Teoría de la NP-Completitud

Complejidad Computacional

Escuela Superior de Ingeniería y Tecnología.
Ingeniería Informática.

Curso 2014/2015

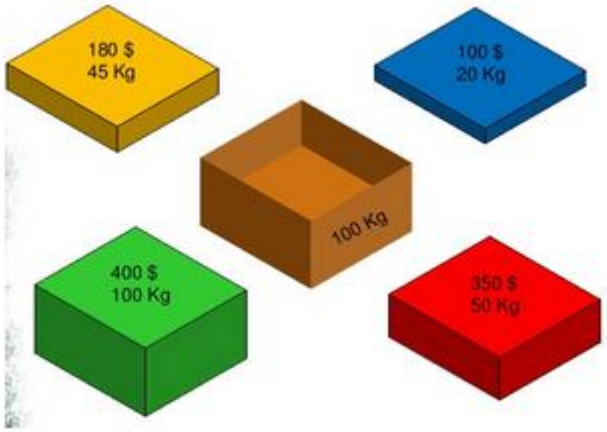
Iván García Campos. Alu4394

1.- Descripción del problema

Se modela una situación análoga al de llenar una **mochila**, incapaz de soportar más de un peso determinado, con todo o parte de un conjunto de objetos, cada uno con un peso y valor específicos. Los objetos colocados en la mochila deben maximizar el **valor** total sin exceder el **peso** máximo.

El problema de la mochila es definido formalmente como:

"Se tiene una determinada instancia de KP con un conjunto de objetos N, que consiste de n objetos j con ganancia p y peso w y una capacidad C, usualmente, los valores son tomados como números enteros positivos. El objetivo es seleccionar un subconjunto de N tal que la ganancia total de esos objetos seleccionados es maximizado y el total de los pesos no excede a c"



El problema de la mochila, comúnmente abreviado por KP (proveniente del inglés Knapsack Problem) es un problema de optimización combinatoria que busca la mejor solución entre un conjunto de posibles soluciones a un problema.
Este problema es uno de los **21 problemas NP completos de Richard Karp**, establecidos por el informático teórico en un famoso artículo de 1972.
En teoría de la complejidad computacional, NP es un acrónimo de nondeterministic polynomial time.
La importancia de esta clase de problemas de decisión es que contienen muchos problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las conocidas. En esta clase están por ejemplo el **viajante del comercio** donde se quiere saber si existe una ruta óptima que pasa por todos los nodos en un cierto grafo y el problema de satisfactibilidad booleana en donde se desea saber si una cierta fórmula de lógica proposicional puede ser cierta para algún conjunto de valores booleanos para las variables.

2.- Descripción del algoritmo voraz

También conocido como ávido, devorador es aquel algoritmo que para resolver un determinado problema sigue una heurística consistente en **elegir una opción óptima en cada paso local** con la esperanza de llegar a una solución general óptima.
Una forma de ver los algoritmos voraces es considerar la estrategia de vuelta atrás, en la cual se vuelve recursivamente a decisiones anteriormente tomadas para variar la elección entonces tomada, pero eliminando esa recursión y eligiendo la mejor opción.
Los algoritmos voraces tienden a ser **bastante eficientes y pueden implementarse de forma relativamente sencilla.**
Su eficiencia se deriva de la forma en que trata los datos, llegando a alcanzar muchas veces una complejidad de orden lineal. Sin embargo mayoría de los intentos de crear un algoritmo voraz correcto fallan a menos que exista previamente una prueba precisa que demuestre la correctitud del algoritmo.

Un algoritmo voraz funciona por pasos:

1. Inicialmente partimos de una solución vacía.,
2. En cada paso se escoge el siguiente elemento para añadir a la solución entre los candidatos.
3. Una vez tomada esta decisión no podrá deshacer.
4. El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución.

```
Cargando el fichero: hola2.txt

        CAPACIDAD:10
        NUM OBJETOS:5

OBJETOS: (PESO  VALOR)
        8       6
        5       5
        5       4
        7       1
        2       3


Tabla de pesos:  [0, 8, 5, 5, 7, 2, ]

Tabla de valores:  [0, 6, 5, 4, 1, 3, ]

Inicialización tabla vectorOrdenado
[7, 1.33333, 1.25, 1, 0.666667, 0, ]

 ******************RESULTADO***************
        Se eligio el peso : 7
        Se eligio el peso : 1.33333
        Se eligio el peso : 1.25

Podremos meter en la mochila: 9.58333
```

Dado los objetos ordenador por orden decreciente de peso, se introducen tantos como se pueda en una caja; cuando está esté llena ponemos cuantos sean posibles en la siguiente.

