3. Algoritmos Voraces (Greedy)

3.1 Introducción

Los algoritmos voraces, también conocidos como algoritmos Greedy o de avance rápido, se usan normalmente en problemas de optimización. Por ejemplo: obtener el mejor beneficio, el menor coste, etc. El planteamiento básico de estos algoritmos es tomar algunos elementos de entre un conjunto de candidatos con objeto de optimizar una función objetivo. El orden en el que se toman los elementos puede ser importante o no en el sentido de que nos conduzca, un determinado orden de selección al óptimo, o de igual el orden para alcanzar el óptimo.

Como ejemplo de un algoritmo voraz que aplicamos cuando vamos a la frutería es el siguiente: Supongamos que queremos comprar 2 kg de patatas. Los pasos generales que llevaría a cabo un algoritmo voraz sería:

1. *Inicialmente partimos de una solución vacía.* Nuestra bolsa de la compra no tiene ninguna patata.
2. *Tenemos una función de selección, que escoge el siguiente elemento a añadir a nuestra solución entre los candidatos*. En nuestra tarea de comprar patatas, seleccionamos a ojo de buen cubero la patata mejor del montón.
3. *Un vez escogido un elemento no podremos devolverlo, no se deshace la elección.* En nuestro caso examinamos la patata y decidimos si la echamos a la bolsa o no. Si no se coge se aparta del montón. Si se coge se mete en la bolsa y ya no se saca.
4. *El algoritmo acaba cuando el conjunto de elementos seleccionados constituye una solución.* Cuando tengamos los 2kg de patatas paramos.

3.2 Funciones Genéricas y Método General

Los elementos necesarios en un algoritmo voraz son:

- **C**. Conjunto de candidatos a seleccionar
- **S**: Candidatos seleccionados para la solución

La función genérica voraz sería:

Funciones Genéricas

- *Solucion(S)*.- Comprueba si un conjunto S es solución. Esta comprobación es independiente de que sea óptima o no.
- *Seleccionar(C)*.-Devuelve el elemento más prometedor de C.
- *Factible(S,X)*.-Al añadir a S X se crea un nuevo conjunto, y Factible indica si a partir de este nuevo conjunto se puede obtener una solución.
- *Objetivo(S)*.-Dada una solución devuelve el coste asociado . Normalmente la función objetivo esta enmascarada en la disposición inicial de los elementos de C o se incrusta dentro de la función Seleccionar. Esta función es útil en problemas de optimización.

3.3 Problema: El cambio de Monedas

Dado un conjunto de monedas y una cantidad P se debe devolver esa cantidad usando el menor número posible de monedas. Además supongamos que tenemos de cada tipo de moneda una cantidad ilimitada. Por ejemplo: Supongamos que disponemos de monedas 1,2,5,10, 20 50 céntimos de euro, 1 y 2 euros. Si $P = 3{,}89$ euros, el conjunto de monedas que se devuelven sería:

Moneda	Numero de monedas	Cantidad restante
2 euros	1	1.89
1 euro	1	0.89
50 céntimos	1	0.39
20 céntimos	1	0.19
10 céntimos	1	0.09
5 céntimos	1	0.04
2 céntimos	2	0

A continuación vamos a plantear un algoritmo Voraz para resolver este problema.

Elementos del Algoritmo Voraz

- *Conjunto de Candidatos, C.*- todos los tipos de monedas en una cantidad ilimitada. c_i tipo i-th de moneda
- *Solución, S.*- Conjunto de monedas que sumen P.
- *Función objetivo.*- minimizar el número de monedas
- *Representación de la solución.*- Un vector $(x_1, x_2, \cdots, x_n)$ con x_i siendo el número de monedas de tipo i. n es el número de monedas diferentes.
- *Formulación.*-

$$\text{minimizar}(\textstyle\sum_{i=1}^{n} x_i) \text{ sujeto a } \sum_{i=1}^{n} x_i \cdot c_i = P \text{ con } x_i \geq 0$$

Funciones del Algoritmo Voraz

1. Inicialización $\Longrightarrow x_i = 0 \quad \forall \quad i = 1$
2. Solución $\Longrightarrow$ El valor actual $((x_1, x_2, \cdots, x_n)$ si cumple que $\sum_{i=1}^{n} x_i \cdot c_i = P$
3. Seleccionar$\Longrightarrow$ Escoger la moneda de mayor valor posible, sin sobrepasar la cantidad que queda por devolver.
4. Factible$\Longrightarrow$ Siempre es verdad. Ya que la función *Seleccionar* escoge la moneda sin pasar la cantidad P.

NOTA 1.- En vez de coger una moneda en cada iteración, tomamos tantas de un tipo como la división entera de la cantidad restante por el valor de la moneda.
NOTA 2.- Vamos a usar una variable *act* para acumular la cantidad devuelta hasta el momento. Además vamos a suponer que las monedas nos las dan ordenadas de menor a mayor valor c_i.

Algoritmo Voraz

Con todas estas consideraciones ya podemos plantear el código del algoritmo.

```
1  /**
2   @brief Devuelve un cantidad con un conjunto de monedas.
3   @param P: cantidad a devolver
4   @param C: conjunto con los tipos de monedas. Ordendas de menor a mayor valor.
5   @param X: numero de monedas escogidas para cada tipo de moneda. ES MODIFICADO.
6   @return true si obtiene alcanza la cantidad P exacta. false en otro caso.
7  */
8  bool DevolverCambio(float P, const vector<float>&C,vector<int> &X){
9    int n= C.size(); // numero de monedas diferentes
10   //X tambien tiene ese tamanho
11   float act=0.0;
12   int j= n-1; //para empezar por la moneda de mayor valor
13   //ponemos todos los xi a 0
14   for (int i=0;i<n;i++)
15    X[i]=0;
16
17   while (act!=P ){//mientras no lleguemos a la cantidad
18     //buscamos la siguiente moneda factible
19     while (C[j]>P-act && j>=0)
20      j=j-1;
21     if (j<0){
22        cout<< ''No hay solucion''<<endl;
23        return false;
24     }
25     //cogemos tantas monedas cj como podamos
26     X[j] = floor((P-act)/C[j]); //entero menor o igual
27     act= act+X[j]*C[j];
28   }
29   return true;
```

```
30 }
```

Eficiencia

La eficiencia de este algoritmo se define en el peor caso por analizar todos los tipos de monedas, dando lugar a un eficiencia en el peor de los caso $\mathbf{O}(n)$.

