Informe de la Práctica 8 de DAA
Curso 2023-2024

B&B
Maximum Diversity Problem

Thomas Edward Bradley

La Laguna, 6 de mayo de 2024

Agradecimientos

Maria Belen Melian Batista

Licencia

Resumen

El siguiente informe viene a redactar los conceptos, pseudocoódigos y resultados obtenidos al evaluar distribuciones de puntos en espacios de varias dimensiones a traves de distintos algoritmos (Voraz, Grasp y Ramificación/Poda). Además se han implementado dos busquedas (una local y una tabu) para mejorar los resultados obtenidos en el caso de desearlo.

Palabras clave: Maximum Diversity Problem, Optimization, Point, 2D-Space, 3D-Space, Greedy, GRASP, Local Search, Tabu Search, Branching / Pruning.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Contexto

Se desea crear un programa capaz de elegir una cantidad predeterminada de puntos, entre un conjunto dado de los cuales, de forma que se ocupe el mayor espacio posible (es decir que las distancias entre los puntos sean tan grandes como se pueda). Se consigue esto a traves de varios algoritmos, los cuales imprimen sus resultados posteriormente por consola.

Cogeremos los resultados obtenidos por cada algoritmo (como los obtenidos por cada cambio pequeño dentro de un mismo algoritmo) para posteriormente evaluar estos y compararlos entre si (de modo que podamos determinar cuales son los más rápidos, los que proveen mejores resultados y los que son más eficientes en ambos campos).

1.1.1. Objetivos

Durante el desarrollo de esta práctica, se han cumplido los objetivos listados a continuación:

- Estructura para leer los contenidos de un directorio y repartir estos a varios algoritmos
- Método facil de agregar tests adicionales
- Diseño e implementación de un algoritmo voraz
- Diseño e implementación de un algoritmo de búsqueda local
- Diseño e implementación de un algoritmo GRASP para el problema
- Diseño e implementación de un algoritmo de búsqueda tabú con memoria a corto plazo
- Diseño e implementación de un algoritmo de ramificación y poda
- Ampliar el apartado anterior para usar una búsqueda en profundidad como estrategia de ramificación

1.2. Motivación

Ampliar conocimientos sobre algoritmos de optimización, asi como mejorar las competencias informaticas. Además, conviene trabajar en un proyecto de una mayor escala a lo que uno esta acostumbrado dentor de la universidad ya que reflejará más realisticamente con lo que uno se encontrara en la vida real.

Capítulo 2

B&B - Maximum Diversity Problem

2.1. Description

En el *Maximum diversity problem* se desea encontrar el subconjunto de elementos de diversidad máxima de un conjunto dado de elementos.

Sea dado un conjunto $\mathbb{S} = \{s_1, s_2, \ldots, s_n\}$ de n elementos, en el que cada elemento s_i es un vector $s_i = (s_{i1}, s_{i2}, \ldots, s_{iK})$. Sea, asimismo, d_{ij} la distancia entre los elementos i y j. Si $m < n$ es el tamaño del subconjunto que se busca el problema puede formularse como:

$$\text{Maximizar } z = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

sujeto a:

$$\sum_{i=1}^{n} x_i = m$$
$$x_i \in \{0, 1\} \qquad i = 1, 2, \ldots, n$$

donde:

$$x_i = \begin{cases} 1 & \text{si } s_i \text{ pertenece a la solución} \\ 0 & \text{en caso contrario} \end{cases}$$

La distancia d_{ij} depende de la aplicación real considerada. En muchas aplicaciones se usa la distancia euclídea. Así:

$$d_{ij} = d(s_i, s_j) = \sqrt{\sum_{r=1}^{K} (s_{ir} - s_{jr})^2}$$

Por lo tanto, para la evaluación de una solución estamos sumando la distancia entre cada punto a cada punto ajeno a ello dentro del conjunto de la solución (teniendo cuidado en solo sumar estos valores una vez, evitando la duplicidad de distancias).

2.1.1. Representación de las soluciones

Para representar nuestra solución tenemos un array de bits (es decir, un array de valores que solo pueden tomar el valor de 0 si no esta incluido en el conjunto o 1 si efectivamente pertenece), el cual sera del mismo tamaño que la cantidad de puntos en nuestro problema a evaluar.

Dicho esto, dentro de nuestras tablas se muestra el número de los puntos (empezando a contar en 1) que se incluyen en la solucion para que dicho resultado sea más facil de percibir visualmente (pero que quede constante que no se guarda la información dentro de la clase de dicha forma).

