Memoria Práctica 4

Algoritmia básica

David Jiménez Omeñaca (825068)
Carlos Giralt Fuixench (831274)

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Ingeniería Informática
Universidad de Zaragoza
Zaragoza,España
2023-2024

Índice

1. Introducción

La presente memoria detalla el planteamiento, la metodología, la implementación y el análisis de resultados obtenidos durante la realización de la cuarta práctica de la asignatura *Algoritmia Básica*. Su objetivo es la implementación de dos soluciones para un problema equivalente al de las dos prácticas anteriores. Dicho problema consiste en determinar la disposición óptima de un conjunto de artículos en una página de un periódico o revista de modo que la cantidad de espacio sobrante sea la mínima posible. En particular, se pide una implementación mediante un algoritmo de ramificación y poda, así como la formalización del problema general en un problema de programación lineal (entera) paramétrico y su posterior implementación. Estos artículos, así como sus dimensiones y demás características se detallan en un fichero de entrada. Del mismo modo, los resultados obtenidos por el algoritmo se almacenan en un fichero de salida. Tanto el formato del fichero de entrada como el de salida pueden consultarse en A.

2. Metodología y recursos empleados

En esta ocasión, al tratarse de un problema análogo a uno con el que ya estábamos familiarizados, pasamos directamente a diseñar una primera aproximación al planteamiento de **ramificación y poda**. Para ello, nos basamos en el esqueleto de código visto en clase para algoritmos de **coste mínimo**. Una vez implementada nuestra solución y comprobado su correcto funcionamiento, comparando los resultados obtenidos con los de los algoritmos implementados en prácticas anteriores, pasamos a la formalización del problema como problema de programación lineal entera paramétrico. Una vez formulado el problema, implementamos su solución. Para ello, utilizamos la biblioteca *Python-PuLP* de **Python**.

3. Diseño e implementación

Se exponen a continuación las decisiones de diseño adoptadas en la realización de cada uno de los algoritmos propuestos en nuestra solución. Como rasgos generales, cabe destacar que la implementación de cada una de las soluciones ha sido realizada con el lenguaje de programación **Python** a fin de poder reutilizar algunas de las **clases** definidas en la práctica anterior. Concretamente, las clases **Solution**, **Variables** y **Article**. De nuevo, cabe destacar el uso de la librería `shapely.geometry`.

3.1. Ramificación y poda

Comenzamos con una serie de definiciones:

1. $A \in \mathbb{R}$ es el área de la página.
2. x_i son los diferentes artículos.
3. $x = (x_1, \ldots, x_k)$ es una solución parcial asociada a un nodo. Se cumple que $x_i < x_{i+1}\ \forall i$ (índices crecientes) y además los artículos x_i no intersecan entre ellos.
4. $A_x = \sum_{i=1}^{k} \text{Area}(x_i)$ es el área de la solución parcial.
5. $S_x = \{x_i \mid x_i \text{ no interseca con } x\}$ son los artículos que no intersecan con la solución x.

Con estas definiciones podemos empezar a modelar. Para el árbol, usamos una implementación de índices crecientes, donde todos los nodos son posibles soluciones. A la hora de expandir los hijos de un nodo, solo lo haremos si este no interseca con la solución del nodo padre.

Las funcion de cota para un nodo x es la siguiente:

$$c(x) = A - A_x$$

es decir, representa el área restante por meter. Igualmente definimos la función de estimación

$$\hat{c}(x) = A - A_x - \sum_{S_i \in S_x} \text{Area}(S_i)$$

Nótese que podríamos usar en vez de la suma de áreas, la área de la unión de la misma.

Para expandir, usamos una cola de doble prioridad $p_x := (p_1, p_2)$ donde $p_1 := c(x)$ es el coste del nodo e $p_2 = n_x$ el número de artículos de la solución parcial x. Esto lo hacemos para desempatar nodos con la misma prioridad, suponiendo que, soluciones con más artículos pueden ser más prometedores (cuando el coste es igual).

