Sebastián Daniel Tamayo Guzmán

Maximum Diversity Problem
Ramificación y poda.
Diseño y desarrollo de algoritmos

Índice.

Introducción.

A lo largo de este informe se estudiarán diversos algoritmos de búsqueda aplicados al *Maximum Diversity Problem*. Se plantea un escenario en el que dado un conjunto de **n** puntos, que serán definidos por sus coordenadas de acuerdo a su dimensión **k**, se pretende encontrar el subconjunto de puntos de tamaño **m** cuyo valor de diversidad sea el máximo posible. Este valor de diversidad será la suma de las distancias euclideas entre todos los pares de puntos del subconjunto.

Para conseguir dicho objetivo se han implementado cuatro tipos de algoritmos de búsqueda, que serán explicados más en detalle en los siguientes apartados. Se trata de un algoritmo voraz constructivo, otro basado en la búsqueda local, un procedimiento de búsqueda adaptativa aleatoria (GRASP) y un algoritmo de ramificación y poda.

Además, antes de profundizar en los algoritmos se explicará la estructura de programación que los engloba, es decir las clases y métodos de los que se hará uso. Por último, en la fase de evaluación experimental se presentarán una serie de tablas que mostraran varias ejecuciones de dichos algoritmos sobre distintos casos prácticos, con el objetivo de comparar los resultados y realizar el estudio.

Lenguaje, estructura y recursos del programa.

He decidido utilizar el lenguaje C++, pues es ideal para este problema dado que principalmente se busca potencia computacional para encontrar mejores soluciones con mayor rapidez. A continuación enumeraré las clases que componen mi programa así como sus atributos y métodos relevantes.

• Class Point

Empezando por lo más básico, he creado esta clase para la representación de los puntos con los que se trabaja en el problema. Cada objeto tendrá como atributo sus coordenadas, que será una lista de números decimales, y un número que indicará su dimensiónes.

Sus métodos mayormente son para acceder y modificar sus atributos. Además **toString** genera una representación en texto del punto en cuestión. Merece la pena mencionar también el método **getDistanceTo** que recibirá otro punto como parámetro y devolverá el valor correspondiente a la distancia entre el invocante y el parámetro.

Por último, para facilitar el acceso a las coordenadas se ha sobrecargado el operador **[ ]**, de modo que se puede acceder a ellas mediante el índice correspondiente.

• Class PointSet

Esta clase servirá para representar los conjuntos de puntos con los que se trabajará. De esta manera, tendrá como atributos un vector de puntos, un entero que indicará la dimensión de dichos puntos y un puntero a un objeto de la clase abstracta **MDAlgorithm**, que será utilizada para el cálculo del subconjunto de diversidad máxima. Dicho puntero está nombrado como *MDSubsetGenerator*. Esta estructura se debe a que mi programa sigue el patrón de diseño conocido como estrategia. En el siguiente apartado explicaré un poco más a detalle dicha clase y sus descendientes.

Respecto a los métodos, además de los necesarios para acceder y modificar sus atributos, merecen mención los siguientes:

- **loadFromFile**

  Recibe una cadena de texto correspondiente a la ubicación de un archivo de texto que contenga la descripción de un caso práctico y procesa dicho archivo para cargar los valores especificados en el archivo de entrada en los atributos del objeto.

- **generateMDSubset**

  Realizará la llamada correspondiente al objeto al que apunta su atributo *MDSubsetGenerator* para iniciar la ejecución del algoritmo especificado y así generar el conjunto de diversidad máxima encontrada por el algoritmo.

- **getGravityCenter**

  Devolverá el centro de gravedad del conjunto invocante. El centro de gravedad será el punto cuyas coordenadas corresponden a la media de coordenadas de los puntos integrantes del conjunto, es decir el punto medio entre todos los del conjunto.

- **getDiversityValue**

  Devolverá el valor de diversidad asociado al conjunto invocante. Como se explicó en la introducción, éste valor corresponde al sumatorio de las distancias entre todos los pares de puntos del conjunto.

- **getFarthestPointTo**

  Devolverá el punto perteneciente al conjunto invocante que se encuentre a una distancia más lejana de un punto que será pasado por parámetro.

- **insert, extract y substract.**

  Estos métodos serán utilizados para modificar el conjunto, pues **insert** servirá para añadir puntos al conjunto y **extract** servirá para quitarlos. El método **substract** recibirá otro conjunto de puntos por parámetro y

devolverá el conjunto que contenga todos los puntos del conjunto invocante, si y solo si estos no se encuentran en el conjunto pasado por parámetro.

- **toString**

  Generará una cadena de texto que representará al conjunto invocante.

Por último, en esta clase también se sobrecarga el operador **[]**, con la finalidad de facilitar el acceso a los puntos que contenga mediante índices.

● Class MDAlgorithm

Esta clase como comenté anteriormente es una clase abstracta sin atributos y un único método virtual **run()** que será sobre escrito por sus clases descendientes para ejecutar el algoritmo correspondiente y devolver el conjunto de diversidad mínima encontrado. De esta clase heredarán las clases **ConstructiveGreedy**, **LocalSearch**, **Grasp** y **BranchAndBound**, las cuales explicaré mas en detalle en los siguientes apartados.

● Class Node

El algoritmo de ramificación y poda precisa la construcción de un arbol cuyos nodos representen soluciones parciales o totales al problema que se plantea resolver. Esta clase hace posible la representación de dichos nodos. De esta forma tendrá como atributos un conjunto de puntos, que contendrá la solución total o parcial asociada al nodo, una lista de sus hijos representada como un vector de punteros a otros nodos, un valor decimal que contendrá la cota superior de la solución asociada al nodo, un entero que indicará la profundidad del nodo y un booleano para guardar un registro de si el nodo ha sido podado.

