PRÁCTICA 7

K-means

Saúl Sosa Díaz
Diseño y análisis de algoritmos
Universidad de La Laguna

Índice

1. K-means

K-Means es un algoritmo de agrupación, una técnica de análisis de datos no supervisada que se utiliza para agrupar conjuntos de datos en función de sus características o similitudes. El objetivo principal de k-means es encontrar grupos homogéneos (clusters) en un conjunto de datos donde los objetos en cada grupo sean lo más similares posible y los objetos de diferentes grupos sean lo más diferentes posible.

El algoritmo se puede orientar de maneras diversas.

1.1. Voraz

El punto de vista voraz sigue el siguiente diagrama de flujo:

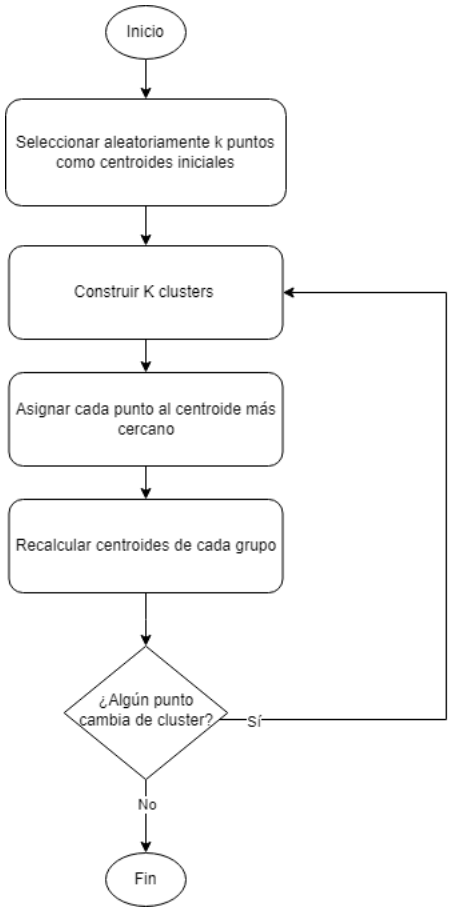

Figura 1: Diagrama de flujo de k-means voraz

1.2. GRASP

Para implementar GRASP se siguió el diagrama de flujo.

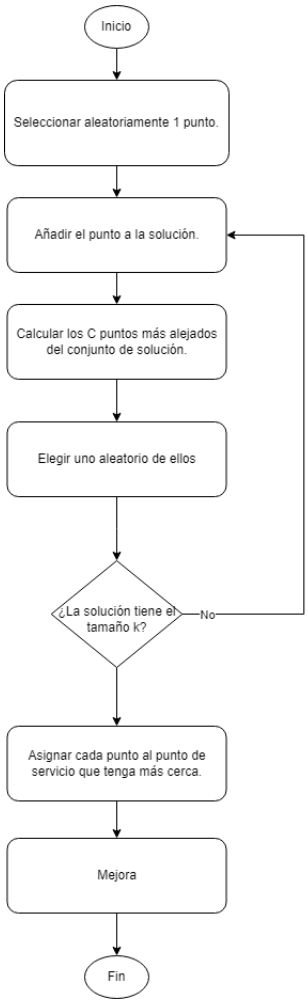

Figura 2: Diagrama de GRASP

GRASP se caracteriza por empezar con una solución vacía e ir añadiendo elementos a esa solución. Una vez terminado la fase constructiva se utiliza una fase de mejora, que para la implementación de esta práctica ha sido la del intercambio. Una de las características sobresalientes del método GRASP es su capacidad para construir soluciones a partir de una solución vacía y agregando elementos a medida que avanza el proceso. Este enfoque permite que el método sea altamente adaptable a diferentes problemas, por eso podemos aplicarlo en nuestro problema de K-means.

Una vez que se completa la fase constructiva, la solución resultante se refina aún más a través de la fase de mejora. En el caso particular donde se implementa esta práctica, se han utilizado técnicas de conmutación durante la fase de desarrollo. Esta técnica ha demostrado ser muy eficaz para mejorar la calidad de la solución y optimizar el resultado final.

2. Búsqueda

Para realizar la búsqueda local se han implementado tres movimientos.