Notar que se cumplen las dos propiedades de la función de estimación. En efecto:

$$\hat{c}(x) \leq c(x)$$

ya que $\sum_{S_i \in S_x} \text{Area}(S_i) \geq 0$. Finalmente se cumple que $\hat{c}(x) = c(x)$ si x es un nodo óptimo ya que en ese caso $S_x = \emptyset$. Esto se debe a que, si hubiese un artículo que no interseca con x, lo podríamos añadir a la solución y ya no sería óptima. Por lo tanto, en este caso $\sum_{S_i \in S_x} \text{Area}(S_i) = 0$ y así se da la igualdad.

3.2. Programación lineal entera

El problema planteado se puede formalizar y resolver como un problema de programación lineal. Empezamos viendo una serie de definiciones que harán más fácil la comprensión del algoritmo.

Sean $\mathbf{y} = y_i$ las variables binarias de decisión tales que:

$$y_i = \begin{cases} 0 \text{ si el artículo i-ésimo } \textbf{no} \text{ forma parte de la solución} \\ 1 \text{ si el artículo i-ésimo forma parte de la solución} \end{cases}$$

Sea, también, la matriz $\mathbf{A}_{nxn} = (a_{ij})$ donde n es el número de artículos y

$$a_{ij} = \begin{cases} 0 \text{ si el artículo } i \text{ } \textbf{no} \text{ interseca con el } j \text{ o si } i = j \\ 1 \text{ si el artículo } i \text{ interseca con el } j \end{cases}$$

Esta es la matriz de intersecciones, que nos permitirá definir las restricciones de solape. Por último, sea $\mathbf{c} = c_i$ el vector de áreas de los artículos.

Así nuestro problema de optimización queda como:

$$\begin{array}{lll} \text{máx} & \sum_{i=1}^{n} c_i \cdot y_i & \\ \text{s.a} & \mathbf{y}^T \mathbf{A} \mathbf{y} = 0 & \text{restricción de solape} \\ & y_i \in \{0,1\} \quad \forall i = 1, \ldots, n & \text{restricción de dominio} \end{array}$$

Nótese que $\mathbf{y}^T \mathbf{A} \mathbf{y}$ se puede expresar como:

$$\mathbf{y}^T \mathbf{A} \mathbf{y} = \sum_{i,j=1}^{n} y_i \cdot y_j \cdot a_{ij}$$

Esta suma (de elementos no negativos) será 0 en el caso de que todos los sumandos lo sean. Pero $y_i \cdot y_j \cdot a_{ij}$ es 0 si:

1. El artículo i o el artículo j (o ambos) no están en la solución

2. En el caso de estar ambos, a_{ij} tiene que ser 0, por lo que los artículos no se intersecan.

Sin embargo, las condiciones no son lineales, pues hay dos variables de decisión multiplicando. Por lo tanto, no se trata de un problema lineal. A pesar de esto, como las variables y_i son binarias, podemos crear variables binarias artificiales x_{ij} que generen restricciones lineales análogas a las ya vistas. Para ello, necesitamos que:

$$x_{ij} = y_i \cdot y_j$$

Esto es equivalente a que:

$$x_{ij} = \begin{cases} 0 & \text{si } y_i = 0 \ \vee \ y_j = 0 \\ 1 & \text{si } y_i = y_j = 1 \end{cases}$$

Por lo tanto surgen las siguientes tres condiciones:

$$\begin{cases} x_{ij} \leq y_j \\ x_{ij} \leq y_i \\ x_{ij} \geq \ y_i + y_j - 1 \end{cases}$$

Veamos estas condiciones fuerzan el valor deseado de x_{ij}. Veámoslo:

1. Si $y_i = 0$ (igual si y_j=0) tenemos que $x_{ij} \leq 0$ por lo tanto, como x_{ij} es binaria tenemos que $x_{ij} = 0$.
2. Si $y_i = y_j = 1$ tenemos que $x_{ij} \geq 1$, por lo tanto $x_{ij} = 1$.