Respecto a sus métodos, aparte de los necesarios para acceder o modificar sus atributos, cabe destacar el método **draw**, que imprimirá por consola el árbol formado por el nodo invocante como nodo raíz y, de manera recursiva, todos aquellos que desciendan de él, mostrando por cada nodo todos sus atributos.

Descripción de los algoritmos implementados.

Todos los algoritmos que se van a explicar a continuación reciben como parámetros un conjunto de puntos, del cual se generará el subconjunto de diversidad máxima, y un entero correspondiente al tamaño deseado para el subconjunto.

- ## Voraz constructivo.

El primer paso será calcular el centro de gravedad del conjunto inicial. Una vez calculado se buscará el punto perteneciente al conjunto que se encuente más alejado a dicho centro de gravedad y se insertará en el subconjunto. A continuación, se iniciará un proceso que consistirá en recalcular el centro de gravedad del subconjunto, pues ya no está vacio, y realizar la inserción del punto perteneciente al conjunto inicial que se encuentre más alejado del último centro de gravedad calculado. Este proceso se repetirá hasta que el subconjunto contenga el número especificado de puntos.

- ## Búsqueda local.

Éste algoritmo en sí no es solo una búsqueda local, pues por definición una búsqueda local no es un procedimiento que por si solo pueda encontrar un subconjunto de diversidad máxima dado un conjunto de puntos. La búsqueda local se encarga de mejorar una solución obtenida por otro procedimiento, creando una estructura de soluciones vecinas y buscando en ella un óptimo local.

En este caso, para generar la solución inicial, utilicé el mismo procedimiento en el que se basa el algoritmo voraz constructivo. A continuación, se genera el espacio de soluciones obtenidas al realizar intercambios entre puntos del conjunto inicial que quedaron fuera de la solución inicial y puntos que si se encuentran en la solución inicial. A medida que las soluciones vecinas se generan, se guarda un registro de la solución cuyo valor de diversidad sea mayor.

- **GRASP.**

Este algoritmo está implementado con la premisa de ser **multiarranque**, es decir que se pondrá en marcha un número de veces especificado. Además, existen dos formas de contabilizar estas iteraciones, de manera normal o de manera que se contabilicen solo aquellas iteraciones en las que no se consiga una solución mejor al resto de soluciones generadas en las iteraciones anteriores. Un arranque estará conformado por dos fases principales, la **constructiva**, que se encargará de generar una solución inicial y la de **mejora,** que se encargará de mejorar dicha solucion.

La fase constructiva generará lo que se conoce como LRC, lista restringida de candidatos, cuyo tamaño **r** será especificado al instanciar un objeto de esta clase. Esta lista contendrá los **r** puntos cuyo aporte al valor de diversidad del subconjunto fuese mayor si fuesen añadidos a la solución. A continuación, se realiza una inserción aleatoria de uno de los candidatos de la lista y se recalcula la LRC para poder continuar con este proceso de manera iterativa hasta conseguir que la solución sea del tamaño requerido. Cabe destacar que en la primera iteración de este proceso, al estar vacía la solución, se toma como referencia el aporte al valor de diversidad del conjunto inicial, a diferencia del resto de iteraciones en las que se toma como referencia el aporte al valor de diversidad de la solucion generada.

En la fase de mejora, se utiliza el procedimiento descrito en el apartado anterior para realizar una búsqueda local sobre la solución generada previamente y encontrar un máximo local. Éste proceso de mejora se repetirá sobre la solución mejorada hasta que no se encuentre ningún máximo local en el espacio de soluciones vecinas generado.

Como se explico al principio, las dos fases descritas anteriormente conformarán la ejecución de un arranque. Se realizarán tantos arranques como sean especificados y se devolverá la mejor solución encontrada entre todos ellos.

- **Ramificación y poda.**

En el problema de diversidad máxima, podemos definir el conjunto inicial de tamaño **n** como *V = { V1, V2, ..., Vn}*. Así mismo, podemos definir una solución total de tamaño **m** como *St = {St1, St2, ... , Stm}* y una solución parcial, es decir una solución no completa de tamaño **f** como *Sp = {Sp1, Sp2, ..., Spf}*. Para esto tenemos que **(n>m>f)** . Si definimos **q** = **m - f**, una solución parcial necesitará la unión de un conjunto *Su = {Su1, Su2, ..., Suq}* para convertirse en una solución total. Cabe destacar que para cualquier elemento *Sui* de *Su*, *Sui* $\in$ (*V - St*). Entonces, tomando como punto de partida una solución parcial, de ella se pueden generar tantas soluciones globales como conjuntos *Su* distintos existan.

Aprovechando este planteamiento, este algoritmo pretende explorar el arbol generado por todas las soluciones parciales posibles, y las generadas a partir de estas. Cada nodo representará una solución parcial, a exepción de los nodos hoja que contendrán soluciones totales. El nodo raiz del arbol representará la solución parcial de tamaño 0 y servirá para generar los nodos de profundidad 1, es decir todas las soluciones parciales posibles de tamaño 1. Así, a medida que nos adentramos en el arbol, los hijos de un nodo serán las soluciones parciales de tamaño igual a su profundidad, que hayan sido generadas a partir del nodo padre.

Al inicio del algoritmo, se utiliza otro ya sea el voraz constructivo o el GRASP para generar una solución inicial cuyo valor de diversidad será considerado como cota inferior del valor de diversidad de la solución global. También, cada vez que un nodo del arbol sea generado se le asociará una cota superior del valor de diversidad de cualquier nodo que se genere a partir de este. De esta forma, si el valor de la cota superior de un nodo es inferior a la cota inferior registrada anteriormente, se procede a podar la rama que sería generada a partir de este nodo y así se evita la exploración de esa zona del conjunto de soluciones global, pues se tiene constancia de que no contendrá una solución mejor a la que ya hemos encontrado. Cada vez que se alcanza un nodo hoja, se analiza su valor de diversidad y si este es mayor a la cota inferior registrada, la cota inferior se actualiza al nuevo valor encontrado y se procede a podar todos los nodos cuyas cotas superiores esten por debajo del nuevo valor para la cota inferior.

