VNIVERSITAT DE VALÈNCIA

Trabajo Fin de Máster - Junio/2023

Problema de la Dispersión

Autor: **Ignacio Arnau Martínez**

Tutor: Anna Martínez Gavara

Índice

1. Introducción

En este documento vamos a estudiar el problema de la dispersión.

2. Definición del Problema

3. Programación

Como hemos comentado anteriormente, los distintos métodos que vamos a comentar los hemos desarrollado en lenguaje **Python**.
Todos los métodos que mencionáremos, se aplican sobre una instancia. Esta instancia es un archivo de texto el cuál contiene la siguiente información:

1. **Nodos:** Representa el número de nodos que componen en el problema. A partir de ahora la denominaremos como n.

2. **Matriz de Adyacencia:** Matriz con las distancias entre cada nodo. Se trata de una matriz simétrica (puesto que la distancia del nodo i al nodo j es la misma que la del nodo j al nodo i) cuya diagonal es 0, ya que la distancia de un nodo consigo mismo es nula. A la esta distancia la denominamos d_{ij}.

3. **Costes:** Coste de cada nodo. Al coste del nodo i lo denominaremos a_i.

4. **Capacidades:** Capacidad de cada nodo. A la capacidad del nodo i lo denominaremos c_i.

5. **Coste Máximo:** Coste máximo que puede tener la solución. A partir de ahora la denominaremos K

6. **Capacidad mínima:** Capacidad mínima que debe tener la solución. A partir de ahora la denominaremos B.

Indicar que las instancias también incluían el coste de unidad para cada nodo, pero no lo tendremos en cuenta para nuestro problema.

Por tanto, el primer paso antes de aplicar el método es leer correctamente la instancias o instancias que queremos simular. Una vez leída la instancia, el paso siguiente es crear la solución (de momento vacía) que nos devolverá nuestro método. De mismo que las instancias, cada solución contiene los siguientes elementos:

1. **Instancia:** Instancia sobre la que calculamos la solución.
2. **Nodos seleccionados:** Lista con los nodos que componen la solución. Esta lista la denominaremos como S.
3. **Función objetivo:** Función objetivo de la solución. En esta caso, se trata de la mínima distancia que hay entre los nodos de la solución, es decir:

$$d^* = \min d_{i,j} \quad \forall i, j \in S$$

4. **Nodos Críticos:** Para cada nodo de la solución, nodo o nodos de la solución con los que tiene la mínima distancia, por tanto:

$$cr_i = \{j / j \in S \wedge d^i_{min} = \min d_{ij}\} \quad \text{donde } d^i_{min} = \min d_{ij} \quad \forall j \in S$$

5. **Coste:** Coste total de los nodos incluidos en la solución.
6. **Capacidad:** Capacidad total de los nodos incluidos en la solución.
7. **Tiempo para la mejor solución:** Tiempo que tarda el método en encontrar la mejor solución. No hay que confundirse con el tiempo que tarda el método en ejecutarse.

Una vez que hemos creado nuestra solución, ya podemos ejecutar los métodos que hemos desarrollado.

3.1. Método LS-GRASP

El método LS-GRASP, como se puede adivinar por el nombre, consta de dos partes: Primero crear la solución mediante un algoritmo GRASP y finalmente mejorarla mediante un algoritmo de Local Search. A continuación, explicaremos detalladamente como funcionan estos dos algoritmos:

3.1.1. Algoritmo GRASP

Es bastante conocido que el algoritmo GRASP (Greedy Randomized Adaptive Search Procedure) es un algoritmo greedy. Resumidamente, estos algoritmos greedy son algoritmos metaheurísticos que consiste en crear una solución eligiendo la opción óptima en cada paso local. Sin embargo, estos algoritmos tienen el inconveniente de que la soluciones halladas son muy parecidas puesto que el único factor aleatorio que hay es el nodo inicial que se añade a la solución.

De esta manera, surgen los algoritmos GRASP. La idea es muy parecida a los algoritmos greedys, pero hay más variedad entre las soluciones que se obtienen. Esto se debe a que en cada iteración del método, el nodo que se añade a la solución se escoge de forma aleatoria entre los nodos que componen la RCL (Restricted Contraints List), la cual está compuesta por aquellos nodos que superan un umbral de la función greedy (que generalmente suele ser la función objetivo del problema). Este umbral o threeshold no es fijo y se puede variar para ver como se comporta el método para distintos threesholds.

Es precisamante este threeshold por el cual conseguimos que el método explore más el espacio del problema (distintas combinaciones que generan los nodos del problema). Coloquialmente, si imaginamos que el espacio del problema es una cordillera con varias montañas, los algorit-

mos greedys escalan hasta la cima montañas y esas montañas suelen ser la misma o estar muy próximas. Por otro lado, el algoritmo GRASP también escala montañas hasta la cima pero no son siempre las mismas, si no que puede ser cualquier montaña de la cordillera. Sin embargo, tampoco podemos asegurar que la montañas que escala sean las más altas de la cordillera.