Optimalidad

Para el sistema monetario que hemos escogido siempre vamos a obtener una solución aunque esta no sea la óptima. Si eliminamos las monedas de 1 céntimo. ¿siempre obtendríamos solución?. La contestación es no. Ya que escogiendo una cantidad P que no tenga una partición con el conjunto de tipos de monedas no llegaríamos a obtener solución.
Otra consideración es si siempre obtenemos la solución óptima, en el sentido de obtener el menor número de monedas. Por ejemplo supongamos que $P = 180$. Si seguimos con la estrategia de escoger aquel tipo de moneda de entre las que nos queda de mayor valor, y suponiendo que nuestros tipos tienen valor 100,90 y 1. Según este criterio tomaríamos 1 de 100 y 80 de 1. Pero la solución óptima sería coger 2 monedas de 90. Por lo tanto el criterio de selección no siempre da la solución óptima.

3.4 Eficiencia en términos generales

Los algoritmos voraces suelen tener una eficiencia polinomial. Y en términos generales podemos definirla como $T(n,m)$:

n: número de candidatos, m: número de elementos de la solución

$$T(n,m) = \mathbf{O}(n \cdot \underbrace{f(m)}_{\text{Eficiencia para ver si el valor actual es solución}} + \underbrace{g(n)}_{\text{Eficiencia de la Selección}} + \underbrace{h(m)}_{\text{Eficiencia de Factible}} + m \cdot \underbrace{j(n,m)}_{\text{Unión de un nuevo elemento a la solución}})$$

3.5 Problema: La Mochila

En este problema nuestro objetivo es llenar una mochila con una determinada capacidad de peso máximo M con objetos de un determinado peso p_i y beneficio b_i.

Objetivo.- Llenar la mochila con objetos, de entre los n, que al sumar los beneficios de los escogidos, este beneficio acumulado sea el mayor posible y no se supera la capacidad de la mochila.

En esta primera aproximación al problema de la Mochila, vamos a suponer que los objetos se pueden partir en trozos.

Datos del Problema

Los datos para representar el problema son:

- n: número de objetos disponibles
- M: capacidad de la mochila
- $p = (p_1, p_2, \cdots, p_n)$: pesos de los objetos
- $b = (b_1, b_2, \cdots, b_n)$: beneficios de los objetos.

Representación de la Solución

Una solución $S = (x_1, x_2, \cdots, x_n)$ con $0 \leq x_i \leq 1$ donde cada x_i representa el trozo escogido del objeto i. Así los casos extremos son:

- Si $x_i = 1$ se coge todo el objeto
- Si $x_i = 0$ no se coge el objeto
- Cualquier otro valor entre $(0,1)$ representa la fracción que se coge del objeto

Formulación Matemática

El objetivo del problema es :

$$\text{Maximizar} \sum_{i=1}^{n} x_i \cdot b_i \text{ sujeto a la restricción } \sum_{i=1}^{n} x_i \cdot p_i \leq M \text{ y } 0 \leq x_i \leq 1$$

Ejemplo

Supongamos que:

- El numero de objetos es $n = 3$
- La capacidad de la mochila $M = 20$
- Los pesos de los objetos son $p = (18, 15, 10)$
- Los beneficios de los objetos son $b = (25, 24, 15)$

Dependiendo del criterio de selección podemos obtener diferentes soluciones:

- *Solucion 1*.
  - $S = (1, \frac{2}{15}, 0)$
  - Beneficio total= $\sum_{i=1}^{3} x_i \cdot b_i = 1 \cdot 25 + \frac{2}{15} \cdot 24 + 0 \cdot 15 = 28{,}2$
  - Peso total = $\sum_{i=1}^{3} x_i \cdot p_i = 1 \cdot 18 + \frac{2}{15} \cdot 15 = 20$
- *Solucion 2*.
  - $S = (0, \frac{2}{3}, 1)$
  - Beneficio total= $\sum_{i=1}^{3} x_i \cdot b_i = 0 \cdot 25 + \frac{2}{3} \cdot 24 + 1 \cdot 15 = \boxed{31}$
  - Peso total = $\sum_{i=1}^{3} x_i \cdot p_i = 0 \cdot 18 + \frac{2}{3} \cdot 15 + 1 \cdot 10 = 20$

Diseño de la Solución

Los elementos y funciones para el problema de la mochila serían:

- Candidatos.- Todo los objetos de partida
- Función de Selección.- Escoger el objeto más prometedor$\Longleftarrow$ **¿Como definirlas?**.- Ideas Felices
- Función Factible.- Como podemos añadir trozos a la mochila, mientras que haya espacio siempre será true.
- Añadir a la solución.- Añadir el objeto si cabe en otro caso la proporción del mimos hasta completar la capacidad de la mochila.
- Función objetivo.- Suma los beneficios de cada candidato por la proporción seleccionada del mismo

Sin aún haber concretado la función de Selección, podemos dar una especificación del algoritmo de la Mochila como sigue:

```
1  /**
2    @brief Selecciona los trozo de un conjunto de objetos para
3           rellenar una mochila de capacidad M
4    @param M: capacidad en peso de la mochila
5    @param b: vector con los beneficios de cada objeto
6    @param p: vector con los pesos de cada objeto
7    @param X: fraccion escogida para cada objeto. Valores entre [0,1]
8    @return beneficio total obtenido.
9   */
10 float Mochila_Voraz(int M, const vector<int> &b, const vector<int >& p, vector <float> &
11   //Inicializacion
12   int n= p.size(); //numero de objetos
13   for (int i=0;i<n;i++)
14    X[i]=0;
15   int pesoAct=0;
16   flaot beneAct=0.0; //beneficio actual
17   while (pesoAct<M){
18     //seleccionamos el mejor objeto restante
19     int i= Seleccion(p,b);
20     if (pesoAct+p[i]<= M){
21        X[i]=1
```

```
22          pesoAct += p[i];
23          beneAct += b[i];
24      }
25      else {
26        // se coge la proporcion de p[i] para
27        //completar hasta M
28          X[i]=(M-pesoAct)/p[i];
29          pesoAct=M;
30          beneAct+= X[i]*b[i];
31      }
32   }
33   return beneAct;
34  }
```