Al completar la exploración de todos los nodos no podados hasta llegar a sus correspondientes nodos hoja, la solución global será aquella que haya sido asignada a la cota inferior por última vez.

Caben destacar que el orden en el que se van explorando los nodos puede afectar al rendimiento de nuestro algoritmo. En esta práctica se proponen dos criterios para elegir qué nodo será el siguiente en ser expandido, aquél que tenga un menor valor como cota superior asignado o aquél cuya profundidad sea menor.

Respecto a mi implementación del algoritmo, inicialmente guardo la solución generada por el algoritmo elegido (voraz constructivo o GRASP), como cota superior. A continuación, creo un nodo vacío que actuará como nodo raiz y le añado tantos hijos como puntos contenga el conjunto inicial, donde cada hijo representa una de las soluciones parciales de tamaño 1 posibles. Para cada uno, será necesario calcular su cota superior asociada, de la cual hablaré mas adelante en profundidad, y serán añadidos a una lista de nodos en la que guardo todos los nodos que están pendientes por expandir. Ésta lista se llama *branchables* y estará siempre ordenada según el criterio especificado, pues cada vez que se inserte un valor se buscará en qué posición ha de ser insertado para mantener el orden de la lista.

De este modo, ya tendremos completo el nivel 0 y 1 de nuestro arbol, y solo quedará ir expandiendo (o ramificando) el primer elemento de la lista *branchables* tras sacarlo de ella, hasta que la lista quede vacía. Es importante denotar que cada vez que se expanda un nodo, para cada uno de sus hijos se calculará su cota superior y si no se trata de nodos hoja, serán incluidos en la lista ordenada *branchables*. Si al generar los nodos, se detecta que estos son nodos hoja, se comprobará si el valor de diversidad de su solución total asociada es mayor a la cota inferior, para en ese caso, actualizar el valor de la cota inferior y podar aquellos nodos cuya cota superior no supere a la nueva cota inferior.

Por último, explicaré el procedimiento seguido para la obtención de la cota superior asociada a una solución parcial. El método utilizado ha sido obtenido de un estudio realizado por miembros de la Universidad de Valencia. Dado un conjunto de puntos inicial V y una solución parcial *SP* de tamaño **f**, este procedimiento se basa en la descomposición del valor de diversidad Z de *SP* en tres componentes Z1, Z2 y Z3. Donde Z1 representará la suma de distancias entre todos los pares de puntos que pertenezcan a *SP*, Z2 representará la suma de distancias entre todos los pares de

puntos cuyo primer punto pertenezca a *SP* y el segundo punto pertenezca al conjunto dado por *V-SP,* y Z3 representará la suma de distancias entre todos los pares de nodos que pertenezcan a V-SP. De esta manera se calcula una cota superior para cada uno de estos valores y se construye una cota superior para Z con la suma de dichas cotas superiores.

No será necesario calcular una cota superior para el valor Z1 pues los nodos pertenecientes a *SP* se conocen y son constantes y por tanto su aporte a Z también lo es. Respecto a Z2, para cada punto de *V-SP* se calcula la suma de sus distancias a los puntos de *SP* y se insertan en una lista ordenada de mayor a menor. Por tanto, si sumamos los **m-f** primeros valores de dicha lista, obtendremos una cota superior para Z2. Por último, respecto a Z3, se calculará para cada punto de *V-SP* sus distancias a los **m-f-1** puntos restantes de *V-SP*, y nuevamente para cada punto de *V-SP*, se calculará su máxima potencial contribución a Z3 realizando la suma de las **m-f-1** mayores distancias calculadas previamente para ese punto. Todas estas máximas potenciales contribuciones para cada punto de *V-SP* serán ingresadas en una lista ordenada de mayor a menor, de la cual se sumarán los primeros **m-f** valores, dando como resultado una cota superior para Z3. Como se explicó previamente, sumando estos tres valores obtenemos una cota superior para Z.

Evaluación experimental.

Todos los archivos de entrada utilizados son aquellos que la profesora puso en nuestra disposición. Es importante denotar que la precisión en punto flotante juega un papel importante en los resultados, pues puede parecer que varía el valor de diversidad para dos subconjuntos iguales, pero esta diferencia en decimales poco significativa se debe a la precisión al trabajar con decimales. Recomiendo tomar como referencia solo las décimas y centécimas.

- Voraz constructivo.

TABLA 1

n	K	m	z	S	CPU (s)
15	2	2	11.859216	(8.65, 9.98) (0.58, 1.29)	0.000235
15	2	3	27.372700	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23)	0.000431
15	2	4	49.546154	(8.65, 9.98) (0.58, 1.29) (9.96, 8.17) (0.16, 4.62)	0.000628
15	2	5	76.653526	(8.65, 9.98) (0.58, 1.29) (1.59, 1.57) (8.41, 9.98) (9.11, 3.23)	0.000851
20	2	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.000231
20	2	3	21.996086	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36)	0.000336
20	2	4	40.002254	(8.67, 3.17) (8.57, 8.36) (1.16, 4.47) (3.47, 9.43)	0.000364
20	2	5	62.872864	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36) (1.16, 4.47) (3.47, 9.43)	0.000559
30	2	2	11.657144	(9.84, 8.96) (2.07, 0.27)	0.000160
30	2	3	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.000322
30	2	4	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8., 1.53)	0.000538
30	2	5	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8., 1.53) (0.65, 3.26)	0.000862
15	3	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.000091
15	3	3	31.868526	(9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (5.17, 1.39, 0.91)	0.000169
15	3	4	59.763760	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (6.65, 9.45, 1.02)	0.000280
15	3	5	96.085831	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.000370
20	3	2	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.000092
20	3	3	30.872660	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34)	0.000181
20	3	4	56.690311	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (0.83, 7.06, 3.34) (9.6, 2.02, 9.)	0.000268
20	3	5	92.829750	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34) (0.83, 7.06, 3.34) (9.6, 2.02, 9.)	0.000419
30	3	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.000131
30	3	3	34.290527	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (8.32, 0.47, 8.98)	0.000213
30	3	4	63.701958	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (6.75, 0.07, 1.8) (8.32, 0.47, 8.98)	0.000355
30	3	5	99.592033	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (6.75, 0.07, 1.8) (7.54, 8.93, 9.66) (8.32, 0.47, 8.98)	0.000553

- Búsqueda local.