Retornando al proyecto, el algoritmo GRASP que hemos planteado es muy similar al algoritmo canónico. El pseudo-código de este algoritmo es el siguiente:

Algorithm 1: Algoritmo GRASP

```
1  S ← ∅;
2  CL ← X;
3  i ← Random(CL);
4  S ← {i};
5  Evaluate(S);
6  CL ← Update(CL);
7  while CL ≠ ∅ do
8  |   g_min = mín_{x∈CL} g(x);
9  |   g_max = máx_{x∈CL} g(x);
10 |   μ ← g_min + α · (g_max − g_min);
11 |   RCL ← {i ∈ CL / g(i) ≥ μ};
12 |   S ← Random(RCL);
13 |   Evaluate(S);
14 |   CL ← Update(CL);
15 end
16 return S
```

El algoritmo comienza añadiendo a la solución un nodo al azar y se crea la CL (lista de candidatos), la cual está compuesta por aquellos nodos que se puedan añadir a la solución sin violar las restricciones, que en este caso solo se trata de la restricción de coste. Una vez creada la CL, el

método comienza a iterar.

En cada iteración del método, evaluamos la función greedy para cada nodo de la CL y según el threeshold escogido creamos la RCL. La condición que hemos escogido para que un nodo se añada a la RCL es la siguiente:

$RCL = \{i \ / \ g(i) \geq \min_{j \in CL} g(j) + \alpha \cdot (\max_{j \in CL} g(j) - \min_{j \in CL} g(j)\}$ donde $g(i)$ es la evaluación de la función greedy para el nodo i y α es el threeshold escogido.

Cuando ya hemos creado la RCL, añadimos a la solución un nodo al azar de la RCL y volvemos a crear la CL para la solución actual. El algoritmo acaba cuando no se pueden añadir nodos a la solución, es decir, cuando la CL está vacía.

A simple vista se puede deducir que la elección de la función greedy g es muy importante y que es la principal responsable de la calidad del método, y por tanto de las soluciones. Por ello, en este trabajo hemos empleado varias funciones greedy, que son:

1. **ProfitableGreedyFuncion:** Esta función se define como:

   $$g(i) = \frac{capacidad(i)}{coste(i)} \quad \text{donde } i \in CL$$

   Como se puede deducir, esta función evalúa como de rentable es el nodo según el coste y la capacidad de éste, es decir, que nodo tiene más capacidad a menor coste. Por tanto, cuanto mayor sea el valor de $g(i)$ mayor será la rentabilidad del nodo.
   La ventaja de esta función es que las soluciones son muy comúnmente factibles. El inconveniente sin embargo, es que no tiene en cuenta las distancias con los nodos de la solución por lo que las soluciones que se hallan no sean de calidad.

2. **ZZZZZZZZZZZZ**

El último parámetro que podemos variar es el threeshold, α. Conforme hemos definido la RCL, tenemos que un α cercano a 1 implica que la condición para entrar a la RCL es muy estricta y por tanto las soluciones halladas pueden ser muy parecidas (es decir, se parecería mucho al algoritmo greedy); mientras que un α cercano a 0 implicaría que la condición para entrar a la RCL es muy flexible y por tanto, prácticamente estaríamos añadiendo al azar un nodo de la CL, lo cual empeora la calidad de las soluciones creadas.

No se puede saber de manera exacta cuál es el mejor α, ya que este puede depender según la instancia que estemos simulando. Por ello, en este trabajo hemos realzado varias simulaciones con distintos valores de α para estudiar el comportamiento de las soluciones.

3.1.2. Algoritmo LS

El algoritmo LS (Local Search) es un algoritmo para mejorar la solución. Si volvemos al ejemplo de la cordillera, lo que hacemos con el algoritmo GRASP es elegir la montaña que vamos a escalar y lo que hace el algoritmo LS es escalar hasta la cima.

El algoritmo adopta este nombre puesto que este algoritmo consiste en para cada nodo, comprobar si hay algún otro nodo fuera de la solución que mejore a esta. De este modo, cuando para ningún nodo de la solución exista un nodo fuera de esta que le mejora, se acabará el método. El pseudo-código de este algoritmo es el siguiente:

Algorithm 2: Algoritmo LS

```
1  SolutionShift ← True;
2  while SolutionShift = True do
3      d*_min ← 0;
4      for i ∈ S do
5          NodeOut ← i;
6          d_out ← MinDist(NodeOut, S) ;   /* Calcula la mínima
               distancia del nodo i con la solución */
7          CL ← {CL / S = S\{i}};
8          if CL ≠ ∅ then
9              for candidate ∈ CL do
10                 d_cand ← MinDist(candidate, S);
11                 if d_cand > d_out & d_cand > d*_min then
12                     NodeIn ← candidate;
13                     d*_min ← d_cand;
14                     SolutionShift ← True;
15                 end
16             end
17         end
18         S ← Shift(S, NodeOut, NodeIn);
19     end
20 end
21 return S
```

La idea principal de este algoritmo es comprobar si para cada nodo de la solución, existe uno fuera de ella que mejora la función objetivo del problema, que en este caso es la mínima distancia. Por tanto, cuando no exista ningún nodo fuera de la solución que mejore a algún nodo de la solución, el algoritmo terminará.

De manera detallada, el algoritmo comienza probando el primer nodo de la solución, i. De este, toma la mínima distancia con la solución (que es la misma que con las de sus nodos críticos) y posteriormente calcula la

CL si el nodo i no estuviese en la solución. Si hay candidatos para añadir en la solución ($CL \neq \emptyset$), para cada candidato, calcula cual sería la mínima distancia con la solución sin el nodo i. Si hay alguna más grande (y mayor la mínima distancia de la solución), se intercambia este nodo por el nodo i. Finalmente, el algoritmo toma el siguiente nodo de la solución y vuelve a comenzar las comprobaciones.