Otro ejemplo, pesos y valores aleatorios:

```
        CAPACIDAD:8

        NUM OBJETOS:12
Tabla de pesos: [0, 8, 12, 4, 6, 10, 4, 10, 14, 1, 14, 10, 1, ]
Tabla de valores: [0, 11, 2, 10, 6, 8, 7, 0, 14, 14, 5, 8, 3, ]

Inicialización tabla vectorOrdenado
[inf, 6, 2.8, 1.25, 1.25, 1, 1, 0.727273, 0.571429, 0.4, 0.333333, 0.0714286, 0, ]


 ******************RESULTADO*****************
        Se eligio el peso : 6
        Se eligio el peso : 1.25
        Se eligio el peso : 0.727273

Podremos meter en la mochila: 7.97727
```

Para ordenar los objetos se requiere un tiempo de ejecución O(nlogn)

3.- Descripción del algoritmo basado en programación dinámica

La idea principal de la **programación dinámica** consiste en intentar reducir el problema de n variables en una secuencia de problemas de una sola variable.
Describiendo el algoritmo de la mochila por programación dinámica se establece que:

Sea n objetos no fraccionables de peso pi y beneficios wi. El peso máximo que puede llevar la mochila es C. Queremos llenar la mochila con objetos, tal que se maximice el beneficio. De este modo, se deberá:

- Ver que se cumple el principio de optimalidad de Bellman.
- Buscar ecuaciones recurrentes para el problema.
- Construir una tabla de valores a partir de las ecuaciones.

La tabla de inicializa de la siguiente forma:

- T[0, j] = 0 para todo 0 ≤ j ≤W.
- T[i, 0] = 0 para todo 0 ≤ i ≤n.

El resto de la tabla se rellena por filas de acuerdo a la siguiente fórmula:

T[i,j] = T[i-1,j] si wi > j
y
T[i,j] = max {T[i-1,j], T[i-1, j-wi] + vi} si wi ≤ j

En el caso del algoritmo dinámico no se ha usado un vector dinámico para dividir pesos y valores. **El resultado de la tabla da los valores máximos a los que se puede optar.**

Los algoritmos de programación dinámica regularmente **utilizan más recursos de memoria**. En este caso, he usado también la técnica **divide y vencerás** para descomponer el problema original en varios subproblemas más sencillos para luego resolver estos mediante un cálculo sencillo. De esta forma aumentamos la eficiencia del algoritmo.

El algoritmo dinámico es un problema débilmente NP-hard admitiendo soluciones pseudo-polinomiales con tiempo de ejecución **O(n*c).**

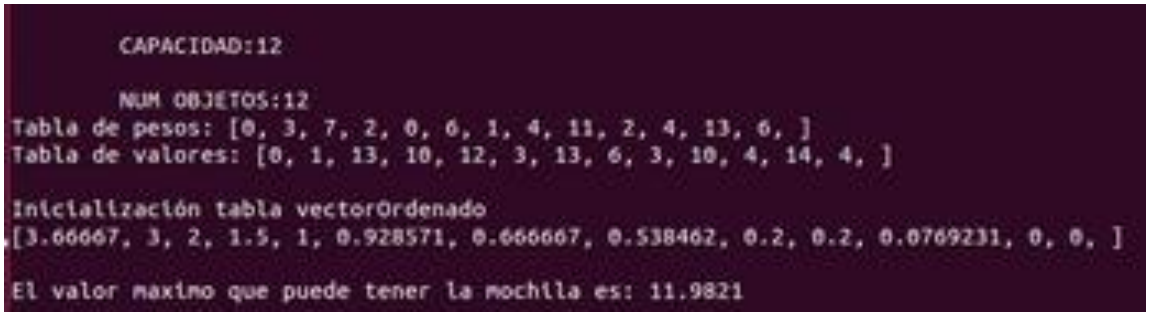
4.- Descripción del algoritmo basado en ramificación-acotación.