TABLA 2

n	K	m	z	S	CPU (s)
15	2	2	11.859216	(8.65, 9.98) (0.58, 1.29)	0.000298
15	2	3	27.372700	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23)	0.000327
15	2	4	49.546154	(8.65, 9.98) (0.58, 1.29) (9.96, 8.17) (0.16, 4.62)	0.000592
15	2	5	76.653526	(8.65, 9.98) (0.58, 1.29) (1.59, 1.57) (8.41, 9.98) (9.11, 3.23)	0.000833
20	2	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.000247
20	2	3	21.996086	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36)	0.000443
20	2	4	40.002254	(8.67, 3.17) (8.57, 8.36) (1.16, 4.47) (3.47, 9.43)	0.000733
20	2	5	62.872864	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36) (1.16, 4.47) (3.47, 9.43)	0.001154
30	2	2	11.657144	(9.84, 8.96) (2.07, 0.27)	0.000336
30	2	3	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.000659
30	2	4	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8., 1.53)	0.001127
30	2	5	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8., 1.53) (0.65, 3.26)	0.000967
15	3	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.000094
15	3	3	31.868526	(9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (5.17, 1.39, 0.91)	0.000144
15	3	4	59.763760	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (6.65, 9.45, 1.02)	0.000189
15	3	5	96.085831	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.000299
20	3	2	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.000085
20	3	3	30.872660	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34)	0.000153
20	3	4	56.690311	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (0.83, 7.06, 3.34) (9.6, 2.02, 9.)	0.000245
20	3	5	92.829750	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34) (0.83, 7.06, 3.34) (9.6, 2.02, 9.)	0.000340
30	3	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.000102
30	3	3	34.290527	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (8.32, 0.47, 8.98)	0.000190
30	3	4	63.701958	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (6.75, 0.07, 1.8) (8.32, 0.47, 8.98)	0.000416
30	3	5	99.592033	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (6.75, 0.07, 1.8) (7.54, 8.93, 9.66) (8.32, 0.47, 8.98)	0.000919

- GRASP.

Cabe destacar que en todas las tablas de este algoritmo se sigue el criterio de conteo de iteraciones en el que se cuentan todas, muestren progreso frente a las soluciones anteriores o no. La decisión de no incluir en el informe las tablas asociadas al criterio en el que se cuentan solo las iteraciones en las que no se muestra progreso se debe a que tras analizar los resultados no se encuentra una diferencia significativa y por tanto no son de mucho interes.

TABLA 3

n	K	m	It	LRC	z	S	CPU (s)
15	2	2	10	2	11.859216	(0.58, 1.29) (8.65, 9.98)	0.004584
15	2	2	10	3	11.859216	(8.65, 9.98) (0.58, 1.29)	0.004961
15	2	2	20	2	11.859216	(0.58, 1.29) (8.65, 9.98)	0.002723
15	2	2	20	3	11.859216	(8.65, 9.98) (0.58, 1.29)	0.002437
15	2	3	10	2	27.372700	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23)	0.002220
15	2	3	10	3	27.372700	(0.58, 1.29) (9.11, 3.23) (8.65, 9.98)	0.002511
15	2	3	20	2	27.372700	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23)	0.004468
15	2	3	20	3	27.372700	(0.58, 1.29) (8.65, 9.98) (9.11, 3.23)	0.004961
15	2	4	10	2	49.826782	(0.58, 1.29) (8.65, 9.98) (0.16, 4.62) (9.11, 3.23)	0.005057
15	2	4	10	3	49.826782	(8.65, 9.98) (0.58, 1.29) (0.16, 4.62) (9.11, 3.23)	0.005520
15	2	4	20	2	49.826782	(8.65, 9.98) (0.58, 1.29) (0.16, 4.62) (9.11, 3.23)	0.009834
15	2	4	20	3	49.826782	(0.58, 1.29) (0.16, 4.62) (8.65, 9.98) (9.11, 3.23)	0.010217
15	2	5	10	2	79.129532	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23) (0.16, 4.62) (9.96, 8.17)	0.007667
15	2	5	10	3	79.129532	(8.65, 9.98) (9.11, 3.23) (0.58, 1.29) (0.16, 4.62) (9.96, 8.17)	0.007244
15	2	5	20	2	79.129532	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23) (0.16, 4.62) (9.96, 8.17)	0.014756
15	2	5	20	3	79.129539	(8.65, 9.98) (0.16, 4.62) (0.58, 1.29) (9.11, 3.23) (9.96, 8.17)	0.014317