Capítulo 3

Algoritmos

3.1. Algoritmo constructivo voraz

Un constructivo voraz

El centro de gravedad de un conjunto de elementos $X = \{s_i : i \in I\}$ con $I \subseteq \{1, 2, \ldots, n\}$ se define como

$$centro(X) = \frac{1}{|X|}\left(\sum_{i\in I} s_{i1}, \sum_{i\in I} s_{i2}, \ldots, \sum_{i\in I} s_{iK}\right)$$

Un algoritmo constructivo voraz para el *Maximum diversity problem* parte del subconjunto S formado por el elemento más alejado del centro de gravedad de $\mathbb{S}$. A continuación, añade iterativamente a este subconjunto el elemento más alejado de su centro de gravedad hasta que S tenga m elementos. El pseudocódigo de este algoritmo se muestra en la figura 3.1.

Algoritmo constructivo voraz

1: $Elem = \mathbb{S}$;
2: $S = \emptyset$;
3: Obtener $s_c = centro(Elem)$;
4: **repeat**
5: Obtener el elemento $s_* \in Elem$ más alejado de s_c;
6: $S = S \cup \{s_*\}$;
7: $Elem = Elem - \{s_*\}$;
8: Obtener $s_c = centro(S)$;
9: **until** $(|S| = m)$
10: Devolver S;

Figura 3.1: Algoritmo constructivo voraz

3.2. Búsqueda Local

A diferencia de la práctica previa, solo se implemento una búsqueda local para esta ya que solo hacia falta una. El funcionamiento básico del cúal se explicara a continuación:

- Guardamos la mejor solucion encontrada hasta el momento y el valor maximo encontrado en la última iteración (ambas se inicializan con la solución inicial generada por greedy o voraz)
- Entramos en un bucle do-while del que no salimos hasta que el máximo de la iteración actual no es mayor que el de la iteración previa
- Anidamos dos bucles for para ir recorriendo todos los puntos dentro de nuestra solución
- Si los dos puntos elegidos son distintos (i != j && ambos no son ni 0, ni 1), se intercambian y se comprueba si la nueva distancia es mayor o no que la almacenada como máximo (en el caso de que lo es, este se guarda como la nueva solución máxima)
- Al acabar los bucles for, vemos si efectivamente se ha producido una mejor solución (en el caso de que no, hemos encontrado un optimo local y este se devuelve, en el caso de que si, actualziamos el máximo previo y reiniciamos los bucles)

3.3. GRASP

Funciona igual que el algoritmo voraz, pero ahora con la diferencia de que vamos a formar una lista con los n mejores candidatos a ser insertados en la solución, y de estas elegimos una al azar (hasta que la solución contenga m puntos).

Para el algoritmo GRASP empleado, tras experimentar con varios tanmaños de instancias (veces que corremos la fase constructiva) opte por 10 ya que era un buen balance entre buenos resultados y una ejecución rápida.

Algoritmo constructivo grasp

Procedure GRASP
Begin
Preprocesamiento
Repeat
Fase Constructiva(Solución); Usando para ello una lista de n candidatos
PostProcesamiento(Solución); Busqueda local / tabú para mejorar solución
Actualizar(Solución, MejorSolución); Si es mejor almacenamos la solución mejorada
Until (Se halla llegado a las iteraciones deseadas);
End.

Figura 3.2: Algoritmo constructivo grasp

3.4. Búsqueda Tabú

La busqueda tabú se parece bastante a la búsqueda local por lo que no volveré a comentar aquellos detalles que comparten. Dicho esto, si que haré mención de todo ello en lo que se diferencian:

- **Condición de parada** - La condición de parada de nuestro bucle do-while pasa a ser un contador que sale del bucle cuando alcanza un cierto número de iteraciones. De manera parecida a como se hace en el algoritmo general de GRASP.

