

Daniel Asensi Roch, 48776120C

# **Ramificación y Poda**

Práctica Final Análisis y Diseño de Algoritmos

Daniel Asensi Roch

Jose Oncina

27 de mayo de 2021

1. Estructura de datos
  1. 1Nodo
  2. Lista de nodos vivo
2. Mecanismos de poda
  1. Poda de nodos no factibles
  2. Poda de nodos no prometedores
3. Cota Pesimista y Optimista
  1. Cota pesimista inicial (inicialización)
  2. Cota pesimista del resto de nodos
4. Otros medios empleados para acelerar la búsqueda
5. Estudio comparativo de distintas estrategias de búsqueda
6. Tiempos de ejecución

# 1. Estructura de datos

## 1.1 Nodo

En este apartado se describirá el contenido de los nodos, variables que lo componen y cometido de las mismas, pero antes de realizar estas explicaciones he de recalcar la manera utilizada para almacenar los datos de este problema propuesto.

La manera elegida para almacenar los datos ha sido en forma de estructura, la que nos proporcionará una forma unificada de almacenar los datos del problema y más adelante trabajar con los mismos de manera más cómoda, esa estructura es la siguiente:

```
struct TDatos
{
    vector<long> m; //Maximas copias realizables
    vector<long> v; //Precio
    vector<double> t; //tiempo de realización (peso)
    double T; //tiempo maximo
    long n; //Cantidad objetos
};
```

Los componentes de los nodos utilizados en la práctica son los siguientes:

```
typedef tuple<double, double, double, soluciones, unsigned> Nodo;
```

Como podemos apreciar la estructura de los Nodos son tuplas que más adelante almacenaremos en una cola de nodos, el primer valor double es una de las soluciones de la "Mochila Voraz Continua" es decir un valor optimista de la solución, el siguiente double es el valor dado por el alfarero a ese objeto, el tercer double es el tiempo empleado por el alfarero, la componente "soluciones" es un vector de long para almacenar las posibles soluciones par el nodo, el valor unsigned es la iteración por la que nos encontramos.

## 1.2 Lista de nodos vivos

La lista de nodos vivos viene representada por una cola compuesta de nodos con la siguiente sintaxis:

```
priority_queue<Nodo> colaNodos;
```

Esta lista de nodos se inicializará de la siguiente manera para poder tener un nodo con el que comparar el resto de los mismos.

```
double nodoInicial = vorazMochilaContinua(datosOrdenados, 0, datosOrdenados.T);  
  
//Metemos el nodo inicial en la cola de prioridad de nodo  
colaNodos.emplace(nodoInicial, 0, 0, soluciones(datosOrdenados.n), 0);
```

El nodo inicial obtendrá utilizando el algoritmo voraz de la mochila continua, que aunque no nos ofrezca una solución óptima si que acelera la búsqueda de los nodos obteniendo un buen valor con el que comparar el resto de los nodos. Esta ha sido implementada de la siguiente manera:

```
double vorazMochilaContinua(const TDatos &datos, long iteracion, double tiempo)  
{  
    double solucion = 0;  
    double tiempoQueda = tiempo;  
    unsigned long i;  
  
    for (i = iteracion; i < datos.n; i++)  
    {  
        if (tiempoQueda > 0)  
        {  
            //Si todavía podemos ir cogiendo nodos continuamos  
            if (datos.t[i] * datos.m[i] <= tiempoQueda)  
            {  
                solucion += datos.v[i] * datos.m[i];  
                tiempoQueda -= datos.t[i] * datos.m[i];  
            }  
            else //No nos queda tiempo para meter más objetos  
            {  
                solucion += datos.v[i] * tiempoQueda / datos.t[i];  
                tiempoQueda = 0;  
            }  
        }  
    }  
    return solucion;  
}
```

La estructura de nuestra cola previamente explicada es la siguiente:



```
//Cabeza de la cola  
auto [optimista, valor, tiempo, x, k] = colaNodos.top();
```

## 2. Mecanismos de poda

### 2.1 Poda de nodos no factibles

Los nodos no factibles son descartados cuando el valor optimista del nodo es menor que el mejor valor obtenido hasta el momento durante el algoritmo, el primer valor a comparar es obtenido utilizando el algoritmo de la mochila discreta,(que explicaré en el siguiente apartado) esta inicialización se realiza con los datos ya ordenados.

```
double mejorValor = vorazMochilaDiscreta(datosOrdenados, 0, datosOrdenados.T);
```

La poda con los valores obtenidos se realiza de la siguiente manera:

```
if (optimista < mejorValor)
{
    descartados++;
    continue; //Pasamos al siguiente nodo
}
```

Al haber descartado el nodo pasamos al siguiente de la cola.

## 2.2 Poda de nodos no prometedores

Los nodos no prometedores de este mi algoritmo se realiza al ver que el valor optimista es menor que el mejor valor obtenido:

```
if (optimista > mejorValor)
    colaNodos.emplace(optimista, nuevoValor, nuevoTiempo, x, k + 1);
else
    noPrometedores++; // En caso contrario la rama explorada no es prometedora y se descarta
```

Esto se debe a que los valores optimista de solución de cada nodo serán los que nos indicarán si debemos analizar o no dicho nodo, si este valor es mayor procederemos a encolarlo para su posterior análisis completo.