TABLA 4

n	K	m	It	LRC	z	S	CPU (s)
20	2	2	10	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.001110
20	2	2	10	3	8.510329	(8.67, 3.17) (0.63, 5.96)	0.001275
20	2	2	20	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.002343
20	2	2	20	3	8.510329	(8.67, 3.17) (0.63, 5.96)	0.002877
20	2	3	10	2	21.996086	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36)	0.002534
20	2	3	10	3	21.996086	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36)	0.002795
20	2	3	20	2	21.996086	(0.63, 5.96) (8.57, 8.36) (8.67, 3.17)	0.005767
20	2	3	20	3	21.996086	(8.57, 8.36) (8.67, 3.17) (0.63, 5.96)	0.005884
20	2	4	10	2	40.002258	(1.16, 4.47) (8.57, 8.36) (8.67, 3.17) (3.47, 9.43)	0.005120
20	2	4	10	3	40.002258	(8.67, 3.17) (8.57, 8.36) (3.47, 9.43) (1.16, 4.47)	0.006683
20	2	4	20	2	40.002258	(1.16, 4.47) (8.67, 3.17) (8.57, 8.36) (3.47, 9.43)	0.010247
20	2	4	20	3	40.002258	(8.67, 3.17) (3.47, 9.43) (8.57, 8.36) (1.16, 4.47)	0.012134
20	2	5	10	2	63.651680	(8.57, 8.36) (8.67, 3.17) (3.47, 9.43) (1.97, 3.5) (0.63, 5.96)	0.008239
20	2	5	10	3	63.651680	(0.63, 5.96) (8.57, 8.36) (8.67, 3.17) (3.47, 9.43) (1.97, 3.5)	0.010618
20	2	5	20	2	63.651680	(0.63, 5.96) (8.57, 8.36) (8.67, 3.17) (3.47, 9.43) (1.97, 3.5)	0.017191
20	2	5	20	3	63.651680	(0.63, 5.96) (8.57, 8.36) (8.67, 3.17) (3.47, 9.43) (1.97, 3.5)	0.019836

TABLA 5

n	K	m	It	LRC	z	S	CPU (s)
30	2	2	10	2	11.657144	(2.07, 0.27) (9.84, 8.96)	0.001666
30	2	2	10	3	11.657144	(9.84, 8.96) (2.07, 0.27)	0.002138
30	2	2	20	2	11.657144	(9.84, 8.96) (2.07, 0.27)	0.003593
30	2	2	20	3	11.657144	(9.84, 8.96) (2.07, 0.27)	0.003727
30	2	3	10	2	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.003325
30	2	3	10	3	28.944302	(1.91, 9.6) (9.84, 8.96) (2.07, 0.27)	0.004448
30	2	3	20	2	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.006282
30	2	3	20	3	28.944302	(9.84, 8.96) (1.91, 9.6) (2.07, 0.27)	0.008415
30	2	4	10	2	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53)	0.005876
30	2	4	10	3	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53)	0.008221
30	2	4	20	2	52.771175	(9.84, 8.96) (1.91, 9.6) (2.07, 0.27) (8, 1.53)	0.013833
30	2	4	20	3	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53)	0.016188
30	2	5	10	2	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53) (0.65, 3.26)	0.012996
30	2	5	10	3	80.910248	(9.84, 8.96) (1.91, 9.6) (2.07, 0.27) (8, 1.53) (0.65, 3.26)	0.014590
30	2	5	20	2	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53) (0.65, 3.26)	0.027478
30	2	5	20	3	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53) (0.65, 3.26)	0.028938

TABLA 6

n	K	m	It	LRC	z	S	CPU (s)
15	3	2	10	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.000924
15	3	2	10	3	13.273240	(0.3, 0.92, 4.23) (9.88, 9.88, 6.26)	0.001180
15	3	2	20	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.001938
15	3	2	20	3	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.002452
15	3	3	10	2	31.868526	(9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (5.17, 1.39, 0.91)	0.002307
15	3	3	10	3	31.868526	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (5.17, 1.39, 0.91)	0.002404
15	3	3	20	2	31.868526	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (5.17, 1.39, 0.91)	0.003935
15	3	3	20	3	31.868526	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (5.17, 1.39, 0.91)	0.004884
15	3	4	10	2	59.763763	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (6.65, 9.45, 1.02)	0.002156
15	3	4	10	3	59.763763	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (6.65, 9.45, 1.02)	0.004000
15	3	4	20	2	59.763763	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (6.65, 9.45, 1.02)	0.005094
15	3	4	20	3	59.763763	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (6.65, 9.45, 1.02)	0.006956
15	3	5	10	2	96.085831	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.005290
15	3	5	10	3	96.085831	(0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.008234
15	3	5	20	2	96.085831	(9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (0.3, 0.92, 4.23) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.011979
15	3	5	20	3	96.085831	(0.3, 0.92, 4.23) (9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.014438

TABLA 7

n	K	m	It	LRC	z	S	CPU (s)
20	3	2	10	2	11.800310	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83)	0.001268
20	3	2	10	3	11.800310	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83)	0.001423
20	3	2	20	2	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.003378
20	3	2	20	3	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.003395
20	3	3	10	2	30.872660	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83) (8.87, 9.56, 5.34)	0.002078
20	3	3	10	3	30.872660	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34)	0.002689
20	3	3	20	2	30.872660	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83) (8.87, 9.56, 5.34)	0.004576
20	3	3	20	3	30.872660	(0.65, 3.76, 9.07) (8.87, 9.56, 5.34) (9.81, 2.05, 1.83)	0.006184
20	3	4	10	2	56.690315	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.005009
20	3	4	10	3	56.690315	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.005601
20	3	4	20	2	56.690315	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.009798
20	3	4	20	3	56.690315	(9.81, 2.05, 1.83) (9.6, 2.02, 9) (0.65, 3.76, 9.07) (0.83, 7.06, 3.34)	0.011897
20	3	5	10	2	92.829750	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83) (8.87, 9.56, 5.34) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.008172
20	3	5	10	3	92.829750	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83) (9.6, 2.02, 9) (0.83, 7.06, 3.34) (8.87, 9.56, 5.34)	0.010491
20	3	5	20	2	92.829750	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83) (8.87, 9.56, 5.34) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.014986
20	3	5	20	3	92.829750	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.019278