¿Cómo definir la función de Selección?. Tenemos que establecer algún criterio que nos produzca la selección de objetos que mayor beneficio nos proporcione. Para el problema de la mochila podemos fijar tres criterios para seleccionar el objeto i^*:

1. El objeto con más beneficio $\Longrightarrow i^* = argmax_{i=1\cdots n}\ b_i$.
2. El objeto con menos peso $\Longrightarrow i^* = argmin_{i=1\cdots n}\ p_i$.
3. El objeto con mayor razón beneficio por peso $i^* = argmax_{i=1\cdots n}\ \frac{b_i}{p_i}$. Hay que tener en cuenta que la fracción $\frac{b_i}{p_i}$ nos da el beneficio por unidad de peso.

Ejemplo 3.5.1
Supongamos que $n = 4$, $M = 10$, el vector de pesos $p = (10,3,3,4)$, el vector de beneficios $b = (10,9,9,9)$ y la razón beneficio por peso $\frac{b}{p} = (1,3,3,2{,}25)$. Veamos cual sería el vector solución X y el beneficio total B_T.

- CRITERIO 1. MAYOR BENEFICIO $\Longrightarrow X = (1,0,0,0)$ $B_T = 10$
- CRITERIO 2. MENOR PESO $\Longrightarrow X = (0,1,1,1)$ $\boxed{\mathrm{B}_T = 27}$
- CRITERIO 3. MAYOR RAZÓN $\frac{b_i}{p_i} \Longrightarrow X = (0,1,1,1)$, $\boxed{\mathrm{B}_T = 27}$

□

Ejemplo 3.5.2
Supongamos que $n = 4$, $M = 10$, el vector de pesos $p = (10,3,3,4)$, el vector de beneficios $b = (10,1,1,1)$ y la razón beneficio por peso $\frac{b}{p} = (1,\frac{1}{3},\frac{1}{3},\frac{1}{4})$. Veamos cual sería el vector solución X y el beneficio total B_T.

- CRITERIO 1. MAYOR BENEFICIO $\Longrightarrow X = (1,0,0,0)$ $\boxed{\mathrm{B}_T = 10}$
- CRITERIO 2. MENOR PESO $\Longrightarrow X = (0,1,1,1)$, $B_T = 3$
- CRITERIO 3. MAYOR RAZÓN $\frac{b_i}{p_i} \Longrightarrow X = (1,0,0,0)$, $\boxed{\mathrm{B}_T = 10}$

□

Parece con los dos ejemplos anteriores que el criterio 3 garantiza que siempre encuentra la solución óptima. Veamos que realmente es así demostrándolo.

Propiedad. En el problema de la Mochila el criterio de selección basado en seleccionar el objeto con mayor razón beneficio por peso $\frac{b}{p}$ obtiene el mayor beneficio acumulado posible.
Demostración.- Supongamos que tenemos una solución óptima:

$$X = (x_1, x_2, \cdots, x_n)$$

que incluye un objeto i y otro j en menor proporción al objeto i.Lo que significa que $x_i > x_j$ y además se cumple que $\frac{b_i}{p_i} < \frac{b_j}{p_j} \Longleftarrow$ **Significa que hemos incluido más de i a pesar de tener una razón menor**. Con esta premisa tenemos que llegar a una contradicción. Para ellos supongamos que construimos otra solución quitando una parte del objeto i y sustituyendo esa parte por una de j. Sea esta cantidad r de tal forma que:

$$0 \leq r \leq x_i \cdot p_i \wedge \quad r \leq (1 - x_j) \cdot p_j \Longleftarrow r \text{ debe ser menor o igual de lo que nos queda por poner de } j$$

Con este cambio, calculamos el beneficio total de esta solución b_{nuevo}:

$$b_{nuevo} = b_{antiguo} - \underbrace{r \cdot \frac{b_i}{p_i}}_{\substack{\text{beneficio que quitamos} \\ \text{al quitar un trozo r de i}}} + \underbrace{r \cdot \frac{b_j}{p_j}}_{\substack{\text{beneficio que añadimos} \\ \text{al poner un trozo r de j}}}$$

$$= b_{antiguo} + r \cdot \left(\frac{b_j}{p_j} - \frac{b_i}{p_i} \right) > b_{antiguo}$$

siendo $b_{antiguo}$ el beneficio total de la solución de partida, X. Llegamos a una contradicción ya que la solución de la que partimos X, con beneficio $b_{antiguo}$, era optimal. Por lo tanto no puede darse que $\frac{b_i}{p_i} < \frac{b_j}{p_j}$, si X es optimal.

□

Eficiencia

Si el vector está ordenado por la razón de beneficio por peso entonces seleccionar un objeto es $\mathbf{O}(1)$. Por lo tanto la eficiencia del algoritmo *Mochila_Voraz* sería en el peor de los caso $\mathbf{O}(n)$.