El método **Ramificación y poda** es un método de búsqueda de soluciones exhaustiva sobre grafos dirigidos, el cual se acelera mediante poda de ramas poco prometedoras.

En la práctica, ordenamos los pesos/valores como en la estrategia Greedy y evaluamos en cada iteración si se elige o no como componente de la solución.
Si no se eligiera, este podrá seguir evaluándose más adelante, al contrario que la estrategia Greedy.
Si se elige, se añade a **un vector dinámico** para evaluarlo con el siguiente nodo a evaluar. La estrategia podría codificarse como un **árbol de decisión.**
Además, el problema se podrá acotar eliminando aquellas combinaciones que superen el valor objetivo. De esta forma, se implementa el método mediante un vector dinámico porque el tamaño raramente será 2numElement.

Esta técnica logra una solución óptima del problema pero también utiliza una gran cantidad de recursos para calcularlo, teniendo **ejecución razonable solo para pocos datos.**

Otro ejemplo, pesos y valores aleatorios:



El tiempo de ejecución dependerá del tamaño de la entrada, de la calidad del código generado, de la rapidez de las instrucciones de la máquina y de la complejidad de tiempo del algoritmo.

5.- Descripción de la batería de problemas de prueba

En nuestro caso, mediremos la eficiencia en términos de tiempo de ejecución y de cercanía al valor objetivo.

Consideraciones:

1. El lenguaje de programación utilizado ha sido C++ ya que ofrece todas las características necesarias para poder implementar este tipo de estudios. Tiene librería para la creación de números aleatorios, posibilidad de cuantificar el tiempo de ejecución de funciones en nanosegundos y paralización de ejecución del programa para el cambio de semilla. Además, es un lenguaje bastante legible y utilizado en la mayoría de asignaturas de la carrera.

2. El programa es capaz de leer desde fichero diferentes problemas o generar instancias aleatorias.
   En el caso de generar una batería de pruebas aleatorias para realizar el estudio es necesario tener en cuenta que se usa la **función rand().**
   Esta función genera números aleatorios distintos **cambiando su semilla de generación pasado 1 segundo**. Si siguiéramos la ejecución del programa de forma normal, todas las pruebas se harían sobre la misma batería de datos por lo que es necesario hacer uso de la **función delay()** que retrasa la ejecución de las siguientes líneas de código los segundos que le pases por parámetro.

3. Por otro lado, el tiempo de ejecución de los algoritmos es del orden de **nanosegundos** por lo que para medir su ejecución tendremos que usar alguna función que calcule el tiempo del sistema en esa medida. Para ello, en la **librería <time.h>** existe un Struct.