TABLA 8

n	K	m	It	LRC	z	S	CPU (s)
30	3	2	10	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.001699
30	3	2	10	3	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.002524
30	3	2	20	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.003424
30	3	2	20	3	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.004609
30	3	3	10	2	34.290527	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (8.32, 0.47, 8.98)	0.004848
30	3	3	10	3	34.290527	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98)	0.004662
30	3	3	20	2	34.290527	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98)	0.008578
30	3	3	20	3	34.290527	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98)	0.009353
30	3	4	10	2	63.701962	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (6.75, 0.07, 1.8)	0.007281
30	3	4	10	3	63.701962	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (6.75, 0.07, 1.8)	0.008452
30	3	4	20	2	63.701962	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (6.75, 0.07, 1.8)	0.016838
30	3	4	20	3	63.701962	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (6.75, 0.07, 1.8)	0.017609
30	3	5	10	2	99.592033	(1.53, 8.09, 9.56) (6.75, 0.07, 1.8) (8.06, 9.59, 0.27) (7.54, 8.93, 9.66) (8.32, 0.47, 8.98)	0.014257
30	3	5	10	3	99.592041	(6.75, 0.07, 1.8) (1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (7.54, 8.93, 9.66)	0.013817
30	3	5	20	2	99.592041	(8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (7.54, 8.93, 9.66) (6.75, 0.07, 1.8) (1.53, 8.09, 9.56)	0.031556
30	3	5	20	3	99.592041	(6.75, 0.07, 1.8) (7.54, 8.93, 9.66) (1.53, 8.09, 9.56) (8.32, 0.47, 8.98) (8.06, 9.59, 0.27)	0.028969

- Ramificación y poda - Solucion inicial: Voraz constructivo.

Cabe destacar que la tabla 9 contiene los resultados para el algoritmo de ramificación y poda en el que se utiliza como criterio de elección de nodo a ramificar, aquel con una menor cota superior. Mientras que en la tabla 10 se utiliza como criterio de elección aquel nodo cuya profundidad sea menor.

TABLA 9

n	K	m	z	S	CPU (s)	Nodes
15	2	2	11.859216	(8.65, 9.98) (0.58, 1.29)	0.020633	225
15	2	3	27.372700	(9.11, 3.23) (0.58, 1.29) (8.65, 9.98)	0.022890	277
15	2	4	49.826782	(0.58, 1.29) (8.65, 9.98) (9.11, 3.23) (0.16, 4.62)	0.188371	2437
15	2	5	79.129532	(9.11, 3.23) (0.16, 4.62) (9.96, 8.17) (8.65, 9.98) (0.58, 1.29)	1.740061	23393
20	2	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.060561	400
20	2	3	21.996086	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36)	0.060668	400
20	2	4	40.002258	(3.47, 9.43) (8.67, 3.17) (1.16, 4.47) (8.57, 8.36)	0.503093	3456
20	2	5	63.651680	(3.47, 9.43) (8.57, 8.36) (1.97, 3.5) (0.63, 5.96) (8.67, 3.17)	4.442759	31768
30	2	2	11.657144	(9.84, 8.96) (2.07, 0.27)	0.318375	900
30	2	3	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.318519	900
30	2	4	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53)	1.338872	3868
30	2	5	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53) (0.65, 3.26)	15.236486	45382
15	3	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.020053	225
15	3	3	31.868526	(5.17, 1.39, 0.91) (1.71, 1.95, 9.22) (9.88, 9.88, 6.26)	0.024437	277
15	3	4	59.763760	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (6.65, 9.45, 1.02)	0.118689	1393
15	3	5	96.085831	(1.15, 9.21, 3.11) (9.47, 3.56, 1.83) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (9.88, 9.88, 6.26)	0.549048	6635
20	3	2	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.063106	400
20	3	3	30.872660	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34)	0.063294	400
20	3	4	56.690315	(0.83, 7.06, 3.34) (9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (9.6, 2.02, 9)	0.372686	2461
20	3	5	92.829750	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34) (0.83, 7.06, 3.34) (9.6, 2.02, 9)	1.019124	6882
30	3	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.332859	900
30	3	3	34.290527	(8.32, 0.47, 8.98) (8.06, 9.59, 0.27) (1.53, 8.09, 9.56)	0.344220	928
30	3	4	63.701958	(8.32, 0.47, 8.98) (6.75, 0.07, 1.8) (8.06, 9.59, 0.27) (1.53, 8.09, 9.56)	1.150612	3193
30	3	5	99.592041	(7.54, 8.93, 9.66) (6.75, 0.07, 1.8) (1.53, 8.09, 9.56) (8.32, 0.47, 8.98) (8.06, 9.59, 0.27)	12.266422	34931

TABLA 10

n	K	m	z	S	CPU (s)	Nodes
15	2	2	11.859216	(8.65, 9.98) (0.58, 1.29)	0.018054	225
15	2	3	27.372700	(9.11, 3.23) (0.58, 1.29) (8.65, 9.98)	0.020037	251
15	2	4	49.826782	(0.16, 4.62) (0.58, 1.29) (8.65, 9.98) (9.11, 3.23)	0.278015	3673
15	2	5	79.129532	(9.11, 3.23) (9.96, 8.17) (0.16, 4.62) (8.65, 9.98) (0.58, 1.29)	2.520017	30996
20	2	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.060560	400
20	2	3	21.996086	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36)	0.060611	400
20	2	4	40.002258	(3.47, 9.43) (1.16, 4.47) (8.57, 8.36) (8.67, 3.17)	0.526182	3621
20	2	5	63.651680	(3.47, 9.43) (8.57, 8.36) (1.97, 3.5) (0.63, 5.96) (8.67, 3.17)	6.013303	42948
30	2	2	11.657144	(9.84, 8.96) (2.07, 0.27)	0.317787	900
30	2	3	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.318338	900
30	2	4	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53)	1.334761	3868
30	2	5	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53) (0.65, 3.26)	15.237915	45382
15	3	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.020042	225
15	3	3	31.868526	(1.71, 1.95, 9.22) (5.17, 1.39, 0.91) (9.88, 9.88, 6.26)	0.023494	264
15	3	4	59.763760	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (6.65, 9.45, 1.02)	0.118886	1393
15	3	5	96.085838	(0.3, 0.92, 4.23) (1.15, 9.21, 3.11) (9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (9.47, 3.56, 1.83)	0.733630	8911
20	3	2	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.062999	400
20	3	3	30.872660	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34)	0.063151	400
20	3	4	56.690315	(0.83, 7.06, 3.34) (9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (9.6, 2.02, 9)	0.355664	2342
20	3	5	92.829750	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34) (0.83, 7.06, 3.34) (9.6, 2.02, 9)	1.022167	6882
30	3	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.332500	900
30	3	3	34.290527	(8.32, 0.47, 8.98) (8.06, 9.59, 0.27) (1.53, 8.09, 9.56)	0.344098	928
30	3	4	63.701965	(1.53, 8.09, 9.56) (8.32, 0.47, 8.98) (8.06, 9.59, 0.27) (6.75, 0.07, 1.8)	1.255157	3495
30	3	5	99.592041	(8.32, 0.47, 8.98) (6.75, 0.07, 1.8) (7.54, 8.93, 9.66) (8.06, 9.59, 0.27) (1.53, 8.09, 9.56)	12.347761	35282