- Intercambio: En el que se va intercambiando un punto de la solución, por un punto del problema hasta que se comprueben todas las soluciones vecinas. Este movimiento es el que más vecinas explora.
- Eliminación: Se elimina un punto de servicio a la solución.
- Inserción: Se agrega un punto de servicio a la solución

Lamentablemente, no he podido llevar a cabo la optimización para el intercambio. Es decir que no utilizo la solucion ya obtenida para evaluar la nueva solución. Sin embargo, sí he podido realizar la optimización para la eliminación y la inserción. Para ello, he seguido los siguientes diagramas de flujo:

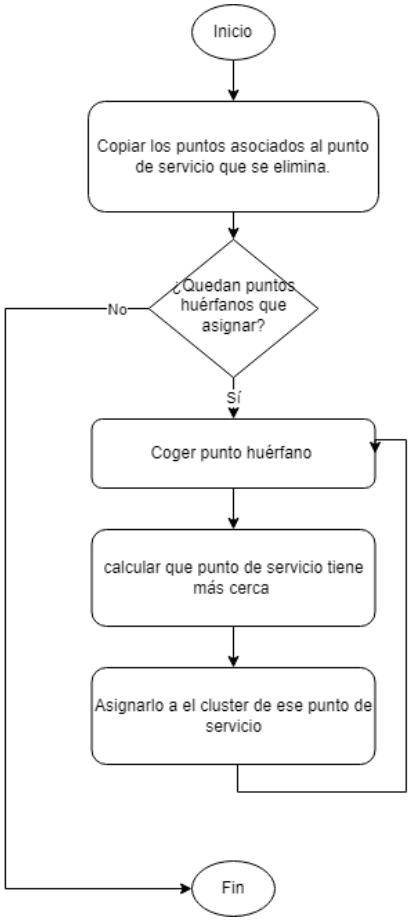

Figura 3: Optimización borrado.

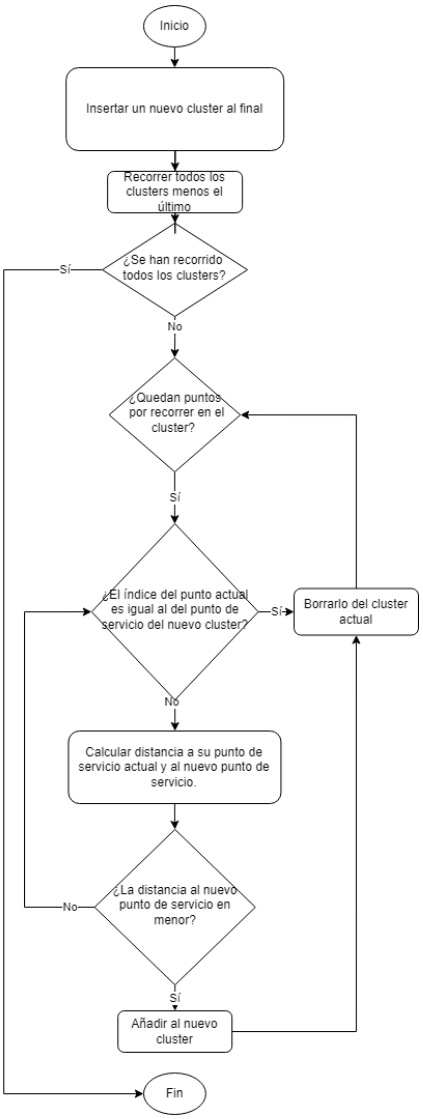

Figura 4: Optimización inserción.

3. GVNS

El algoritmo GVNS combina dos técnicas de optimización: búsqueda local y perturbación. La búsqueda local explora la vecindad de soluciones para encontrar una mejor solución. No podemos asegurar que nos de un óptimo global pero si podemos estar seguros que nos devolverá un óptimo local. En el caso de perturbaciones, se generan nuevas soluciones que no están en el vecindario existente y se busca una solución mejor que la actual.

El pseudocódigo de GVNS es el siguiente:

```
k = 1
mejorSolucion = Solucion
Solucion_Actual = mejorSolucion
while k <= Solucion.size():
  Perturbar(Solucion_Actual, k)
  Solucion_Actual = Busqueda(Solucion_Actual)
  if mejorSolucion es mejor que la Solucion_Actual:
        k = k + 1
  else:
        mejorSolucion = actualSolucion
        k = 1
```

La función **Perturbar** se encarga de realizar intercambios aleatorios en la solución, dependiendo del valor de k. Por ejemplo, si partimos de la solución ***[5, 2, 10, 6]*** y $k = 2$, una posible salida generada por la función Perturbar podría ser *[3, 1, 10, 6]*.
Es importante señalar que, si k alcanza su valor máximo, se producirá un cambio total en la solución, y se avanzará a una espacio de soluciones completamente distinto.