- **Soluciones peores** - Como nuestra solución puede empeorar, ahora tenemos que guardar la mejor solución actual (no solo la mejor de todos los casos) en cada iteración. De modo que tenemos la mejor solución de todas (la cual solo sirve para actualizarse con mejores valores y ser devuelto al final) y la mejor de la iteración actual (la que usaremos en los cálculos (usada en los calculos actuales y para pasar la solución de la iteración previa a la siguiente, despues de la cual se devolvera a su estado inicial)

- **Bloqueo de intercambios** - Ahora también tenemos memoria a corto plazo por lo que no podemos volver a realizar un intercambio hasta que hayan pasado 3 turnos de la última vez que lo hicimos. Para esto simplemente hace falta comprobar despues de verificar que son puntos distintos que no son iguales a ninguno de los movimientos ilegales (guardamos los movimientos en un vector swap, que no se mira hatsa que tiene tamaño 1). Para que funcione correctamente tambien tenemos que almacenar los valores de la mejor i y j actual para meterlos en swaps al final de los for (se quita el primer elemento en caso de que swaps tenga un tamaño mayor que 3, garantizando que solo se bloquea durante 3 turnos)

3.5. Ramificación y Poda

Para el caso de la Ramificación y Poda tenemos que recorrer el grafo de soluciones parciales (ramificación) hasta que lleguemos a una solución parcial que cumple con el número m de puntos que debe tener o es una hoja, que quiere decir se ha llegado al final de una rama del arbol (podar).

Debemos partir de una solución inicial, la cual se puede generar tanto con un algoritmo voraz como un GRASP. Tras esto cogemos el valor de las distancias sumadas y la establecemos como la cota inferior (es decir, que para explorar un nodo en el arbol este debe poseer una cota supeior mayor que la cota inferior que acabamos de establecer).

Además se pueden emplear distintas estrategias de ramificación. En particular para este trabajo se usa la tradicional (donde miramos los nodos a explorar y entramos en aquel que tenga mayor cota superior) o una enfocada a una búsqueda en profundidad (donde simplemente vamos explorando el primer nodo que cumple la condición de la cota inferior).

He de mencionar tambien que como guardo mi solución en un vecotr binario, en cada paso solo se pueden generar dos nuevos nodos (agregando un 0 a nuestra solución parcial, o agregando un 1) lo que simplifica bastante la implementación y me permite hacer un if donde escojo la primera opción o la segunda en lugar de tener que demorar el tiempo en orgnizarlos. Además es por esto que utilizo recursividad para explorar el arbol en lugar de exploración de nodos (aunque se puede explicar como tal para un comprensión más facil) ya que no lo vi necesario en el diseño.

Lo último que me gustaria mencionar es que no supe como implementar correctamente el cálculo de la cota superior por lo que de momento simplemente hace un constructivo voraz con los elementos restantes que no han sido decididos por la exploración de los nodos. Por esta razón el algoritmo no me funciona muy bien cuando se ejecuta con un algoritmo greedy como solución inicial.

A continuación esta un pseudocódigo que muestra el proceso básico de la búsqueda para el algoritmo:

1. Compute an initial lower bound *LB* with a heuristic algorithm
2. Make *ActiveNodes*={ Initial node and its child nodes in the search tree}
While (*ActiveNodes* $\neq \emptyset$)
    3. Take *Node* from *ActiveNodes*
    If (*Node* is a leaf node)
        4. Compute the value z' of its associated solution
        If ($z' > LB$)
            5. $LB = z'$
        6. Remove *Node* from *ActiveNodes*
    Else
        7. Let $Sel = \{s_1, s_2, \ldots, s_k\}$ be the partial solution associated with *Node*
        8. Compute $d_{\max}(s_j)$ for $j = 1, 2, ..., k$ and $z(v)$ as well as $d_{\min}(v)$ for $v \in V \backslash Sel$
        If($d_{\max}(s_j) < d_{\min}(v)$ for any $j = 1, 2, ..., k$ and $v \in V \setminus Sel$)
            9. Remove *Node* from *ActiveNodes*
        Else
            10. Compute the upper bound $z_1 + UB_{23}$ of *Node*
            11. Let $v_1^*, v_2^*, ..., v_{m-k}^*$ be the vertices in $V \setminus Sel$ with maximum z-values
            12. Compute the value z' of solution $x = \{s_1, s_2, \ldots, s_k, v_1^*, v_2^*,.., v_{m-k}^*\}$
            If ($z' > LB$)
                13. $LB = z'$
            If ($z_1 + UB_{23} < LB$)
                14. Remove *Node* from *ActiveNodes*
            Else If ($z' < z_1 + UB_{23}$)
                15. $IUB = \max(z, z_1 + UB_{23}(v_1^*), z_1 + UB_{23}(v_2^*),..., z_1 + UB_{23}(v_{m-k}^*))$
                If ($IUB < LB$)
                    16. Remove *Node* from *ActiveNodes*
            Else
                17. Add the child nodes of *Node* to *ActiveNodes*