Ejercicio 3.1

Suponed en el problema de la mochila que los objetos no se pueden partir. Dar el algoritmo y deducir si podemos dar siempre la solución óptima.
Solución.-

```
1  float Mochila_Voraz01(int M, const vector<int> &b, const vector<int >& p,
2                        vector <float> &X){
3    //Inicializacion
4    int n= p.size(); //numero de objetos
5    for (int i=0;i<n;i++)
```

```
    X[i]=0;
  int pesoAct=0;
  flaot beneAct=0.0; //beneficio actual
  while (pesoAct<M){
    //seleccionamos el mejor objeto restante
    int i= Seleccion(p,b);
    if (pesoAct+p[i]<= M){
       X[i]=1
       pesoAct += p[i];
       beneAct += b[i];
    }

 }
 return beneAct;
}
```

Como se puede observar al no poder partir los objetos no siempre va a llenar completamente la Mochila y por lo tanto el criterio de selección no siempre va a obtener la solución óptima. Por ejemplo suponed una mochila de capacidad $M = 4$ con tres objetos de pesos $p = (3,1,2)$, beneficios $b = (5,2,4)$, luego la razón es $\frac{b}{p} = (5/3,2,2)$. Siguiendo el criterio de escoger el objeto de mayor razón, escogeríamos el objeto 3 con peso 2 y beneficio 4, a continuación escogemos el segundo objeto ya que el primero no entra en el resto de la mochila. Así nuestro beneficio total es 6. Si hubiésemos escogido el primer y segundo objeto obtendríamos un peso total de 4 y un beneficio total de 7. □

Dos Mochilas

Suponed que tenemos n objetos y dos mochilas con capacidades M_1 y M_2. Suponiendo que podemos partir los objetos. Diseñar un algoritmo voraz para llenar las dos mochilas con los n objetos.
Solución.- Mientras que entre el objeto que seleccionemos en la primera mochila lo echaremos en ella. Cuando no entre en la primera mochila y si en la segunda lo podremos en la segunda mochila. En el caso de que no entre en la primera y en la segunda, veremos si el resto que queda es mayor que el peso del objeto. Si es así completamos la primera mochila, y echamos el resto en la segunda. En otro caso nos puede quedar un resto en la segunda mochila que se completa con la proporción que entre.

```
 float DosMochilas(int M1,int M2, const vector<int>&b, const vector<int>&p,
                   vector<float >&X){
int n = X.size(); // numero de objetos
//Inicializacion
for (innt i=0;i<n;i++)
  X[i]=0;

int pesoAct=0; //peso acumulado
float b_Act=0.0;//beneficio acumulado
int en_m1=0; // peso acumulado en la primera mochila
in  en_m2=0; // peso acumulado en la segunda mochila
```

```
12 while (pesoAct<M1+M2){
13    int i=Seleccion(p,b); //escogemos el objeto mas prometedor
14   if (p[i]<=(M1-en_m1)) { //si entra en la primera mochila
15       X[i]=1; pesoAct+=p[i]; en_m1+=p[i]; b_Act+=b[i];
16   }
17   else
18    if (p[i]<=(M2-en_m2)){ //si entra en la segunda mochila
19       X[i]=1; pesoAct+=p[i]; en_m2+=p[i]; b_Act+=b[i];
20    }
21    else
22     if (pesoAct+p[i]<=M1+M2){ //el objeto puede entrar entero pero
23                          // un trozo en la primera mochila
24                          // y el resto en la segunda mochila
25             X[i]=1; pesoAct+=p[i];
26             //trozo en la primera mochila
27             float a = (M1-en_m1)/p[i]; en_m1+=a;
28             en_m2+=(p[i]-a);//resto en la segunda mochila
29             b_Act+=b[i];
30     }
31     else{ //solamente entra un trozo del objeto
32      X[i]= ( (M1-en_m1)+(M2-en_m2))/p[i];
33      pesoAct=M1+M2; //se completa las dos mochilas
34      b_Act+=X[i]*b[i];//beneficio de esa parte
35     }
36 }
37 return b_Act;
38 }
```

3.6 Problema: Planificación de Tareas

Este problema consiste en dada una secuencia de tareas o trabajos tenemos que establecer un orden temporal de las tareas que se van a realizar. Puede que todas no se puedan realizar por sus restricciones de cuando debe hacerse y cuanto tardan en hacerse. Realizar un tarea consume por lo tanto un tiempo y además por realizarse obtenemos un beneficio. Así las consideraciones son:

- Tenemos un procesador y n tareas a realizar
- Todas las tareas requieren una unidad de tiempo para ejecutarse (p.e 1 segundo). Por cada tarea además tenemos:
  - b_i: beneficio que se obtiene si se ejecuta la tarea i.
  - d_i: plazo máximo para ejecutarse la tarea i. La tarea i podrá solamente ejecutarse si se hace en un instante igual o anterior a d_i

En general puede que no sea posible ejecutar todas las tareas. El objetivo y subproblemas a resolver son:

Objetivo.- Dar una planificación de las tareas a ejecutar $(s_1, s_2, \cdots, s_m)$ de forma que se maximice el beneficio total:

$$b_{TOTAL} = \sum_{i=1\cdots m} b_{s_i} \Longrightarrow$$

s_i : representa la tarea que se ejecuta en el en el instante i

Problemas a resolver

1. ¿Qué tareas se ejecutan?
2. ¿En qué orden se ejecutan?

Ejemplo 3.6.1
Supongamos que tenemos 4 tareas $n = 4$, con beneficios $b = (100, 10, 15, 27)$, y plazos máximo $d = (2, 1, 2, 1)$. En este caso la tarea 1 debe ejecutarse como máximo en el segundo 2, la tarea 2 como máximo en el segundo 1, y así el resto. Posibles temporizaciones sería las siguientes:

Tiempo T	1	2
S_{Tarea}	1	3
b	100	15
d	2	2
$B_{TOTAL} = 115$		

Tiempo T	1	2
S_{Tarea}	4	3
b	27	15
d	1	2
$B_{TOTAL} = 42$		

Tiempo T	1	2
S_{Tarea}	1	4
b	100	27
d	2	1
NO FACTIBLE		

Tiempo T	1	2
S_{Tarea}	4	1
b	27	100
d	1	2
$B_{TOTAL} = 127$		

Como se puede observar en la tabla la mejor planificación es la cuarta en la que en el segundo 1 se ejecuta la tarea 4 y en el segundo 2 se ejecuta la tarea 1, obteniendo un beneficio total de 127. Por contra la tercera planificación no es factible ya que la tarea 1 y tarea 4 se debe ejecutar en el segundo 1 como máximo.