## 3. Cotas pesimistas y optimistas

### 3.1 Cota pesimista inicial

Cómo hemos visto en clase la eficiencia de la poda viene determinada en gran medida por la calidad de las soluciones provisionales encontradas (y almacenadas) durante la ejecución del algoritmo. Lo que mejora en gran medida cuando el algoritmo (como es el caso) es muy extenso. Por lo que llamaremos cota pesimista  $h(X)$  a cualquier función sencilla que calcula el valor de la función objetivo  $\varphi$  para una solución  $\hat{x}$ , no necesariamente óptima, de  $X$ .

En este caso particular realizaremos la primera poda con el algoritmo de la mochila discreta dándonos este un valor optimo, pero no el más optimo de todos, para empezar a podar nodos, este ha sido implementado de la siguiente manera:

```
double vorazMochilaDiscreta(const TDatos &datos, long iteracion, double tiempo)
{
    double solucion = 0;
    double tiempoQueda = tiempo;
    unsigned long i;
    long caben = 0;

    for (i = iteracion; i < datos.n; i++)
    {
        if (tiempoQueda > 0)
        {
            if (datos.m[i] <= trunc(tiempoQueda / datos.t[i]))
                caben = datosOrdenados.m[i];
            else
                caben = trunc(tiempoQueda / datosOrdenados.t[i]);
            solucion += datosOrdenados.v[i] * caben;
            tiempoQueda -= datosOrdenados.t[i] * caben;
        }
    }
    return solucion;
}
```



## 3.2 Cota pesimista del resto de nodo

Realizaremos la cota pesimista del resto de nodos, inicializando el primer nodo mediante el algoritmo de la mochila continua previamente explicado en la inicialización de los nodos.

## 3.3 Cota Optimista

```
//Pasamos ahora analizar las maximas
for (unsigned long int j = 0; j <= datosOrdenados.m[k]; j++)
{
    visitados++;
    x[k] = j;
    //Adjudicamos los nuevos tiempos y nuevos valores que luego comparearemos
    double nuevoTiempo = tiempo + x[k] * datosOrdenados.t[k];
    double nuevoValor = valor + x[k] * datosOrdenados.v[k];

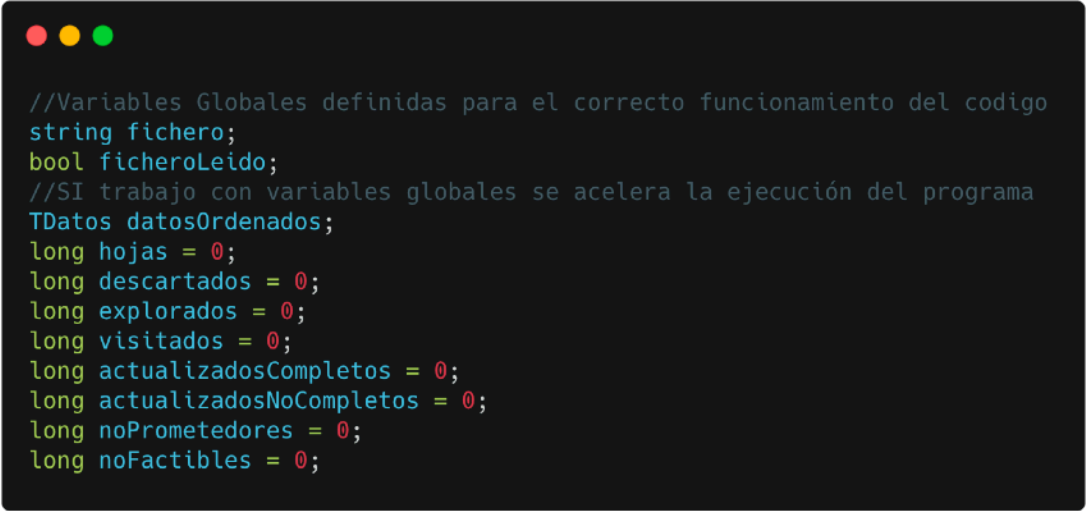
    if (nuevoTiempo <= datosOrdenados.T) //El nodo es factible
    {
        //Ahora probamos a mejorar el ultimo valor que teniamos
        //Mediante el uso de la mochila Discreta
        double valorViejo = mejorValor;
        mejorValor = max(mejorValor, nuevoValor +
        vorazMochilaDiscreta(datosOrdenados, k + 1, datosOrdenados.T - nuevoTiempo));
        //Si el nuevo valor es diferente debemos incrementar los actualizados
        completos y no completos cuando sea k = al número objetos -1
        if (valorViejo != mejorValor)
        {
            if (k == datosOrdenados.n - 1)
                actualizadosCompletos++;
            else
                actualizadosNoCompletos++;
        }
        //Conseguimos el nuevo valor Optimista que luego compararemos
        optimista = nuevoValor + vorazMochilaContinua(datosOrdenados, k + 1,
        datosOrdenados.T - nuevoTiempo);
        //Si este es mejor que el mejorValor meteremos el nodo en la cola
        //SI VEMOS QUE EL NODO PUEDE CONTENER LA MEJOR SOLUCION LO METEMOS EN LA
        COLA!!

        if (optimista > mejorValor)
            colaNodos.emplace(optimista, nuevoValor, nuevoTiempo, x, k + 1);
        else
            noPrometedores++; //En caso contrario la rama explorada no es
            prometedora y se descarta
        }
        else
        {
            noFactibles++;
        }
    }
}
```