Figura 3.3: Pseudocódigo para el algorítmo de ramificación y poda usado en un articulo de Elsevier, el cual se incluye en la bibliografía por si resulta de interés

Capítulo 4

Experimentos y resultados computacionales

4.1. Constructivo voraz

Voraz, m = 2

Problema	n	k	m	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	2	11.8592	7,9	28
max_div_15_3.txt	15	3	2	13.2732	9,12	19
max_div_20_2.txt	20	2	2	8.51033	18,19	20
max_div_20_3.txt	20	3	2	11.8003	13,14	23
max_div_30_2.txt	30	2	2	11.6571	9,28	27
max_div_30_3.txt	30	3	2	13.0737	7,17	31

Cuadro 4.1: Voraz, m = 2. Tabla de resultados

Voraz, m = 3

Problema	n	k	m	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	3	25.7262	4,7,9	22
max_div_15_3.txt	15	3	3	30.3241	5,9,12	25
max_div_20_2.txt	20	2	3	21.9961	9,18,19	28
max_div_20_3.txt	20	3	3	30.8727	8,13,14	33
max_div_30_2.txt	30	2	3	28.9443	2,9,28	40
max_div_30_3.txt	30	3	3	33.8423	7,17,24	45

Cuadro 4.2: Voraz, m = 3. Tabla de resultados

Voraz, m = 4

Problema	n	k	m	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	4	48.4139	4,7,9,11	29
max_div_15_3.txt	15	3	4	59.7638	5,9,11,12	33
max_div_20_2.txt	20	2	4	39.5682	3,9,18,19	37
max_div_20_3.txt	20	3	4	56.5347	3,8,13,14	42
max_div_30_2.txt	30	2	4	52.7712	2,9,11,28	53
max_div_30_3.txt	30	3	4	63.5184	7,14,17,24	60

Cuadro 4.3: Voraz, m = 4. Tabla de resultados

Voraz, m = 5

Problema	n	k	m	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	5	73.5619	2,4,7,9,11	35
max_div_15_3.txt	15	3	5	94.7487	5,9,11,12,14	41
max_div_20_2.txt	20	2	5	61.2393	3,9,13,18,19	46
max_div_20_3.txt	20	3	5	92.8298	3,8,13,14,17	52
max_div_30_2.txt	30	2	5	80.9102	2,9,11,13,28	65
max_div_30_3.txt	30	3	5	99.5088	7,14,15,17,24	75

Cuadro 4.4: Voraz, m = 5. Tabla de resultados

4.2. GRASP

Para el siguiente caso primero miraremos los resultados obtenidos empleando como estrategia de mejora para el algoritmo GRASP una búsqueda local. Cabe mencionar que para todos los ejemplos a continuación el tamaño elegido para la lista de candidatos en la fase constructiva (tras experimentar con ello) es 3. Además el nuemero de iteraciones del GRASP que se llevan a cabo es 10.

No obstante, los resultados mejoran con respecto al algoritmo voraz pero también aumenta el tiempo de ejecución.

GRASP - búsqueda local, m = 2

Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	2	3	11.8592	7,9	2005
max_div_15_3.txt	15	3	2	3	13.2732	9,12	2001
max_div_20_2.txt	20	2	2	3	8.51033	18,19	2100
max_div_20_3.txt	20	3	2	3	11.8003	13,14	2539
max_div_30_2.txt	30	2	2	3	11.6571	9,28	4099
max_div_30_3.txt	30	3	2	3	13.0737	7,17	3548

Cuadro 4.5: GRASP - búsqueda local, m = 2. Tabla de resultados

GRASP - búsqueda local, m = 3

Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	3	3	27.3727	1,7,9	3691
max_div_15_3.txt	15	3	3	3	31.8685	5,7,12	4384
max_div_20_2.txt	20	2	3	3	21.9961	9,18,19	5523
max_div_20_3.txt	20	3	3	3	30.8727	8,13,14	6828
max_div_30_2.txt	30	2	3	3	28.9443	2,9,28	9899
max_div_30_3.txt	30	3	3	3	34.2905	6,17,24	12759

Cuadro 4.6: GRASP - búsqueda local, m = 3. Tabla de resultados

GRASP - búsqueda local, m = 4

Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	4	3	49.8268	1,6,7,9	9818
max_div_15_3.txt	15	3	4	3	59.7638	5,9,11,12	12871
max_div_20_2.txt	20	2	4	3	40.0023	2,3,9,19	14734
max_div_20_3.txt	20	3	4	3	56.6903	3,13,14,17	15395
max_div_30_2.txt	30	2	4	3	52.7712	2,9,11,28	25033
max_div_30_3.txt	30	3	4	3	63.702	6,14,17,24	28169