□

Un algoritmo trivial para la planificación de taras sería comprobar todos los posibles órdenes (todas las permutaciones) de las tareas y quedarse con la mejor entre las factibles. Este algoritmo tiene como cota inferior $\Omega(n!)$. Por lo tanto es de necesidad formular un algoritmo que reduzca sustancialmente el tiempo de ejecución. Veamos como lo hace la técnica voraz.

3.6.1 Algoritmo Voraz

Los pasos y elementos que nuestro algoritmo voraz tendrán son los siguientes:

1. *Inicialización.-* Empezamos con un planificación sin tareas. Los candidatos son todas las tareas. En cada paso añadiremos un tarea entre las candidatas que nos quedan.
2. *Solución.-* La solución está formada por el conjunto de tareas seleccionadas junto con el instante en el que se ejecuta cada una de ellas. Así la solución se especifica como $S = (s_1, s_2, \cdots, s_m)$ siendo s_i la tarea que se ejecuta en el instante i.
3. *Función de Selección.-* Escoger de entre las candidatas la que tenga mayor beneficio.
4. *Función Factible.-* Comprueba que el plazo máximo de la tarea i, que es d_i, es menor o igual al instante en el que se ejecuta.

El algoritmo para la planificación sería el siguiente:

```
1  /**
2   * @brief Obtiene una planificacion de tareas realizar
3   *          en el tiempo para maximizar el beneficio total
4   * @param T: vector con las tareas ordenadas de mayor a menor beneficio
5   * @param S: vector con las indices en T de las tareas seleccionadas
6   * @pre Se supone que T esta ordenado por beneficio
7   * @return el beneficio total obtenido
8  **/
```

```
int Planificacion(const vector<tarea> &T ,vector<int> &S){
  int tiempos=S.size();
  int n_tareas=T.size();
  int i=0,puestas=0;
  int bene_total=0;
  while (i<n_tareas && puestas<tiempos){
          int candidata=i;
          if (Factible(T,S,candidata,puestas)){
            bene_total+=T[candidata].beneficio;
            puestas++;
          }
          i++;
  }
  return bene_total;
}
```

Una tarea se define de la siguiente forma:

```
 struct tarea{
  int beneficio;
  int d;//plazo maximo
  int key; //clave de la tarea
};
```

A la función Planificación se le debe dar las tareas ordenadas de mayor a menor beneficio.
La función Factible comprueba si se puede ejecutar la tarea.Una posible implementación podría ser:

```
/**
 * @brief Comprueba si se puede anhadir un nuevo elemento a la solucion
 * @param T: conjunto de tareas
 * @param S: vector con la solucion hasta el momento. Si la nueva tarea
             se puede anhadir S ES MODFIICADO con dicha nueva tarea
 * @param nueva: indice en T de la nueva tarea candidata a anhadir
 * @param k: numero de tareas ya anhadidas en S
 * @return true si la nueva tarea se ha anhadido a S siendo esta nueva
          solucion factible.  false en caso contrario
 *
 * */
bool Factible(const vector<tarea> &T,vector<int> &S,int nueva,int k){
  int n=S.size();
  vector<int> Saux(n,-1);
  int i =0;
  //Mientra que el tiempo es menor que el plazo de la nueva tarea
  while (i<k && (i+1)<T[nueva].d){
      Saux[i]=S[i];
      i++;
  }
  Saux[i]=nueva; //Anhadimos la nueva tarea a la solucion
                 //se pone en su plazo maximo
  while(i<k){ //Anhadimos el resto de S
      Saux[i+1]=S[i];
      i++;
  }
  i=0;
```

```
28    //Se comprueba si di>=i en otro caso la solucion construida no es factible
29    while (i<=k && T[Saux[i]].d>=(i+1)) i++;
30
31    if (i>k) { //Es factible
32      S=Saux; //Nos quedamos con la solucion construida
33      return true;
34    }
35    else return false; //NO era factible
36  }
```

Se puede comprobar que este algoritmo tarda en el peor de los caso $\mathbf{O}(n^2)$. Ya que recorre todas las tareas en el peor de los casos y para cada tarea ejecuta Factible que tiene un tiempo en el peor de los casos n.

3.7 Problema de la Asignación

El problema de la asignación consiste en dado n trabajadores y n trabajos hacer una asignación a un trabajador a un trabajo. La peculiaridad es que cada trabajador puede realizar un trabajo con más o menor rendimiento porque esta más o menos cualificado. Para formular nuestro problema tenemos como entrada una matriz de beneficios (rendimiento) B en la que cada elemento $B[i,j]$ representa el beneficio que se obtiene al asignar al trabajador i el trabajo o tarea j.
Objetivo.- Asignar una tarea a cada trabajador de manera que se maximice la suma del beneficio o rendimiento:

$max \sum_{i=1}^{n} B[p_i, t_i]$ sujeto a $\underbrace{p_i \neq p_j \Longrightarrow t_i \neq t_j \forall i \neq j}$

Esta condición exige que la asignación es uno a uno

Ejemplo 3.7.1
Supongamos que tenemos la siguiente matriz de beneficios (en las filas los trabajadores o personas y en las columnas trabajos o tareas):

		Tareas		
		1	**2**	**3**
Personas	**1**	4	9	1
	2	7	2	3
	3	6	3	5

- **Asignación 1.-** (P1,T1) (P2,T3) (P3,T2) (P son personas, T son Tareas). Esta asignación tiene un beneficio total

$$B_{TOTAL} = 4+3+3 = 10$$

- **Asignación 2.-** (P1,T2) (P2,T1) (P3,T3) (P son personas, T son Tareas). Esta asignación tiene un beneficio total

$$B_{TOTAL} = 9+7+5 = 21$$

□

3.7.1 Algoritmo Voraz

Los pasos y elementos que nuestro algoritmo voraz tendrán son los siguientes:

