX/max_k, |sol|)
    end
    stop ← true
    i ← 0
    while i < max_i do
        element_out ← sol.v[ i ]
        oldContribution ← singleContribution(sol.v, element_out)
        k ← 0
        while k < max_k do
            j ← best_ordering[ k ] if j ∉ sol.v then
                newContribution ← singleContribution(sol.v, j) − MAT[j][element_out]
                if newContribution > oldContribution then
                    sol.v[ i ] ← j
                    sol.fitness ← sol.fitness + newContribution − oldContribution
                    pairs[ i ].first ← i
                    sol ← false
                    return
                end
                k ← k + 1
            end
        end
        i ← k + 1
    end
end
```

Algorithm 12: localSearchDet

```
Data: none
Result: sol : solution
begin
    sol ← randomSolution()
    stop ← false
    while !stop do
        stop , sol ← stepInNeighbourhoodDet(sol)
    end
end
```

Cabe destacar que este algoritmo no es determinista por completo ya que la solución inicial tomada es puramente aleatoria. Este detalle es importante puesto que por ello merecerá la pena ejecutarlo múltiples veces en vez de una única vez.

3.4. Búsqueda local con greedy

El último algoritmo presentado es otra mejora a la búsqueda local. Consiste simplemente en tomar como solución inicial la obtenida por el greedy.

Algorithm 13: localSearchGreedy

```
Data: none
Result: sol : solution
begin
    sol ← greedy()
    stop ← false
    while !stop do
        stop , sol ← stepInNeighbourhoodDet(sol)
    end
end
```

4. Procedimiento de desarrollo

Todos los algoritmos se han implementado en $C++$ y se encuentran en la carpeta adjunta. El código esta dividido independientes y autosuficientes que contienen cada uno un algoritmos de los explicados anteriormente. Adicionalmente hay dos archivos más para el tratamiento de los datos.

Se han implementado también una serie de scripts en *bash*, así como un $makefile$ que permite la automatización de todo el proceso. El $makefile$ tiene esencialmente dos tipos de comandos: $examples < algoritmo >$ y $measure < algoritmo >$, donde $< algoritmo > \in Greedy, LS, LSD, LGS$. El primer comando compila y ejecuta $< algoritmo >$ para tres ejemplos, uno de cada set de entrenamiento. El segundo comando ejecuta 500 veces $< algoritmo >$ en cada caso del problema y hace la media de los datos proporcionados para cada uno de los casos. Los datos de salida se almacenan en $output/ < algoritmo > .dat$.

Finalmente el comando *measureAll* ejecuta *measure* sobre todos los algoritmos excepto *greedy*, ya que no tienen ningún componente aleatorio.

5. Experimentos realizados

En esta sección describiremos los experimentos realizados y estudiáremos los resultados obtenidos.

112135	0.443419
112109	0.514877
112541	0.500335
112590	0.466543
112204	0.497922
112347	0.475533
112583	0.460252
112023	0.452859
112411	0.45161
112634	0.453235
20420	0.005389
35574	0.00858
4542	0.003229
16888	0.00773
36065	0.012445
62294	0.019682
6938	0.005524
25843	0.016009
55406	0.024983
96593	0.038524
17831.8	0.004663
17973	0.005126
17973.3	0.004189
17860	0.004934
17822.9	0.004579
17741.1	0.004476
18016.5	0.005428
17897.7	0.004446
17784.5	0.005842
17860.1	0.007314