Para la parte de la búsqueda utilizaremos el siguiente orden de movimientos:

1. Intercambio.
2. Inserción.
3. borrado.

El Pseudocódigo para la ejecucion de estas es el siguiente:

```
l = 0
solucionAnterior = colucion
while l <= 2:
  if l == 0:
    solucion = BusquedaPorIntercambio(solucion)
    l +=1
  elif l == 1:
    solucion = BusquedaPorInsersion(solucion)
    if solucionAnterior == solucion:
      l += 1
    else:
      solucionAnterior = solucion
```

```
        l = 0
    else:
      solution = BusquedaPorBorrado(solucion)
      if solucionAnterior == solucion:
        l += 1
      else:
        solucionAnterior = solucion
        l = 0
```

Es importante mencionar que, de las tres búsquedas disponibles, solo la de intercambio se satura. Las otras dos búsquedas (inserción y eliminación) se ejecutan únicamente una vez durante la resolución del problema.

4. Detalles de la implementación.

Esta práctica está escrita en python.

4.1. Número de clusters

Para determinar el número de clusters en el método implementado, se optó por fijar este valor en un 10 % del número total de puntos presentes en el problema, con un mínimo de dos clusters. Sin embargo, el usuario tiene la opción de especificar la cantidad de clusters deseada mediante el uso del parámetro -k.

4.2. Salida de resultados

Por defecto, los resultados del método implementado se muestran en pantalla. No obstante, es posible guardar los resultados en archivos .csv si se utiliza la opción -o al momento de ejecutar el programa.

4.3. Instancias de problemas

Por defecto, el programa genera una instancia aleatoria para su ejecución. Sin embargo, si se desea utilizar una instancia previamente generada, se debe especificar el archivo correspondiente utilizando la opción -f.

4.4. Solución

Para la solución se optó por emplear la codificación indexal, una estrategia en la que se almacenan los índices de los puntos de servicio presentes en la solución en una tupla, que puede estar ordenada o no. Por ejemplo, para la Figura **??** la solución sería la siguiente:

```
[10, 7, 4, 11, 12]
```

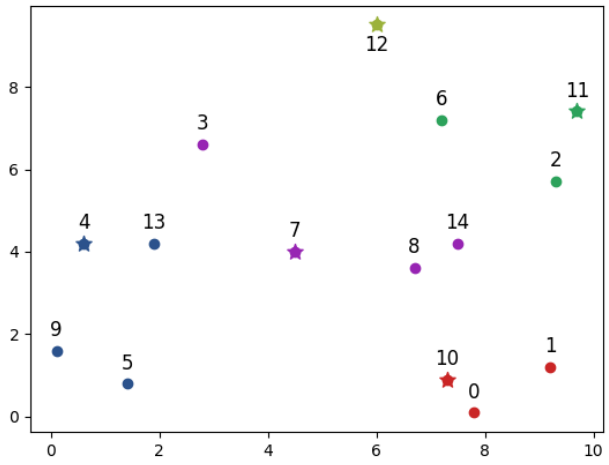

Figura 5: Gráfica de prueba con cinco clusters

Para la resolución del problema, se implementó una función objetivo que combinaba la p-mediana con un factor de penalización por el número de clusters. En este caso particular, se utilizó un factor de penalización de 10, que fue seleccionado tras realizar diversas pruebas. La fórmula es la siguiente: Donde S son los puntos que forman parte de la solución y P son todos los punntos del problema

$$\sum_{p \notin S}^{P} (d(p, PuntoServicioAsignado)) + 10 * |s|$$

En el repositorio de la práctica hay un readme en el que se explican mejor las diferentes opciones del programa, las cuales fueron nombradas en las secciones: 4.1,4.2,4.3,

5. Tablas de resultados

Para elaborar estas tablas de resultados se han creado nuevos problemas con el objetivo de analizar y comparar el comportamiento de los distintos algoritmos. Estos ejemplos son solo problemas de muestra o de prueba, y es importante tener en cuenta que pueden ser simples en comparación con problemas más complejos. Se ha llevado a cabo una evaluación del rendimiento del programa implementado para la resolución de ciertos problemas. Se ha observado que, a partir de los 40 puntos, el programa empieza a experimentar un aumento significativo en el tiempo de ejecución. En consecuencia, se han identificado problemas de escalabilidad en el programa que requieren atención y optimización.