- **Candidatos.-** Personas a las que no se les ha asignado ningún trabajo o tarea.
- **Solución.-** $S = (t_1, t_2, \cdots, t_n)$ siendo t_i: la tarea que se le asigna al trabajador o persona i. S es solución cuando se le ha asignado a todos los trabajadores una única tarea y a cada tarea un único trabajador.
- **Función de Selección.-** Escoge para el trabajador i la tarea aún no asignada que mayor beneficio proporciona.
- **Función Factible.-** Siempre es true
- **Función Objetivo.-**

$$max \sum_{i=1}^{n} B[i,t_i] \text{ sujeto a } i \neq j \Longrightarrow t_i \neq t_j \forall i \neq j$$

A continuación veamos el algoritmo voraz.

```
/**
 @brief Obtiene la asignacion uno a uno de trabajadores y tareas buscando
        maximizar el beneficio total
 @param B: matriz de beneficios. filas trabajadores, columnas tareas o trabajos
 @param S: vector solucion. en cada posicion i se indica el trabajo asignado al
        trabajador i
 @return el beneficio total obtenido.
 */
int Asignacion_Voraz(const Matriz &B, vector<int> &S){
  int n=S.size(); //n numero de trabajadores y trabajos
  vector<bool> asignadas(n,false);
  int i=0;
  int bene_total=0;
  while (i<n){
     //busca entre las tareas libre la mejor para el trabajador i
     int j = Seleccion(i,B,asignadas);
     asignadas[j]=true;
     bene_total+=B(i,j);
     S[i]=j;
     i++;
  }

}
```

La función *Seleccion* busca la tarea de las que quedan que de mayor beneficio para ser asignada a un trabajador.

```
 /**
  @brief Devuelve la tarea de las que quedan libre para ser asignada a un trabajador.
  @param trabajador: indice del trabajador al que se le busca tarea
```

```
4     @param B: matriz de beneficios
5     @param asignadas: vector indicando por cada tarea si esta asignada o no
6    */
7  int Seleccion(int trabajador, const Matriz &B, const vector <bool>&asignadas){
8   int n=asignadas.size();
9   int max = numeric_limits<in>::min(); //- infinito
10  int job=-1;
11  for (int i=0; i<n;i++)
12   if (!asignada[i]){
13     if (max<B(trabajador,i)){
14       max = B(trabajador,i);
15       job = i;
16     }
17   }
18   return job;
19 }
```

3.7.2 Optimalidad y Eficiencia

Con respecto a la eficiencia de la función *Asignacion_Voraz* tiene un bucle while (línea 14) que recorre todos los trabajadores (n), y para cada trabajador en el peor de los casos recorre todas las tareas, también n. Por lo tanto la eficiencia en el peor de los casos de nuestro algoritmo voraz es $\mathbf{O}(n^2)$.

Con respecto a si nuestro algoritmo encuentra siempre la solución óptima, el mejor beneficio, la respuesta es no. Para ver esto, supongamos que tenemos la siguiente matriz de beneficios:

		Tareas		
		1	**2**	**3**
Personas	**1**	20	19	10
	2	20	2	10
	3	10	1	20

La asignación que hace nuestro algoritmo voraz sería:

- Trabajador 1 $\Longrightarrow$ Tarea 1
- Trabajador 2 $\Longrightarrow$ Tarea 3
- Trabajador 3 $\Longrightarrow$ Tarea 2

Con un beneficio total de **31**.

En cambio la solución óptima sería

- Trabajador 1 $\Longrightarrow$ Tarea 2
- Trabajador 2 $\Longrightarrow$ Tarea 1
- Trabajador 3 $\Longrightarrow$ Tarea 3

Con un beneficio total de **59**.

En conclusión es un algoritmo rápido pero proporciona soluciones aproximadas.

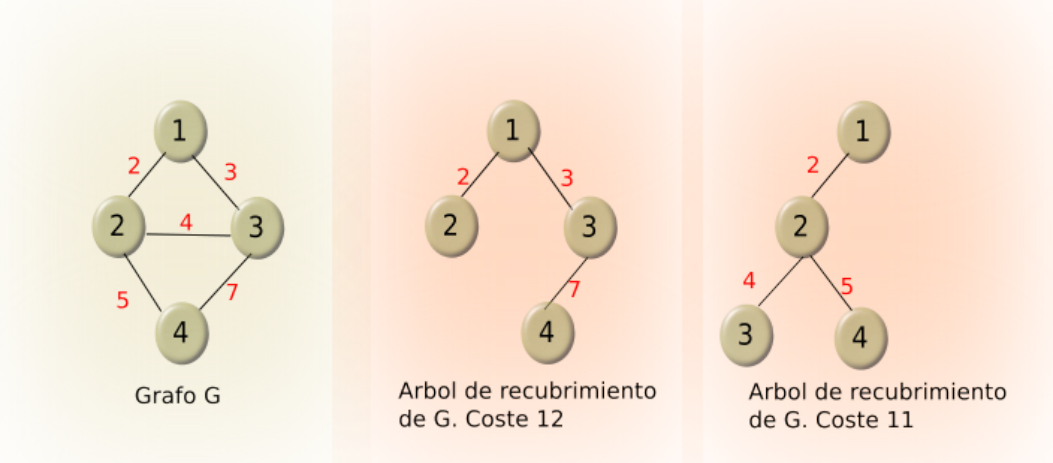
3.8 Grafos

En esta sección analizaremos diferentes algoritmos que trabajan con grafos. Un grafo es una estructura de datos que permite modelizar muchos problemas: circuitos eléctrico, rutas de comercio, relaciones, redes sociales, etc. De forma matemática podemos decir que un Grafo **G** está compuesto por un conjunto de vértices $V = (v_1, v_2, \cdots, v_n)$ y un conjunto de aristas $A = (v_i, v_j)$ tal que $v_i, v_j \in V$, que conectan a vértices. Con estas definiciones diremos que $\mathbf{G} = (V, A)$.

3.8.1 Arbol de recubrimiento minimal

Árbol de recubrimiento de G. El árbol de recubrimiento de un grafo se define al subgrafo de **G** sin ciclos (cada vértice puede aparecer como final de una arista una sola vez, tiene un único padre) y contiene todos sus vértices.