Cuadro 4.7: GRASP - búsqueda local, m = 4. Tabla de resultados

GRASP - búsqueda local, m = 5

Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	5	3	79.1295	1,4,6,7,9	14040
max_div_15_3.txt	15	3	5	3	96.0858	4,5,9,12,14	21417
max_div_20_2.txt	20	2	5	3	63.6517	2,9,14,18,19	27258
max_div_20_3.txt	20	3	5	3	92.8298	3,8,13,14,17	28722
max_div_30_2.txt	30	2	5	3	80.9102	2,9,11,13,28	51114
max_div_30_3.txt	30	3	5	3	99.592	6,14,15,17,24	67378

Cuadro 4.8: GRASP - búsqueda local, m = 5. Tabla de resultados

A continuación en lugar de emplear una búsqueda local usaremos la búsqueda tabú. A pesar de tardar más acabamos callendo en más o menos los mismos resultados. Lo último a mencionar, se esta ejecutando la búsqueda tabú con 10 iteraciones (tanto tabú como el grasp general se ejecutan 10 veces aunque estos son valores separados).

GRASP - búsqueda tabú, m = 2

Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	2	3	11.8592	7,9	5560
max_div_15_3.txt	15	3	2	3	13.2732	9,12	6262
max_div_20_2.txt	20	2	2	3	8.51033	18,19	8597
max_div_20_3.txt	20	3	2	3	11.8003	13,14	7920
max_div_30_2.txt	30	2	2	3	11.6571	9,28	13903
max_div_30_3.txt	30	3	2	3	13.0737	7,17	14001

Cuadro 4.9: GRASP - búsqueda tabú, m = 2. Tabla de resultados

GRASP - búsqueda tabú, m = 3

Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	3	3	27.3727	1,7,9	13693
max_div_15_3.txt	15	3	3	3	31.8685	5,7,12	16844
max_div_20_2.txt	20	2	3	3	21.9961	9,18,19	19555
max_div_20_3.txt	20	3	3	3	30.8727	8,13,14	26877
max_div_30_2.txt	30	2	3	3	28.9443	2,9,28	43768
max_div_30_3.txt	30	3	3	3	34.2905	6,17,24	37411

Cuadro 4.10: GRASP - búsqueda tabú, m = 3. Tabla de resultados

GRASP - búsqueda tabú, m = 4

Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	4	3	49.8268	1,6,7,9	39330
max_div_15_3.txt	15	3	4	3	59.7638	5,9,11,12	36838
max_div_20_2.txt	20	2	4	3	40.0023	2,3,9,19	44544
max_div_20_3.txt	20	3	4	3	56.6903	3,13,14,17	46905
max_div_30_2.txt	30	2	4	3	52.7712	2,9,11,28	62982
max_div_30_3.txt	30	3	4	3	63.702	6,14,17,24	73439

Cuadro 4.11: GRASP - búsqueda tabú, m = 4. Tabla de resultados

GRASP - búsqueda tabú, m = 5							
Problema	n	k	m	$\|LRC\|$	z	s	$CPU(ms)$
max_div_15_2.txt	15	2	5	3	79.1295	1,4,6,7,9	48245
max_div_15_3.txt	15	3	5	3	96.0858	4,5,9,12,14	48777
max_div_20_2.txt	20	2	5	3	63.6517	2,9,14,18,19	69084
max_div_20_3.txt	20	3	5	3	92.8298	3,8,13,14,17	78729
max_div_30_2.txt	30	2	5	3	80.9102	2,9,11,13,28	119902
max_div_30_3.txt	30	3	5	3	99.592	6,14,15,17,24	127563

Cuadro 4.12: GRASP - búsqueda tabú, m = 5. Tabla de resultados

4.3. Ramificación y Poda

Los resultados obtenidos en este apartado son buenos (si evaluamos todos los nodos vemos que nos devuelve los mismos valores que las busquedas locales por lo que los resultados de este partado son relativamente buenos). Aunque no sean mejores que los del GRASP si que se nota una grande diferencia en cuanto al tiempo de ejecución (de nuevo, el vector binario simplifica el proceso). También he de mencionar que guardo todos los nodos generados (incluyendo la original) y no las exploradas, lo cual seria un número distinto.

