Práctica 6

Algoritmos probabilistas

Algoritmos Avanzados

Víctor Fernández Fernández
Mikayel Mardanyan Petrosyan

15 de Diciembre de 2019

0 Problema

El problema de la práctica es el *problema del llenado de cajas*. En este problema se dispone de n objetos, cada uno con un peso p_i, $0 \leq i \leq n$, y un suministro ilimitado de cajas de capacidad c. El problema se resuelve introduciendo los objetos en el mínimo numero de cajas.

El objetivo de la práctica es implementar al menos un algoritmo probabilista que resuelva el problema y usar OptimEx para comparar su optimalidad.

Los datos de entrada del algoritmo serán un vector llamados *ps* y un entero *c* que contendrán los pesos de los objetos y la capacidad de las cajas, respectivamente.

1 Desarrollo del algoritmo

Una manera de abordar el problema consiste en empezar sin cajas y usar una nueva cada vez que sea necesario. Cada objeto nuevo puede introducirse en una caja disponible con espacio suficiente elegida al azar o en una caja nueva. La cabecera del método será:

```
public static int menosCajas (int[] ps, int c)
```

, donde *ps* es un vector que contiene los pesos de cada uno de los objetos y *c* la capacidad de las cajas.

```
public static int menosCajasProb(int[] ps, int c) {
     // Numero de cajas menos uno
     int numCajas = 0;
     // Vector con la capacidad de las cajas
     int[] cajas = new int[ps.length];
     // Posicion del vector obtenida al azar
     int randomPos = 0;
     // Asigna la capacidad inicial a las cajas
     Arrays.fill(cajas, c);

     // Para cada objeto
     for (int i = 0; i < ps.length; i++) {
         // Obtiene una caja al azar
         randomPos = intRandom(0, numCajas);
         // Si la caja tiene capacidad para el objeto
         if (cajas[randomPos] >= ps[i]) {
             // Resta de la capacidad de la caja el
peso del objeto
             cajas[randomPos] -= ps[i];
         // Si no
         } else {
             // Usa una caja nueva
             numCajas++;
             // y resta de la capacidad inicial el peso
del objeto
             cajas[numCajas] -= ps[i];
         }
     }
     // Devuelve el numero de cajas utilizadas
     return numCajas + 1;
 }
```

El método recibe un vector de enteros que contiene el peso de cada objeto y un entero que representa la capacidad de cada caja. El máximo número de cajas que se pueden usar es el número de objetos (una caja para cada objeto), por lo que se declara un vector de n enteros llamado *cajas*. Cada posición contendrá la capacidad disponible de la caja correspondiente. Se comienza asignando la capacidad c a las cajas. Para controlar el número de cajas disponibles se declara una variable llamada *numCajas*. Se empieza con una caja, pero se inicializa la variable a 0 para indexar el vector. Aunque haya n posiciones declaradas, la variable va a limitar cuáles están disponibles para buscar solo en ellas. Para que el algoritmo sea probabilista, la caja en la que se elige meter el objeto se obtiene de forma aleatoria y se almacena

en la variable *randomPos*.

Para la inserción de cada uno de los objetos se calcula una posición aleatoria. A continuación, se consulta si el objeto cabe en la caja. Si es así, se resta a la capacidad de la caja el peso del objeto; y si no es así, se toma una caja nueva, se incrementa el número de cajas y se resta el peso del objeto a la capacidad de la nueva caja. Por último, se devuelve el número de cajas más uno.