Ejemplo 3.8.1
En la siguiente imagen se muestra un grafo con cuatro vértices. En este caso las aristas están etiquetadas por pesos. A la derecha se muestro dos posibles árboles de recubrimiento minimal con coste 12 y coste 11.



□

En caso de existir varios árboles de recubrimiento, nos interesa aquel con coste mínimo, es decir, que la suma de los pesos de las aristas del árbol obtenido sea el menor posible.
Los algoritmos basados en la técnica voraz que obtiene el árbol de recubrimiento minimal son los algoritmos de Prim y el algoritmo de Kruskal. En ambos algoritmos se tiene como conjunto de candidatos las aristas y en cada paso se añade una arista o arco. La forma de escoger esa arista es lo que distingue a los dos algoritmos.

Algoritmo de Kruskal

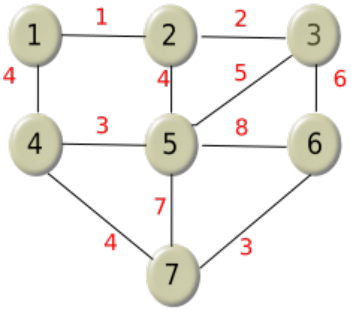
A continuación vamos a ver la forma de actuar del algoritmo de Kruskal para obtener el árbol de recubrimiento minimal de un grafo **G**. Los pasos a seguir de este algoritmos son:

1. Ordenar las aristas de forma creciente por peso de la arista
2. Escoger una aristas con vértices en dos componentes conexas diferentes

3. Estas dos componentes conexas pasan a ser un única componente conexa.

Veamos un ejemplo de funcionamiento.

Paso 1.- Ordenamos las aristas de forma creciente por peso:
(1,2) (2,3) (4,5)
(6,7) (1,4) (2,5)
(4,7) (3,5) (3,6)
(5,7) (5,6)



Grafo G

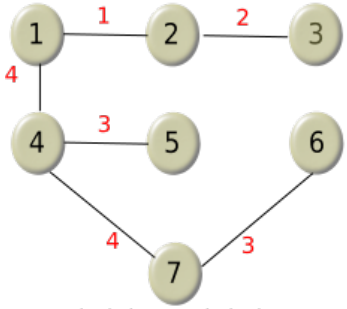
Inicialmente cada vértice forma una componente conexa. En nuestro ejemplo tenemos 7 componentes conexas:

$$\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\}$$

El algoritmo para cuando nos quede solamente un única componente conexa, y esta contiene todos los vértices del grafo. Veamos los siguientes pasos del algoritmo sobre nuestro grafo ejemplo:

Iteración	Arista Seleccionada	Componentes Conexas
1	(1,2)	{1,2} {3} {4} {5} {6} {7}
2	(2,3)	{1,2,3} {4} {5} {6} {7}
3	(4,5)	{1,2,3} {4,5} {6} {7}
4	(6,7)	{1,2,3} {4,5} {6,7}
5	(1,4)	{1,2,3,4,5} {6,7}
6	(4,7)	{1,2,3,4,5,6,7}

El árbol de recubrimiento minimal sería el que se muestra a la derecha, con un coste de 17.



Arbol de recubrimiento de G. Coste 17

```
1  int Kruskal(const Grafo &G) {
2    vector <pair<int,int> > Aristas;
3
4    vector<int > vertices;
5    vertices = G.GetVertices();
6    Aristas= G.GetAristas();
7    sort(Aristas.begin(),Aristas.end()); //ordena de menor a mayor peso
8    int n = vertices.size();
9
10   Grafo  Conexas(vertices);//componente conexas aristas cero
11   int i=0;
12   while(i<Aristas.size()) {
13    pair<int,int> e = A[i];
14    int compu = Conexas.GetComponente(e.first);
15    int compv = Conexas.GetComponente(e.second);
16    if (compu!=compv){
17       Conexas.AddArista(e.first,e.second);
18    }
19    i++;
20   }
21 }
```

El árbol de recubrimiento encontrado por Kruskal es minimal.

Eficiencia.- Sea a el número de aristas del grafo y n el número de vértices. Como máximo se puede considerar todas las aristas del grafo. Al fusionar dos componentes (en el código añadiendo la arista a conexas, línea 17) nos quedamos con una componente menos. Si tenemos los vértices ordenados y asociados a ellos su componente conexa, obtener la componente gasta $log_2(n)$. Fusionar podemos hacerlo en tiempo constante. Luego como el while (línea 12) en el peor de los casos itera a veces, la eficiencia total en el peor de los casos es $\mathbf{O}(a \cdot log_2(n))$. Por otro lado a se puede limitar al menor número que es cuando nuestro grafo es un camino, $n-1$ aristas. Y en el peor caso a puede adoptar el valor $n \cdot \frac{n-1}{2}$ que ocurre cuando nuestro grafo es completo. Por lo tanto:

$$\underbrace{n-1}_{\substack{\text{grafo que}\\ \text{es un camino}}} \leq a \leq \underbrace{n \cdot \frac{n-1}{2}}_{\text{grafo completo}}$$

Algoritmo de Prim

En el algoritmo de Prim la principal diferencia con el algoritmo de Kruskal es que coge la arista en el que uno de los vértices ya pertenece al árbol de recubrimiento minimal el otro vértice no. El pseudocódigo de este algoritmo sería el siguiente: En el pseudocódigo B representa el conjunto de vértices del árbol minimal y T el conjunto de aristas del árbol minimal. Veamos un ejemplo del ejecución del algoritmo de Prim.