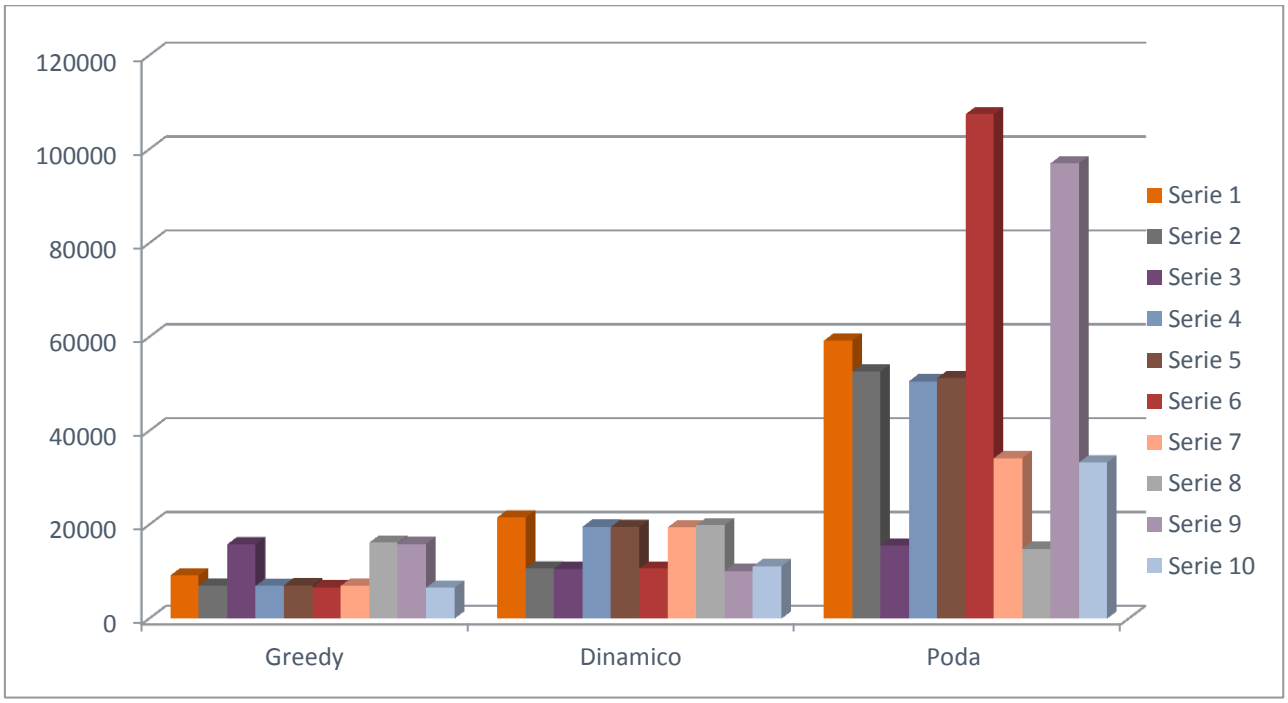
```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
};
```

En el siguiente código vemos como almacenamos los tiempos de ejecución de los algoritmos en los vectores correspondientes. Además, hacemos uso de la función delay() para cambiar los valores utilizados en la siguiente ejecución del for.

```
timespec b;
clock_gettime(CLOCK_REALTIME, &b);
vectordina[i] = (b.tv_nsec-ts.tv_nsec);
cout << "Dina " << vectordina[i] << endl;

timespec h;
clock_gettime(CLOCK_REALTIME, &h);

mochilaGreedy B(12, 12, -1);
B.estudiomochila(B.get_tamMochila(),0, 0);
B.~mochilaGreedy();

timespec j;
clock_gettime(CLOCK_REALTIME, &j);
vectorgre[i] = (j.tv_nsec-h.tv_nsec);
cout << "Greedy " << vectorgre[i] << endl;

timespec z;
clock_gettime(CLOCK_REALTIME, &z);

 mochila_Ramificada G(12, 12, -1);
  G.mochilaram();
  G.~mochila_Ramificada();

  timespec p;
 clock_gettime(CLOCK_REALTIME, &p);
 vectorpoda[i] = (p.tv_nsec-z.tv_nsec);
 cout << "Poda " << vectorpoda[i] << endl;

 delay(2);
```

6.- Gráfica comparativa en tiempos de ejecución de los tres algoritmos

Resultados ejecución para 10:



Greedy	9213	7007	15837	7002	7116	6661	7019	16212	15925	6592
Dinámico	21558	10682	10610	19575	19520	10667	19476	19891	10167	11142
Poda	59260	52697	15578	50599	51270	107495	34204	14853	97026	33276

6.1.- Interpretación de los resultados

En tiempos de ejecución, queda demostrado que el algoritmo Greedy es el que consume menos recursos seguido del algoritmo dinámico.
Ambos, ofrecen ejecuciones cuyos recursos suelen ser constantes.