Primero revisaremos los resultados obtenidos con una cota inferior incial generada por un algoritmo voraz. Como mencione antes, el cálculo de la cota superior lo hice con una evaluación voraz por lo que no se exploran muchos nodos en este apartado (cosa que se arregla como iniciamos con GRASP).
Los primeros dos conjuntos de tablas serán empleando como estrategia de ramificación la que expande el nodo con cota superior más pequeña.

Ramificación y Poda - Expande - Voraz, m = 2

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	2	11.8592	7,9	37	3
max_div_15_3.txt	15	3	2	13.2732	9,12	52	3
max_div_20_2.txt	20	2	2	8.51033	18,19	45	3
max_div_20_3.txt	20	3	2	11.8003	13,14	62	3
max_div_30_2.txt	30	2	2	11.6571	9,28	71	3
max_div_30_3.txt	30	3	2	13.0737	7,17	82	3

Cuadro 4.13: Ramificación y Poda - Expande - Voraz, m = 2. Tabla de resultados

Ramificación y Poda - Expande - Voraz, m = 3

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	3	27.2312	1,2,7	135	15
max_div_15_3.txt	15	3	3	30.3241	5,9,12	62	3
max_div_20_2.txt	20	2	3	21.9961	9,18,19	68	3
max_div_20_3.txt	20	3	3	30.8727	8,13,14	78	3
max_div_30_2.txt	30	2	3	28.9443	2,9,28	96	3
max_div_30_3.txt	30	3	3	34.2905	7,17,24	110	3

Cuadro 4.14: Ramificación y Poda - Expande - Voraz, m = 3. Tabla de resultados

Ramificación y Poda - Expande - Voraz, m = 4							
Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	4	49.8268	1,6,7,9	277	19
max_div_15_3.txt	15	3	4	59.7638	5,9,11,12	82	3
max_div_20_2.txt	20	2	4	39.5682	3,9,18,19	90	3
max_div_20_3.txt	20	3	4	56.5347	3,8,13,14	104	3
max_div_30_2.txt	30	2	4	52.7712	2,9,11,28	128	3
max_div_30_3.txt	30	3	4	63.5184	7,14,17,24	148	3

Cuadro 4.15: Ramificación y Poda - Expande - Voraz, m = 4. Tabla de resultados

Ramificación y Poda - Expande - Voraz, m = 5							
Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	5	79.1295	1,4,6,7,9	343	19
max_div_15_3.txt	15	3	5	94.7487	5,9,11,12,14	102	3
max_div_20_2.txt	20	2	5	61.2393	3,9,13,18,19	114	3
max_div_20_3.txt	20	3	5	92.8298	3,8,13,14,17	129	3
max_div_30_2.txt	30	2	5	80.9102	2,9,11,13,28	159	3
max_div_30_3.txt	30	3	5	99.5088	7,14,15,17,24	180	3

Cuadro 4.16: Ramificación y Poda - Expande - Voraz, m = 5. Tabla de resultados

Ahora miraremos los mismos resultados inicializando la cota inferior con el cálculo total de las distancias de una sdolución obtenida usando un algoritmo GRASP (aqui se nota mejor la exploración de distintos nodos). Además seguimos con un tamaño de la lista de candidatos de 3 como en las ejecuciones anteriores. Empleamos la misma estrategia de ramificación que en las tablas previas.

Ramificación y Poda - Expande - GRASP, m = 2

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	2	11.6972	2,7	121	15
max_div_15_3.txt	15	3	2	10.8584	1,12	45	3
max_div_20_2.txt	20	2	2	8.369	3,9	162	19
max_div_20_3.txt	20	3	2	10.1731	1,12	184	25
max_div_30_2.txt	30	2	2	10.11	2,11	240	23
max_div_30_3.txt	30	3	2	13.0737	7,17	77	3

Cuadro 4.17: Ramificación y Poda - Expande - GRASP, m = 2. Tabla de resultados

Ramificación y Poda - Expande - GRASP, m = 3

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	3	27.2312	1,2,7	138	15
max_div_15_3.txt	15	3	3	28.1891	1,9,12	67	3
max_div_20_2.txt	20	2	3	21.1817	3,9,19	378	39
max_div_20_3.txt	20	3	3	28.4421	1,12,14	391	29
max_div_30_2.txt	30	2	3	28.9443	2,9,28	628	57
max_div_30_3.txt	30	3	3	32.1179	1,6,17	495	35