- Ramificación y poda - Solucion inicial: GRASP.

Cabe destacar que la tabla 11 contiene los resultados para el algoritmo de ramificación y poda en el que se utiliza como criterio de elección de nodo a ramificar, aquel con una menor cota superior. Mientras que en la tabla 12 se utiliza como criterio de elección aquel nodo cuya profundidad sea menor.

TABLA 11

n	K	m	z	S	CPU (s)	Nodes
15	2	2	11.859216	(0.58, 1.29) (8.65, 9.98)	0.022962	225
15	2	3	27.372700	(0.58, 1.29) (9.11, 3.23) (8.65, 9.98)	0.020696	225
15	2	4	49.826782	(8.65, 9.98) (0.58, 1.29) (0.16, 4.62) (9.11, 3.23)	0.177487	2227
15	2	5	79.129532	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23) (0.16, 4.62) (9.96, 8.17)	1.489357	19581
20	2	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.063378	400
20	2	3	21.996086	(0.63, 5.96) (8.57, 8.36) (8.67, 3.17)	0.065013	400
20	2	4	40.002258	(1.16, 4.47) (8.67, 3.17) (3.47, 9.43) (8.57, 8.36)	0.481346	3208
20	2	5	63.651680	(0.63, 5.96) (8.67, 3.17) (8.57, 8.36) (3.47, 9.43) (1.97, 3.5)	3.586625	24886
30	2	2	11.657144	(2.07, 0.27) (9.84, 8.96)	0.333504	900
30	2	3	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.334577	900
30	2	4	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53)	1.367117	3868
30	2	5	80.910248	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53) (0.65, 3.26)	15.288421	45382
15	3	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.021537	225
15	3	3	31.868526	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (5.17, 1.39, 0.91)	0.022753	225
15	3	4	59.763763	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (6.65, 9.45, 1.02)	0.120794	1369
15	3	5	96.085831	(9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (0.3, 0.92, 4.23) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.518703	6158
20	3	2	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.065279	400
20	3	3	30.872660	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34)	0.066805	400
20	3	4	56.690315	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.358267	2308
20	3	5	92.829750	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83) (8.87, 9.56, 5.34) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	1.037310	6882
30	3	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.342230	900
30	3	3	34.290527	(1.53, 8.09, 9.56) (8.06, 9.59, 0.27) (8.32, 0.47, 8.98)	0.339521	900
30	3	4	63.701958	(8.06, 9.59, 0.27) (1.53, 8.09, 9.56) (6.75, 0.07, 1.8) (8.32, 0.47, 8.98)	1.104165	3026
30	3	5	99.592041	(8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (1.53, 8.09, 9.56) (7.54, 8.93, 9.66) (6.75, 0.07, 1.8)	12.162709	34602

TABLA 12

NODE BRANCH CRITERIA: DEPTH

n	K	m	z	S	CPU (s)	Nodes
15	2	2	11.859216	(8.65, 9.98) (0.58, 1.29)	0.025392	225
15	2	3	27.372700	(8.65, 9.98) (0.58, 1.29) (9.11, 3.23)	0.020489	225
15	2	4	49.826782	(8.65, 9.98) (0.58, 1.29) (0.16, 4.62) (9.11, 3.23)	0.178721	2227
15	2	5	79.129532	(8.65, 9.98) (9.96, 8.17) (9.11, 3.23) (0.58, 1.29) (0.16, 4.62)	1.492161	19581
20	2	2	8.510329	(0.63, 5.96) (8.67, 3.17)	0.062311	400
20	2	3	21.996086	(0.63, 5.96) (8.57, 8.36) (8.67, 3.17)	0.063733	400
20	2	4	40.002258	(8.67, 3.17) (8.57, 8.36) (3.47, 9.43) (1.16, 4.47)	0.478850	3208
20	2	5	63.651680	(0.63, 5.96) (8.57, 8.36) (8.67, 3.17) (3.47, 9.43) (1.97, 3.5)	3.512977	24886
30	2	2	11.657144	(2.07, 0.27) (9.84, 8.96)	0.320814	900
30	2	3	28.944302	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6)	0.323929	900
30	2	4	52.771175	(9.84, 8.96) (2.07, 0.27) (1.91, 9.6) (8, 1.53)	1.368083	3868
30	2	5	80.910248	(2.07, 0.27) (9.84, 8.96) (1.91, 9.6) (0.65, 3.26) (8, 1.53)	15.300777	45382
15	3	2	13.273240	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23)	0.021599	225
15	3	3	31.868526	(9.88, 9.88, 6.26) (1.71, 1.95, 9.22) (5.17, 1.39, 0.91)	0.022948	225
15	3	4	59.763763	(1.71, 1.95, 9.22) (9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (6.65, 9.45, 1.02)	0.122219	1369
15	3	5	96.085831	(9.88, 9.88, 6.26) (0.3, 0.92, 4.23) (1.71, 1.95, 9.22) (9.47, 3.56, 1.83) (1.15, 9.21, 3.11)	0.524925	6158
20	3	2	11.800310	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07)	0.066183	400
20	3	3	30.872660	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (8.87, 9.56, 5.34)	0.073777	400
20	3	4	56.690315	(9.81, 2.05, 1.83) (0.65, 3.76, 9.07) (9.6, 2.02, 9) (0.83, 7.06, 3.34)	0.362874	2308
20	3	5	92.829750	(0.65, 3.76, 9.07) (9.81, 2.05, 1.83) (0.83, 7.06, 3.34) (9.6, 2.02, 9) (8.87, 9.56, 5.34)	1.033919	6882
30	3	2	13.073737	(8.06, 9.59, 0.27) (6.71, 0.35, 9.42)	0.344209	900
30	3	3	34.290527	(8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (1.53, 8.09, 9.56)	0.343479	900
30	3	4	63.701958	(8.06, 9.59, 0.27) (8.32, 0.47, 8.98) (1.53, 8.09, 9.56) (6.75, 0.07, 1.8)	1.108150	3026
30	3	5	99.592041	(8.32, 0.47, 8.98) (6.75, 0.07, 1.8) (7.54, 8.93, 9.66) (8.06, 9.59, 0.27) (1.53, 8.09, 9.56)	12.189257	34628