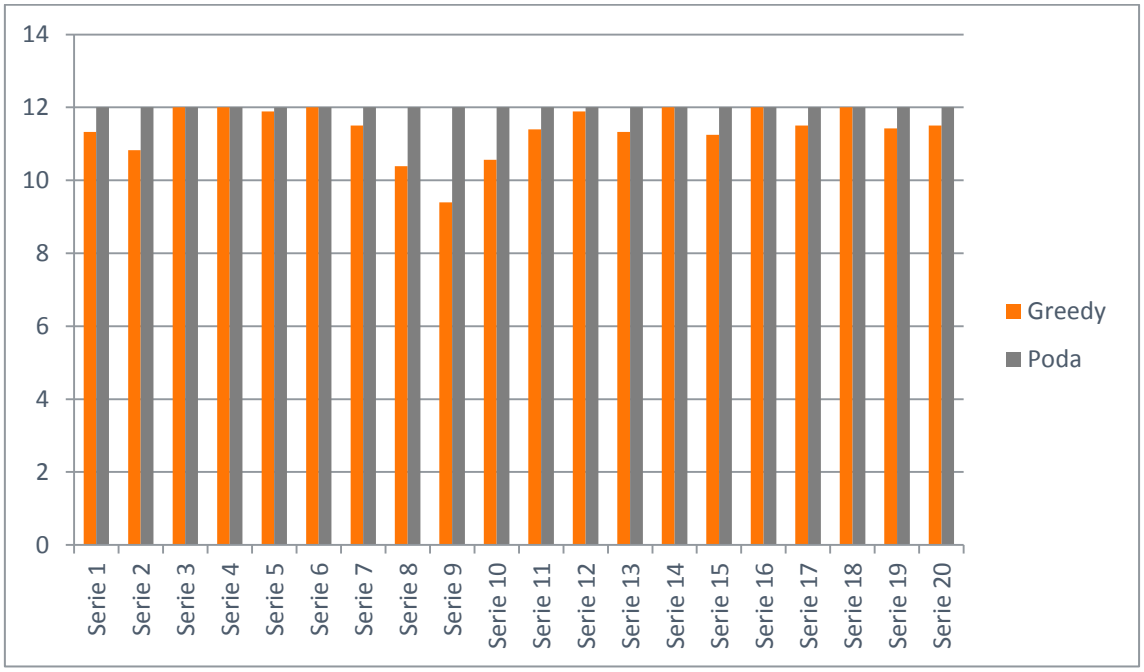
Sin embargo, analizando el algoritmo de Ramificación y Poda vemos que depende muchísimo de los datos de entrada para buscar la solución óptima del problema. Es un algoritmo bastante inestable con los recursos que necesita pero ofrece la mejor solución.

Si existiera una solución a este tipo de problemas NP-Completo, estas aproximaciones no serían necesarias. Será decisión del programador elegir la mejor estrategia a desarrollar para resolver este problema de forma aproximada.

7.- Gráfica comparativa en los valores objetivo

Comparación algoritmo Greedy y Ramificación.

Objetivo: 12.
Entrada aleatoria.




```
Introduzca el numero de elementos con el que quiere hacer el estudio
20

Indique ahora el peso maximo de la mochila
12
        [Greedy --      Poda y Ramificacion]

        11.3333 --      12
        10.8333 --      12
        12      --      12
        12      --      12
        11.8889 --      11.9936
        12      --      12
        11.5    --      12
        10.3905 --      12
        9.4     --      12
        10.5615 --      11.9997
        11.4    --      12
        11.8889 --      12
        11.3333 --      12
        12      --      12
        11.25   --      12
        12      --      12
        11.5    --      12
        12      --      12
        11.4231 --      12
        11.5    --      12

La diferencia media entre los dos algoritmos es de: 0.589523
```

No se ha comparado el algoritmo dinámico con los otros dos ya que la **estrategia Greedy y Ramificación y Poda utilizan pesos/valor** y el algoritmo dinámico usa por separado los pesos y los valores. De esta forma, no se pueden comparar los valores objetivos de esos dos algoritmos con el dinámico.

7.1.- Interpretación de los resultados

Queda demostrado que el algoritmo de Ramificación y Poda ofrece mejores resultados que el algoritmo Greedy. Sin embargo, este también logra una aproximación razonable al objetivo.