Cuadro 4.18: Ramificación y Poda - Expande - GRASP, m = 3. Tabla de resultados

Ramificación y Poda - Expande - GRASP, m = 4

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	4	49.8268	1,6,7,9	273	19
max_div_15_3.txt	15	3	4	55.8006	1,9,12,14	407	29
max_div_20_2.txt	20	2	4	39.5682	3,9,18,19	572	39
max_div_20_3.txt	20	3	4	53.7322	1,3,13,17	508	35
max_div_30_2.txt	30	2	4	52.7712	2,9,11,28	875	57
max_div_30_3.txt	30	3	4	59.9796	1,6,17,24	953	49

Cuadro 4.19: Ramificación y Poda - Expande - GRASP, m = 4. Tabla de resultados

Ramificación y Poda - Expande - GRASP, m = 5							
Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	5	79.1295	1,4,6,7,9	342	19
max_div_15_3.txt	15	3	5	90.0101	1,4,5,12,14	486	29
max_div_20_2.txt	20	2	5	61.2393	3,9,13,18,19	756	39
max_div_20_3.txt	20	3	5	89.6383	1,3,13,14,17	724	35
max_div_30_2.txt	30	2	5	80.9102	2,9,11,13,28	1380	57
max_div_30_3.txt	30	3	5	97.2453	1,6,8,17,24	1230	49

Cuadro 4.20: Ramificación y Poda - Expande - GRASP, m = 5. Tabla de resultados

Ahora mostaré las tablas voraz pero con una estrategia de ramificación en profundidad, el resto de parámetros permanecen igual.

Ramificación y Poda - Profundidad - Voraz, m = 2

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	2	11.8592	7,9	47	1
max_div_15_3.txt	15	3	2	13.2732	9,12	40	1
max_div_20_2.txt	20	2	2	8.51033	18,19	44	1
max_div_20_3.txt	20	3	2	11.8003	13,14	48	1
max_div_30_2.txt	30	2	2	11.6571	9,28	61	1
max_div_30_3.txt	30	3	2	13.0737	7,17	67	1

Cuadro 4.21: Ramificación y Poda - Profundidad - Voraz, m = 2. Tabla de resultados

Ramificación y Poda - Profundidad - Voraz, m = 3

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	3	27.2312	1,2,7	130	14
max_div_15_3.txt	15	3	3	30.3241	5,9,12	59	1
max_div_20_2.txt	20	2	3	21.9961	9,18,19	67	1
max_div_20_3.txt	20	3	3	30.8727	8,13,14	75	1
max_div_30_2.txt	30	2	3	28.9443	2,9,28	93	1
max_div_30_3.txt	30	3	3	33.8423	7,17,24	105	1

Cuadro 4.22: Ramificación y Poda - Profundidad - Voraz, m = 3. Tabla de resultados

Ramificación y Poda - Profundidad - Voraz, m = 4

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	4	49.8268	1,6,7,9	258	18
max_div_15_3.txt	15	3	4	59.7638	5,9,11,12	78	1
max_div_20_2.txt	20	2	4	39.5682	3,9,18,19	104	1
max_div_20_3.txt	20	3	4	56.5347	3,8,13,14	100	1
max_div_30_2.txt	30	2	4	52.7712	2,9,11,28	125	1
max_div_30_3.txt	30	3	4	63.5184	7,14,17,24	143	1

Cuadro 4.23: Ramificación y Poda - Profundidad - Voraz, m = 4. Tabla de resultados

Ramificación y Poda - Profundidad - Voraz, m = 5							
Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	5	79.1295	1,4,6,7,9	324	18
max_div_15_3.txt	15	3	5	94.7487	5,9,11,12,14	117	1
max_div_20_2.txt	20	2	5	61.2393	3,9,13,18,19	111	1
max_div_20_3.txt	20	3	5	92.8298	3,8,13,14,17	124	1
max_div_30_2.txt	30	2	5	80.9102	2,9,11,13,28	157	1
max_div_30_3.txt	30	3	5	99.5088	7,14,15,17,24	179	1

Cuadro 4.24: Ramificación y Poda - Profundidad - Voraz, m = 5. Tabla de resultados

Ahora mostaré las tablas GRASP pero con una estrategia de ramificación en profundidad, el resto de parámetros permanecen igual.