La cota optimista de los nodos es realizada sumando el nuevo valor previamente declarada utilizando la suma de los valores del nodo más la posición en su propio vector de soluciones por su tiempo empleado. Esta cota optimista será utilizada para podar los nodos no prometedores.

## 4.Otros medios empleados para acelerar la búsqueda

El único método empleado para acelerar la búsqueda de la mejor solución ha sido el declarar los datos ordenados del problema como variables globales ,ya que eso acelera la ejecución del programa.



```
//Variables Globales definidas para el correcto funcionamiento del codigo
string fichero;
bool ficheroLeido;
//SI trabajo con variables globales se acelera la ejecución del programa
TDatos datosOrdenados;
long hojas = 0;
long descartados = 0;
long explorados = 0;
long visitados = 0;
long actualizadosCompleto = 0;
long actualizadosNoCompleto = 0;
long noPrometedores = 0;
long noFactibles = 0;
```

Uno de los mecanismos que se podrían haber empleado par acelerar la búsqueda de la mejor solución es ordenar la cola de prioridad por el mejor valor.

## 5. Estudio comparativo de distintas estrategias de búsqueda

En las estrategias utilizadas para aminorar la cantidad de nodos explorados y aumentar la cantidad de nodos podados para obtener la mejor solución es el uso de los algoritmos voraces de las mochilas discretas y continuas.

```
//Adjudicamos los nuevos tiempos y nuevos valores que luego comparearemos
double nuevoTiempo = tiempo + x[k] * datosOrdenados.t[k];
double nuevoValor = valor + x[k] * datosOrdenados.v[k];

if (nuevoTiempo <= datosOrdenados.T) //El nodo es factible
{
    //Ahora probamos a mejorar el ultimo valor que teniamos
    //Mediante el uso de la mochila Discreta
    double valorViejo = mejorValor;
    mejorValor = max(mejorValor, nuevoValor +
    vorazMochilaDiscreta(datosOrdenados, k + 1, datosOrdenados.T - nuevoTiempo));
    //Si el nuevo valor es diferente debemos incrementar los actualizados
    completos y no completos cuando sea k = al número objetos -1
    if (valorViejo != mejorValor)
    {
        if (k == datosOrdenados.n - 1)
            actualizadosCompletos++;
        else
            actualizadosNoCompletos++;
    }
    //Conseguimos el nuevo valor Optimista que luego comparearemos
    optimista = nuevoValor + vorazMochilaContinua(datosOrdenados, k + 1,
    datosOrdenados.T - nuevoTiempo);
```

La estrategia empleada ha sido actualizar los valores mejores y optimistas a medida que avanzamos a lo largo de los nodos recortaremos los que no optimistas viendo si los nodos son óptimos para ser analizados o debemos saltarlos realizando el recorte, realizaremos lo mismo con el mejor el valor de obtenido hasta el momento.

```
optimista = nuevoValor + vorazMochilaContinua(datosOrdenados, k + 1,
datosOrdenados.T - nuevoTiempo);
//Si este es mejor que el mejorValor meteremos el nodo en la cola
//SI VEMOS QUE EL NODO PUEDE CONTENER LA MEJOR SOLUCION LO METEMOS EN LA
COLA!!

if (optimista > mejorValor)
    colaNodos.emplace(optimista, nuevoValor, nuevoTiempo, x, k + 1);
else
    noPrometedores++; //En caso contrario la rama explorada no es
prometedora y se descarta
```

Promedio del número de iteraciones para 100 instancias aleatorias del problema de la mochila con 100 objetos

	<b>Optimista</b>	<b>Inicializado</b>	<b>Pesimista</b>
<b>Vuelta Atrás</b>	45331ms	40122ms	40111ms
<b>RyP (por Valor)</b>	51848ms	49918ms	44124ms
<b>RyP (por cota optimista)</b>	53595 ms	60116 ms	54511ms

## 6. Tiempos de Ejecución

Fichero potter\_n15.def: 0,521 ms

Fichero potter\_n20.def: 0,298 ms

Fichero potter\_n30.def: 56,129 ms

Fichero potter\_n40.def: 0,926 ms

Fichero potter\_n50.def: 402,143 ms

Fichero potter\_n60.def: 4,356 ms

Fichero potter\_n70.def: 59670,8 ms

Fichero potter\_n80.def: 83190,8 ms

Fichero potter\_n90.def: ?

Fichero potter\_n100.def: ?