8. - Conclusión:

- Si bien, se encuentra una solución buenas con la ayuda de algoritmos voraces, esto solo es posible si la cantidad de elementos que se evalúan son pocos ya que en otro caso las soluciones son más dispersas con respecto al objetivo. Con respecto a la implementación, la cantidad de recursos utilizados es bastante bajo resultando muy útil en cualquier caso.
- En el caso del algoritmo dinámico, este utiliza más recursos que la estrategia Greedy y su estrategia se basa en optimizar el valor máximo de la mochila. Siendo útil solo en algunos casos.
- Nos damos cuenta que el uso de Ramificación y Poda nos proporciona las soluciones más óptimas pero su complejidad depende de la entrada del problema utilizando una cantidad de recursos desproporcionada comparado con las otras dos estrategias. Esto hace, que esta heurística sea recomendable solo para problemas reducidos o cuando sea necesaria una solución lo más óptima posible.

Finalmente, tenemos que si se encuentra alguna forma de desarrollar un algoritmo para resolver los problemas de tipo NP-completo se habrá hecho un gran hallazgo en el campo de las Ciencias de la Computación. Además de resolver este algoritmo, permitiría también resolver otros problemas de tipo NP-Completo de acuerdo a su definición.

9. – Códigos de los distintos algoritmos.

Código del algoritmo voraz para resolver el problema de la mochila:

```
double mochilaGreedy::estudiomochila(int size, int conta, int c)
{
   int contador = conta;
   int tamanyo =size;
   double resultado = 0.0;

    while ((contador< numElementos) && (tamanyo>0))
    {
          if (get_vectorOrdenado(contador) <= tamanyo)
          {    //cout << "\tSe eligio el peso : " << get_vectorOrdenado(contador) << endl;
               tamanyo = tamanyo - get_vectorOrdenado(contador);
               resultado += get_vectorOrdenado(contador);
               contador++;

          }else{

              contador++;

          }
    }

      return resultado;
}
```

Código del algoritmo Dinámico para resolver el problema de la mochila:

```
void mochilaD::recurrencia(int a, int b){//En la llamada b equivale a tamMochila- a numElementos
    if(a>0){
        if(b<pesoItems[a]){ //El tamaño es menor que la capacidad, llamamos a un peso menor.
            recurrencia(a-1,b);
        }else{
            if((matriz[a-1][b-pesoItems[a]]+ valorItems[a]) > matriz[a-1][b])
            {
                recurrencia(a-1,b-pesoItems[a]);
                //cout << "Objeto de peso = " << pesoItems[a] << " con valor = " << valorItems[a] << endl;
                totalItems++;
            }
            else
                recurrencia(a-1,b);
        }
    }
}

int  mochilaD::queItemsHay()
{
    recurrencia(numElementos, tamMochila);
    return totalItems;
}

double mochilaD::resolverestudio(){
  construirMatriz();
  return matriz[filas-1][columnas-1];
}
double mochilaD::resolver(){
     construirMatriz();
     cout<<"//////////////MATRIZ//////////////"<<endl<<endl;
     mostrarMatriz();
     cout<<endl;

     cout<<"Total de objetos:";
     cout << queItemsHay() <<endl<<endl;
     cout<<"El valor maximo de la mochila sera: "<<matriz[filas-1][columnas-1]<<endl;
     return matriz[filas-1][columnas-1];
}
```

Código del algoritmo de Ramificación y Poda:

```
double mochila_Ramificada::mochilaram()
{
  //vector dinamico de tam 0 < v.size <= 2^n
  vector<double> v;

  double aux = 0.0;
  double aux2 = 0.0;
  int a= 0;

  for (int i= 0; i<numElementos; i++)
  {

    for (int k= 0; k<=1; k++)
    {
      if (k==1)
        if (v.size() == 0){
          if (get_vectorOrdenado(i) <= get_tamMochila())
          {     v.push_back(get_vectorOrdenado(i));

          }
        }else{
          a = v.size();
          for (int l= 0; l< a; l++)
          {
            aux = get_vectorOrdenado(i) + v[l];
            if(aux<= get_tamMochila())
              v.push_back(aux);

          }
        }
    }
  }
  for (int i=0; i< v.size(); i++)
    if ((v[i] > aux2) && (v[i] <= get_tamMochila()))
      aux2 = v[i];
  return aux2;

}
```