2 Experimentación con la optimidad del algoritmo

2.1 Material del experimento

Para experimentar con la optimalidad de distintos algoritmos sobre este problema se ha utilizado los algoritmos proporcionados en el paquete de algoritmos aproximados y el algoritmo presentado en el apartado anterior. Estos son:

- Algoritmo heurístico 1 (menosCajasAprox1). Selecciona los objetos por peso creciente y los va metiendo en cajas en orden. Si un objeto no cabe en la última caja usada, lo mete en una nueva.
- Algoritmo heurístico 2 (menosCajasAprox2). Selecciona los objetos por orden decreciente y los va metiendo en la primera caja en la que caben.
- Algortimo de vuelta atrás (menosCajasBack). Explora el árbol de búsqueda completo y encuentra la solución óptima.
- Algortimo probabilístico 1 (menosCajasProb). Elige una caja al azar de las disponibles. Si el objeto cabe en ella, lo mete; si no, usa una caja nueva y lo mete en ella.

Los valores de experimentación han sido generados con las siguientes restricciones:

- Número de juegos de datos: 1000
- Tamaño de los vectores de entrada (número de objetos): 10
- Rango de peso de los objetos: 1-15
- Peso disponible en las cajas: 30 (rango 30-30).

2.2 Conclusión

Como se puede ver en la evidencia presentada en el apartado siguiente, el algoritmo probabilista 1 obtiene un porcentaje de soluciones óptimas aceptable (20%) y un valor subóptimo medio también aceptable, aunque el extremo es demasiado alto. Esto último se soluciona con la mejora propuesta en el apartado 3.

2.3 Evidencias

En las figuras 1 y 2 se observan los resultados de la experimentación con los algoritmos presentados antes. El algoritmo de vuelta atrás es exacto, ya que obtiene un 100% de soluciones óptimas. Que no haya soluciones sobreóptimas es una señal de que los algoritmos no tienen fallos.

Medidas	menosCajasAprox1	menosCajasAprox2	menosCajasBack	menosCajasProb
Núm. ejecuciones	1000	1000	1000	1000
Núm. ejec. válidas del método	1000	1000	1000	1000
Núm. ejec. válidas en total	1000	1000	1000	1000
% soluciones subóptimas	23.40 %	3.50 %	0 %	78.50 %
% soluciones óptimas	76.60 %	96.50 %	100 %	21.50 %
% soluciones sobreóptimas	0 %	0 %	0 %	0 %
% valor medio subóptimo	134.65 %	133.33 %	0 %	145.24 %
% valor extremo subóptimo	150 %	133.33 %	0 %	250 %
% valor medio sobreoptimo	0 %	0 %	0 %	0 %
% valor extremo sobreóptimo	0 %	0 %	0 %	0 %

Figure 1: Tabla de resultados numéricos

En el primer gráfico de la figura 2 se pueden apreciar los resultados mencionados anteriormente. En el segundo gráfico, el rectángulo ofrece información sobre el valor medio devuelto en los casos subóptimos y, la línea delgada, sobre la desviación máxima de los casos subóptimos. Este gráfico muestra que, los algoritmos heurísticos y los probabilistas se parecen. El algoritmo de vuelta atrás no tiene resultados subóptimos, y por ello tan solo aparece una pequeña línea.