Ramificación y Poda - Profundidad - GRASP, m = 2

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	2	11.6972	2,7	122	14
max_div_15_3.txt	15	3	2	11.3591	3,12	50	1
max_div_20_2.txt	20	2	2	8.369	3,9	166	18
max_div_20_3.txt	20	3	2	11.8003	13,14	57	1
max_div_30_2.txt	30	2	2	10.11	2,11	238	22
max_div_30_3.txt	30	3	2	11.3982	3,24	446	48

Cuadro 4.25: Ramificación y Poda - Profundidad - GRASP, m = 2. Tabla de resultados

Ramificación y Poda - Profundidad - GRASP, m = 3

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	3	27.2312	1,2,7	248	27
max_div_15_3.txt	15	3	3	30.5747	4,5,12	264	24
max_div_20_2.txt	20	2	3	21.1817	3,9,19	393	38
max_div_20_3.txt	20	3	3	29.7909	1,13,14	87	1
max_div_30_2.txt	30	2	3	28.9443	2,9,28	629	56
max_div_30_3.txt	30	3	3	32.1179	1,6,17	1194	81

Cuadro 4.26: Ramificación y Poda - Profundidad - GRASP, m = 3. Tabla de resultados

Ramificación y Poda - Profundidad - GRASP, m = 4

Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	4	49.8268	1,6,7,9	426	31
max_div_15_3.txt	15	3	4	55.8006	1,9,12,14	664	51
max_div_20_2.txt	20	2	4	39.5682	3,9,18,19	570	38
max_div_20_3.txt	20	3	4	56.6903	3,13,14,17	630	34
max_div_30_2.txt	30	2	4	52.7712	2,9,11,28	868	56
max_div_30_3.txt	30	3	4	60.2957	3,7,17,24	1012	48

Cuadro 4.27: Ramificación y Poda - Profundidad - GRASP, m = 4. Tabla de resultados

Ramificación y Poda - Profundidad - GRASP, m = 5							
Problema	n	k	m	z	s	$CPU(ms)$	nodos_generados
max_div_15_2.txt	15	2	5	79.1295	1,4,6,7,9	749	31
max_div_15_3.txt	15	3	5	89.068	2,5,9,11,12	496	24
max_div_20_2.txt	20	2	5	61.2393	3,9,13,18,19	8568	38
max_div_20_3.txt	20	3	5	92.8298	3,8,13,14,17	790	34
max_div_30_2.txt	30	2	5	80.9102	2,9,11,13,28	1187	56
max_div_30_3.txt	30	3	5	97.2453	1,6,8,17,24	2327	95

Cuadro 4.28: Ramificación y Poda - Profundidad - GRASP, m = 5. Tabla de resultados

Capítulo 5

Conclusiones y trabajo futuro

Si modificamos el algoritmo de ramificación y poda para evaluar todos los posibles caminos y soluciones. Vemos que el GRASP acierta en los mismos (ya sea por búsqueda local o búsqeuda tabú). De las dos opciones disponibles para ello, la búsqueda local produce los mismos resultados en menos tiempo por lo que dado el analísis de los algoritmos desarrollados es el más eficiente, en segundo lugar siendo la búsqueda tabú que obtiene los mismos resultados pero tardando más tiempo. Es por esto que el GRASP con búsqueda local es el mejor algoritmo para la precisión del resultado.

En cuanto a los algoritmos de ramificación y poda, las dos estrategias de ramificación producen resultados bastante similares. Sin embargo, la estrategia de ramificación que expande el nodo con cota superior más pequeña resulta ser el más eficaz en tiempo (analizando los casos con cota inferior inicial generada por GRASP ya que son más precisos) por lo que resultaria el más eficaz. Si analizamos los tiempos y los resultados resulta que el algoritmo de ramificación sirve como un buen punto medio entre el voraz y el GRASP (tanto en tiempo como resultados). Incluso tras realizar el informe implemente un pequeño cambio al código que pilla el centro del conjunto dentro del cálculo de la cota superior, el cual mejora la calidad del resultado final sin realmente emperorar el tiempo, lo cual solidifica su posición como el algoritmo más eficiente globalmente (teniendo en cuenta la exactitud del resultado y el tiempo a la vez)

Gracias a la práctica también puedo concluir que en el caso de implementar un algoritmo que usa una búsqueda por las ramas de un árbol, conviene guardar el resultado dentro de un vector binaria dado que simplifica bastante el proceso de ramificación.

Bibliografía

Repositorio de Código
Ramificación y Poda: explicación del cálculo de cota superior