Así pues, el nuevo problema de programación lineal es:

$$\begin{array}{lll} \text{máx} & \sum_{i=1}^{n} c_i \cdot y_i & \\ \text{s.a} & \sum_{j>i} x_{ij} \cdot a_{ij} = 0 & \text{(restricciones de solape)} \\ & \begin{cases} x_{ij} \leq y_j \\ x_{ij} \leq y_i \\ x_{ij} \geq \ y_i + y_j - 1 \end{cases} & \forall i, j = 1, \ldots, n \quad \text{(restricciones de var auxiliar)} \\ & y_i \in \{0, 1\} & \forall i = 1, \ldots, n \quad \text{(restricciones de dominio)} \\ & x_{ij} \in \{0, 1\} & \forall i, j = 1, \ldots, n \quad \text{(restricciones de dominio)} \end{array}$$

Para simplificar el modelo podemos trabajar con los índices $i = 1, \ldots, n$ y $j > i \ , \ j \in \{1, \ldots, n\}$ ya que tanto las variables x_{ij} como los elementos a_{ij} son simétricos. A pesar de esta simplficicación, este modelo puede llegar a ser bastante complejo ya que:

1. Para n artículos tenemos
$$n + \frac{n(n+1)}{2} = \frac{n(n+3)}{2} \ \sim \ O\left(n^2\right)$$
variables de decisión y
$$1 \ + \ 3 \cdot \frac{n(n+1)}{2} \ \sim \ O\left(n^2\right)$$
condiciones.
2. Tenemos que calcular la matriz $\mathbf{A}$ que requiere de $n \cdot (n+1)/2$ comparaciones de artículos (es decir $\sim O\left(n^2\right)$).

4. Análisis temporal

Se han realizado las siguientes pruebas: dos pruebas de volumen (`Test0-1`) que nos ha permitido realizar los gráficos temporales y cuatro pruebas (`Test2-5`) de corrección que nos han permitido realizar la comparación de recorrido de nodos.

El gráfico temporal que compara el tiempo en milisegundos frente al número de artículos se puede ver a continuación:

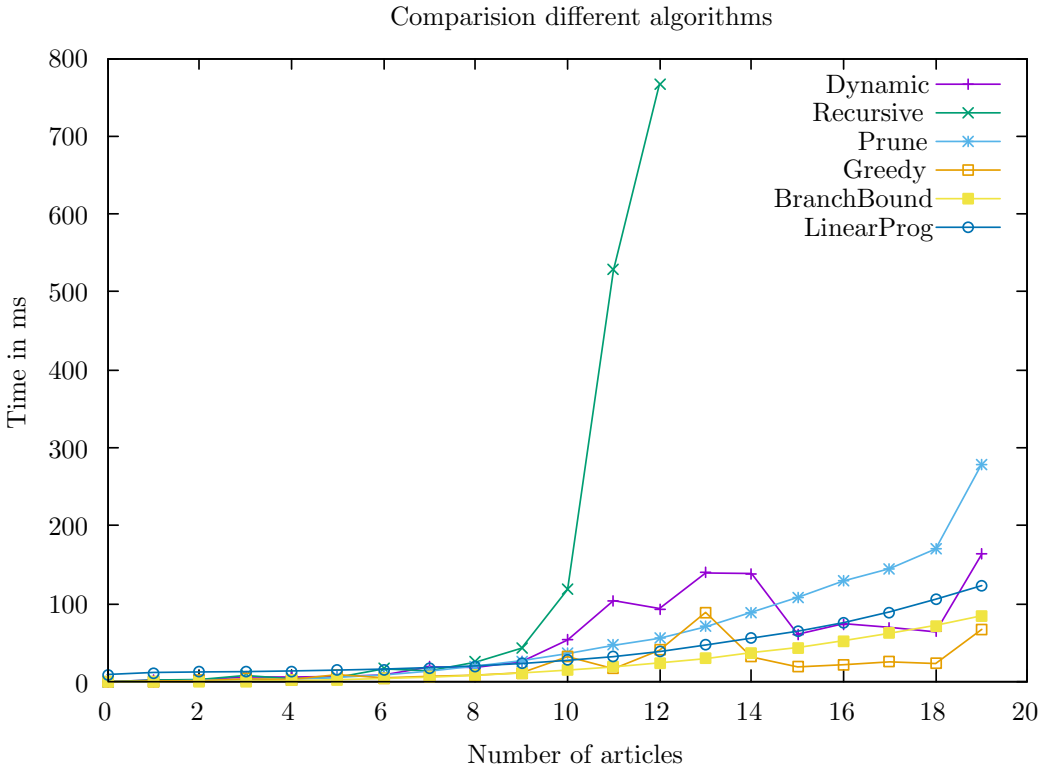

Figura 1: Comparación diferentes algoritmos

La tabla de comparación de visitas de nodos se expone a continuación:

	Backtracking/Prune	Recursive	Branch and Prune
Prueba 2	107	256	37
Prueba 3	40	256	52
Prueba 4	92	2048	35
Prueba 5	18	32	2
Average	**64,25**	**648**	**31,5**

Figura 2: Visited nodes comparison table

Se puede ver que los mejores algoritmos son los programados en la presente práctica ya que, a pesar de tener tiempos parecidos, el algoritmo voraz y dinámico son heurísticas de la solución. Además se puede ver que el algoritmo de ramificación y poda disminuye en gran medida la cantidad de nodos visitados, algo que era de esperar.

5. Resultados y conclusiones

La realización de esta práctica, junto con las dos anteriores, pone de manifiesto un principio fundamental en la computación: un mismo problema puede ser resuelto aplicando diversas metodologías algorítmicas. La calidad de resultados obtenidos depende de la elección del algoritmo empleado, pues algunos acaban siendo meras heurísticas. Por otro lado, los tiempos de ejecución dependen del tamaño del problema y aproximaciones que pueden ser muy buenas para problemas pequeños resultan ser muy ineficientes al escalarlos. De ahí la importancia de realizar un análisis previo del problema. Finalmente, cabe destacar que los algoritmos que proporcionan los mejores resultados y de la manera más eficiente son los implementados en esta práctica: algoritmos de ramificación y poda y programación lineal. Del mismo modo, para este problema en concreto, el algoritmo basado en programación dinámica proporciona meras heurísticas que, si bien pueden proporcionar soluciones óptimas, es igualmente probable que la solución a la que lleguen no sea buena.

A. Ficheros de entrada y salida

Para garantizar la correcta ejecución de la solución propuesta, se debe asegurar que los ficheros de entrada siguen el siguiente formato: el fichero de entrada está organizado en bloques. Cada bloque es una instancia diferente del problema. La primera línea de cada bloque consta de 3 números reales: el número de artíiculos n, la anchura W y altura H de la página. Las n líneas siguientes representan los artículos a colocar en la página. Para cada artículo hay cuatro números: anchura w_i, altura h_i, y coordenadas cartesianas x_i,y_i de su extremo superior izquierdo. Por último, los ficheros de entrada deben contener una línea en blanco después del último bloque de datos. A continuación se muestra un ejemplo de fichero de entrada que especifica dos instancias del problema.

```
1   3 280 400
2   140 200 0 0
3   140 200 140 0
4   200 400 0 0
5   4 280 400
6   120 120 0 0
7   130 130 0 0
8   210 210 20 10
9   260 380 20 20
10
```

Figura 3: Ejemplo de fichero de entrada

Ambas páginas tienen la misma dimensión `(280 x 400 mm)`. En la primera hay $n = 3$ artículos a colocar, mientras que en la segunda hay $n = 4$.

B. Ficheros de salida

La primera línea del fichero de salida indica el fichero de entrada a partir del cual se han obtenido los resultados. A continuación, hay tantas líneas como número de bloques tiene el fichero de entrada además de una última línea en blanco. Cada línea contiene, en primer lugar, el área (en mm) total ocupada por los artículos elegidos, el número de nodos visitados en el proceso de construcción de la solución y, finalmente, el tiempo (en ms) empleado por el algoritmo en encontrar la solución óptima. La siguiente figura contiene, a modo de ejemplo, el fichero de salida correspondiente a la ejecución de nuestra solución, usando el fichero **test0.txt** como fichero de entrada.

```
1   Results from test1.txt
2   0.0 1 0.0059604644775390625
3   0.0 1 0.0820159912109375
4   0.0 2 0.2970695495605469
5   0.0 3 0.61798095703125
6   0.0 4 1.157999038696289
7   0.0 5 1.9540786743164062
```

Figura 4: Ejemplo de fichero de salida