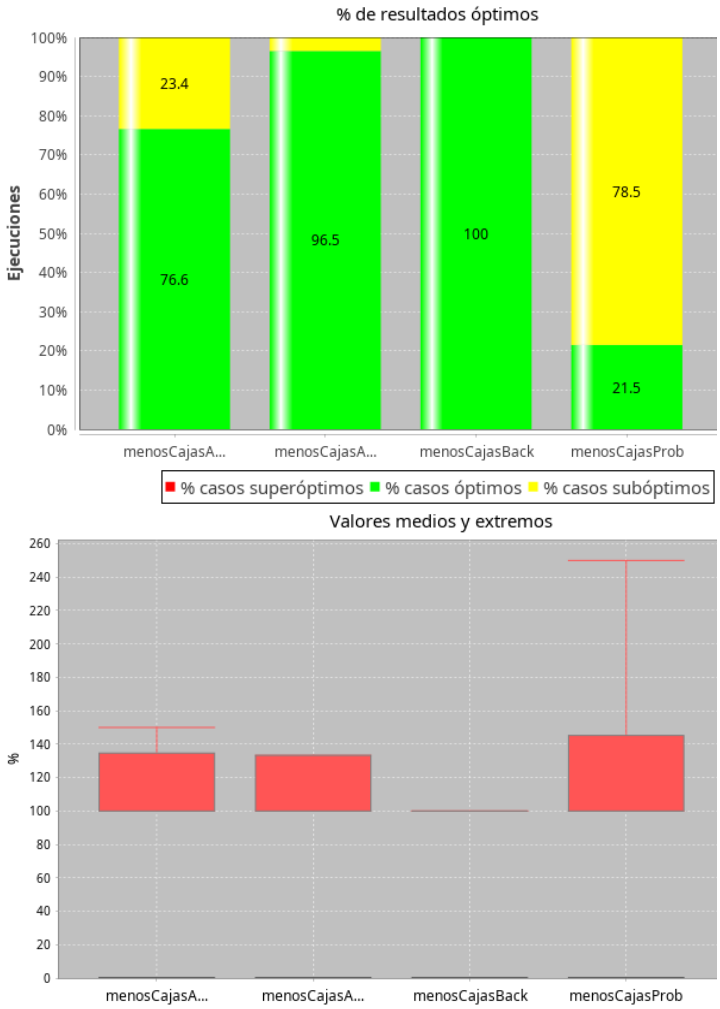

Figure 2: Gráficos

3 Mejora

Como se ve en el estudio de la optimalidad, el algoritmo presentado en el apartado 1 tiene un problema: en algunas ejecuciones utiliza muchas más cajas de las necesarias.

Para solucionar este problema simplemente es necesario limitar la generación de nuevas cajas mediante una comprobación sobre la última caja, y no sobre la caja seleccionada al azar como se hacía en el algoritmo original. Además, si el objeto actual no cabe en la caja seleccionada aleatoriamente, va a caber en otra caja porque en el peor caso se habrá creado una caja nueva, así que se vuelve a seleccionar una caja aleatoriamente hasta encontrar una

en la que quepa.

Este algoritmo se ha utilizado en la experimentación para confirmar la mejora, y la implementación es la que sigue:

```
public static int menosCajasProbMejora(int[] ps, int c){
     // Vector con la capacidad de las cajas
     int[] cajas = new int[ps.length];
     // Numero de cajas menos uno
     int nCajas = 0;
     // Caja en la que intenta meter el objeto
     int cajaObjetivo = -1;
     // Asigna la capacidad inicial a las cajas
     Arrays.fill(cajas, c);

     // Para cada objeto
     for (int i = 0; i < ps.length; i++) {
         // Si el objeto no cabe en la ultima caja
         if (ps[i] > cajas[nCajas]) {
             // Se usa una mas
             nCajas++;
         }
         // selecciona una caja al azar
         do {
             cajaObjetivo = intRandom(0, nCajas);
             // mientras el objeto no quepa en la
seleccionada
         } while (ps[i] > cajas[cajaObjetivo]);
         // Resta a la capacidad de la caja el peso del
objeto
         cajas[cajaObjetivo] -= ps[i];
     }
     // Devuelve el numero de cajas utilizadas
     return nCajas + 1;
 }
```

Al igual que en la implementación anterior, se declara un vector con el número máximo de cajas que se pueden usar y una variable *nCajas* para indexar y controlar las que se usan. Esta variable contiene el índice de la última caja utilizada, es decir, si hay 2 cajas, el valor de *nCajas* será 1. También se declara una variable que almacena el valor de la caja en la que se va a intentar introducir el objeto. Se comienza asignando la capacidad inicial a todas las cajas.