Problem	Centroids	m	k	Objetive Value	CPU
prob1.txt	[[2.47, 4.41], [8.09, 3.79]]	15	2	61.78	0.00022
prob2.txt	[[5.07, 7.2, 4.26, 5.25], [3.41, 2.12, 3.68, 4.32]]	20	2	110.64	0.00038
prob3.txt	[[6.41, 6.52, 10.91, 3.95], [5.45, 5.9, 4.25, 11.44]]	10	2	80.11	0.00021
prob4.txt	[[14.02, 12.91], [5.55, 5.16], [8.19, 12.04]]	27	3	126.56	0.00046
prob5.txt	[[12.34, 12.32, 7.94], [6.21, 5.64, 8.81]]	19	2	144.76	0.00033
prob6.txt	[[3.14, 9.21, 6.8, 11.61], ... , [13.09, 10.35, 6.78, 7.4]]	25	3	200.03	0.00058
prob7.txt	[[4.55, 6.68, 3.57], [4.28, 12.4, 6.66]]	11	2	68.72	0.00019
prob8.txt	[[3.06, 12.61, 6.7, 9.83], ... , [6.22, 5.55, 7.75, 3.49]]	29	3	203.98	0.000661
prob9.txt	[[9.98, 12.29, 5.68], [4.16, 5.81, 3.42], [10.27, 6.15, 11.58]]	27	3	154.302	0.0005603
prob10.txt	[[3.91, 12.68, 10.48], [12.32, 12.13, 6.21], [9.37, 3.4, 4.74]]	24	3	142.700	0.00051

Cuadro 1: Tabla resultado Voraz

Problem	Point of services	m	k	$\|LRC\|$	Objetive Value	CPU
prob1.txt	[14, 13]	15	2	3	59.68	0.004333599999999993
prob2.txt	[11, 17]	20	2	3	108.93	0.011771900000000002
prob3.txt	[2, 4]	10	2	3	80.75	0.0017209999999999726
prob4.txt	[20, 3, 25]	27	3	3	118.56	0.03588089999999999
prob5.txt	[11, 8]	19	2	3	138.83	0.008794599999999986
prob6.txt	[0, 3, 9]	25	3	3	207.05	0.044454800000000017
prob7.txt	[5, 10]	11	2	3	66.02	0.0025735000000000063
prob8.txt	[12, 19, 28]	29	3	3	206.48	0.0772679
prob9.txt	[21, 6, 4]	27	3	3	149.1	0.05487289999999995
prob10.txt	[12, 23, 0]	24	3	3	140.82	0.0353405

Cuadro 2: Tabla resultado GRASP

Problem	Points of services	m	k initial	k final	Objetive Value	CPU
prob1.txt	[14, 13]	15	2	2	59.68	0.07897769999999998
prob2.txt	[7, 17, 16]	20	2	3	104.77	0.44983290000000004
prob3.txt	[2, 4, 3, 0]	10	2	4	75.41	0.13383620000000002
prob4.txt	[6, 13, 25, 24]	27	3	4	103.36	0.9470333
prob5.txt	[1, 15, 18, 13]	19	2	4	113.8	0.5590890999999999
prob6.txt	[22, 7, 9, 17, 13, 24, 14, 12]	25	3	8	178.99	5.8486329
prob7.txt	[5, 10, 4]	11	2	3	65.43	0.1124888
prob8.txt	[1, 14, 23, 6, 15, 13, 2, 18]	29	3	8	190.13	5.9049917
prob9.txt	[21, 6, 26, 23, 7, 1]	27	3	6	137.61	2.0314335999999997
prob10.txt	[0, 12, 17, 22]	24	3	4	133.48	1.3640891000000002

Cuadro 3: Tabla de resultados de GVNS

Una observación inicial y clara es que el enfoque voraz, en su mayoría, presenta peores resultados que los otros dos enfoques. A pesar de esto, tiene una menor duración en la ejecución. Los errores en estos ejemplos no difieren significativamente con los otros dos enfoques. Por otro lado, GRASP tiende a mejorar ligeramente el valor objetivo obtenido por el enfoque voraz, aunque su tiempo de ejecución suele ser mayor.

Centrándonos en GVNS, esta técnica siempre obtendrá un mejor valor objetivo que su contraparte en GRASP, al partir de la solución optenida en GRASP. No obstante, GVNS suele tardar significativamente más que los otros dos enfoques. Esto se debe no solo al tiempo que necesita para realizar sus movimientos y evaluar las soluciones, sino también al tiempo requerido para proporcionarle una solución inicial.

Por ejemplo, en el archivo de prueba *prob8.txt*, GVNS tarda casi 6 segundos en completar su ejecución, pero mejora el valor objetivo en más de 16 unidades en comparación con GRASP. En consecuencia, el debate a considerar es si se buscan los mejores resultados o se prefiere la velocidad de ejecución.