Análisis y conclusiones.

Centrandome en los resultados del algoritmo voraz constructivo, como es de esperar es el algoritmo que menor tiempo de CPUrequiere. Además, si nos fijamos en las diferencias de tiempo de CPU entre las ejecuciones variando los valores **n**, **k** y **m**, vemos que el unico valor que influye es la **m**. Esto tiene sentido pues a el número de iteraciones depende directamente del valor **m**.

Respecto a la búsqueda local, no se ha conseguido mejora en ninguna de las soluciones frente al algoritmo anterior. Los tiempos de CPU tienden a ser mayores, aunque no de manera muy significativa.

En el algoritmo GRASP se han generado dos soluciones mejores frente a las que se encontraron con el algoritmo de búsqueda local. Estas mejoras corresponden a las ejecuciones 17-20 (Tabla 2) con una mejora de 2,47 y las ejecuciones 38-41 (Tabla 3) con una mejora de 0,78. Además, los tiempos de CPU como es esperable incrementan de manera significativa, pues se trata de un algoritmo multiarranque con un proceso de mejora cuya condición de parada puede tardar en encontrarse si se encuentran optimos locales mejores de manera consecutiva. Cabe destacar que, como ya se ha mencionado, se realizó el experimento cambiando el tipo de conteo de arranques para que no se contaran aquellos en los que se encontraba una solución mejor, pero los resultados son prácticamente los mismos. Tras analizar el numero de iteraciones totales en ambos casos, comprobé que también eran prácticamente iguales, cosa que se debe a que siempre se encontraba un óptimo local sin más óptimos locales en su entorno en los primer o segundo arranque del algoritmo.

En cuanto al algoritmo de ramificación y poda, no se observan mejoras frente al algoritmo anterior, lo cual indica que con las ejecuciones del GRASP logramos obtener los máximos globales del espacio de soluciones global. Cabe destacar que en la mayoria de casos, el tiempo de CPU aumenta drásticamente a medida que el tamaño del subconjunto buscado aumenta. Esto tiene mucho sentido, pues a mayor sea **m**, más hijos serán generados por cada nodo, concretamente el número total de nodos del arbol, sin considerar la poda ni el nodo raiz, viene dado por la siguiente expresión: **n! - (n-m)!**.

Comparando las ejecuciones cuando el algoritmo utilizado para generar una cota inferior inicial, se puede ver claramente que cuando se utiliza el GRASP se obtiene un valor estrictamente menor o igual de nodos generados al que se obtiene cuando se utiliza el constructivo voraz, diferencia que se ve especialmente en los casos en los que se habia obtenido una mejora cuando se ejecutó el GRASP, frente al constructivo voraz. Esto es totalmente lógico y esperable, pues mientras mejor sea la cota inferior inicial proporcionada al algoritmo de ramificación y poda, los nodos tardarán menos en ser podados y por tanto se explorarán menos nodos y a su vez, se generará una menor cantidad de estos. También con respecto a los nodos generados, en la mayoría de casos se generan una menor o igual cantidad cuando el criterio de elección de nodo a expandir es aquel con menor cota superior, frente a cuando se elige a aquel con menor profundidad.

Por último, resaltar de manera global que la ausencia de mejoras y el encontrar los máximos globales sin necesidad de recurrir al algoritmo de ramificación y poda, se debe a que el espacio de soluciones global de los ejemplos utilizados para el estudio es en general pequeño y por tanto, algoritmos más sencillos consiguen explorarlo en su totalidad y encontrar los óptimos globales. A causa de esto también se debe el poco tiempo de CPU y la poca candidad de nodos generados, en general, que se obtiene con el algoritmo de ramificación y poda, pues se le está suministrando la solución gobal desde el inicio del algoritmo y en consecuencia se podan la mayoría de nodos.

Como apreciación personal puedo decir que he encontrado el último algoritmo estudiado muy interesante, pues es versión elegante, visual e ingeniosa del proceso de encontrar la solución global por fuerza bruta. Además, el cálculo de las cotas superiores, a pesar de ser un poco confuso, también me parece ingenioso y acertado.

Referencias.

- PAPER - Maximum diversity problem - Upper Bound Computation.
- Patrón Estategia - Wikipedia.
- Repositorio del código.