Para cada objeto comprobamos si cabe en la última caja usada (empezamos con una caja y en la primera iteración comprueba si cabe en la única que hay). En el caso de que no haya capacidad suficiente, se usa una más incrementando *nCajas*. A continuación, se selecciona una caja objetivo al azar y se comprueba si el objeto cabe en ella. Se seguirá iterando en el bucle hasta que se cumpla la condición. La comprobación de la sentencia *if* asegura de que siempre habrá una caja en la que se pueda introducir. Por último, restamos el peso del objeto a la capacidad de la caja en la que lo introducimos. El método devuelve el número de cajas más uno.

Para comprobar si esta implementación realmente es una mejora, se ha experimentado con la optimalidad de este. El identificador del algoritmo es *menosCajasProbMejora* y su funcionamiento consiste en consultar si el objeto a introducir cabe en la última caja usada. Si es así, escoge cajas aleatoriamente hasta buscar una en la que quepa. Si no es así, antes de buscar hace uso de una nueva caja.

Utilizando los mismos valores que en el apartado anterior se obtienen los siguientes resultados:

Medidas	menosCajasAprox1	menosCajasAprox2	menosCajasBack	menosCajasProb	menosCajasProbMejora
Núm. ejecuciones	1000	1000	1000	1000	1000
Núm. ejec. válidas del método	1000	1000	1000	1000	1000
Núm. ejec. válidas en total	1000	1000	1000	1000	1000
% soluciones subóptimas	23.40 %	3.50 %	0 %	78.50 %	19.10 %
% soluciones óptimas	76.60 %	96.50 %	100 %	21.50 %	80.90 %
% soluciones sobreóptimas	0 %	0 %	0 %	0 %	0 %
% valor medio subóptimo	134.65 %	133.33 %	0 %	145.24 %	134.21 %
% valor extremo subóptimo	150 %	133.33 %	0 %	250 %	150 %
% valor medio sobreoptimo	0 %	0 %	0 %	0 %	0 %
% valor extremo sobreóptimo	0 %	0 %	0 %	0 %	0 %

Figure 3: Tabla de resultados numéricos

Como se puede ver en la tabla, esta segunda implementación resuelve el problema del valor extremo subóptimo del algoritmo probabilista 1 reduciendo el número a casi la mitad y obtiene un porcentaje de soluciones óptimas bastante alto (80%).

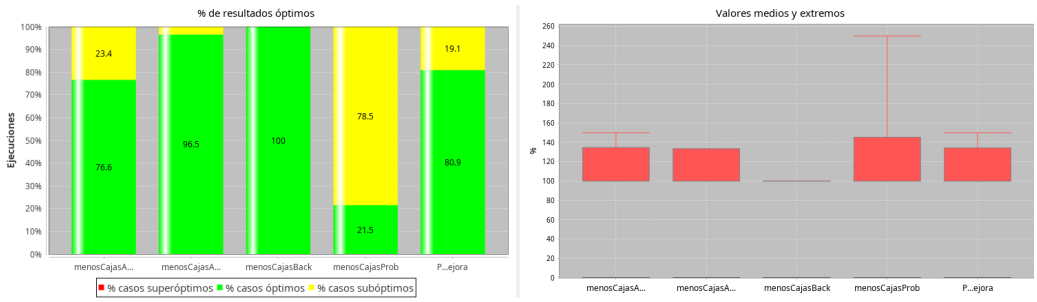

Figure 4: Gráficos

El nuevo algoritmo tampoco presenta soluciones sobreóptimas. Con él se obtiene un mayor porcentaje de soluciones óptimas y soluciones no tan extremas.

4 Conclusiones

Los algoritmos probabilistas han demostrado ser una buena opción dentro de los algoritmos aproximados debido a los buenos resultados que ofrecen con una complejidad tan baja. La parte más dificil de realizar fue la experimentación con Optimex porque realizaba una sola ejecución de los datos y daba fallos al exportar los gráficos. El error se resolvió al cambiar el nombre del método que provocaba el fallo.

La práctica ha sido entretenida y didáctica.