Algorithm 1 Algoritmo de Prim

function PRIM(G=<N,A>)

$T \leftarrow \emptyset$

$B \leftarrow$ {un miembro arbitrario de N}

Mientras $B \neq N$ hacer

Buscar $e = (u,v)$ de longitud mínima tal que $u \in B$ y $v \in N \setminus B$

$T \leftarrow T \cup \{e\}$

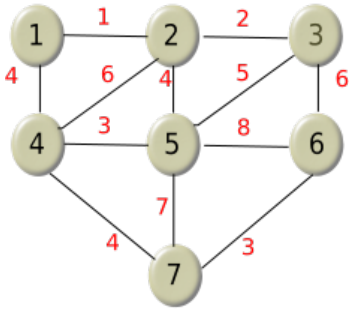
$B \leftarrow B \cup \{v\}$

end

devolver T

Paso 1.- Ordenamos las aristas de forma creciente por peso:

$(1,2)$ $(2,3)$ $(4,5)$
$(6,7)$ $(1,4)$ $(2,5)$
$(4,7)$ $(3,5)$ $(2,4)$
$(3,6)$ $(5,7)$ $(5,6)$



Grafo G

Veamos los siguientes pasos del algoritmo sobre nuestro grafo ejemplo:

ITERACIÓN	ARISTA (u,v)	B
1	(1,2)	{1,2}
2	(2,3)	{1,2,3}
3	(1,4)	{1,2,3,4}
4	(4,5)	{1,2,3,4,5}
5	(4,7)	{1,2,3,4,5,7}
6	(7,6)	{1,2,3,4,5,7,6}

Como era de esperar el resultado obtenido por Prim es el mismo que el obtenido por Kruskal. Una implementación sencilla del algoritmo se podría llevar a cabo suponiendo que los nodos de **G** están enumerados de 1 a n $N = \{1,2,\cdots,n\}$. Además vamos a tener una matriz simétrica L que da el peso de las aristas, por ejemplo $L[i,j]$ da el peso de la arista que une el vértice i y j. Si $L[i,j] = \infty$ significa que no existe arista entre los nodos. Supongamos que tenemos un vector *mas_proximos* que para todo $i \in N \setminus B$ que nos dice el nodo en B más cercano a i, e i aún no pertenece a B. Además disponemos el vector *dis_min* que da para cada $i \in N \setminus B$ la distancia desde i al nodo más próximo en B. Para todo nodo en B hacemos $dis\_min[i] = -1$.

```
typedef pair<int,int> arista;
void Prim(const Matriz &L, list<arista> & T, int N){
  vector<int> mas_proximo(N+1); //vamos a indexar desde 1
  vector<float>dist_min(N+1,-1); //vamos a indexar desde 1

  //Inicializamos mas_proximo y distmin
  for (int i=1;i<=N;i++){
   mas_proximo[i]=1;//el vertice escogido arbitrario es 1
   dist_min[i]=L(i,1);
  }
  int k;
  for (int=l=1;l<=N-1;l++){
   float min= numeric_limits<float>::max();
   for (int j=1;j<=N;j++){
     if (dist_min[j]>=0 && dist_min[j]<min){
       min=dist_min[j];
       k=j;
     }
   }

    arista a (mas_proximo[k],k]);
    T.insert(T.end(),a);
    dist_min[k]=-1;
    //corregimos la distancia minima de los nodos a un
    //no insertados en B
    for (int j=2;j<=N;j++)
    if (L(j,k)<dist_min[j]){
      dist_min[j]=L(j,k);
      mas_proximo[j]=k;
    }
  }
}
```

La eficiencia de este algoritmo es $\Theta(n^2)$.

3.8.2 Caminos Mínimos

Desde el punto de vista que un grafo puede representar rutas, vías, circuitos, y las arista nos informan sobre la distancia entre dos vértices, un problema en este contexto sería que camino seguir dentro del grafo para hacer el menor recorrido para partiendo de un vértice llegar a otro.
El planteamiento general es: Dado un grafo $\mathbf{G} < N, A >$ siendo N: el conjunto de nodos o vértices y A: el conjunto de aristas, el problema consiste en encontrar como ir desde un vértice origen al resto de los vértice con longitud mínima. Unos de los algoritmos que resuelven este problema es el algoritmo de Dijkstra.

Algoritmo de Dijkstra

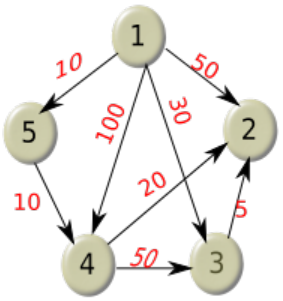
Los elementos del algoritmo de Dijkstra son los siguientes:

- C: conjunto de candidatos que son los vértices.
- S: vértices seleccionados
- *Función de Selección.-* Se selecciona aquel nodo en C cuya distancia al origen sea mínima
- *Añadir a la Solución.-*. El nodo seleccionado se añade a la solución
- *Solución.-*. Obtenemos una solución cuando S contiene todos los vértices. Es decir $S = N$.
- *Función Objetivo.-*. El camino obtenido debe ser aquel de menor longitud.

En todo momento se debe cumplir que $N = C \cup S$. Para plantear el algoritmo tenemos que tener un vector D que contiene la distancia mínima de cualquier vértice al vértice origen escogido.Además un segundo vector P no hace falta, de forma que $P[i]$ nos indica el nodo anterior por el que hay que pasar para llegar i. En cada iteración si el vértice escogido es k, se recalcula para todo vértice aún no escogido si es mejor pasar por k. De esta forma se evalúa en cada iteración:

$$D[j] = minD[j], D[k] + L[k, j])$$

Si $D[j]$ se modifica con $D[k] + L[k, j]$ entonces $P[j]$ se le asigna k, ya que para llegar a j lo mejor es pasar por k.
Veamos el modo de funcionamiento del algoritmo para el siguiente grafo:



ITERACIÓN	v	C	D	P
Inicialización			$D[2] = 50 \quad D[3] = 30$	$P[2] = 1 \quad P[3] = 1$
1	1	$\{2,3,4,5\}$	$D[4] = 100 \quad D[5] = 10$	$P[4] = 1 \quad P[5] = 1$
2	5	$\{2,3,4\}$	$D[2] = 50 \quad D[3] = 30$ $D[4] = 20$	$P[2] = 1 \quad P[3] = 1$ $P[4] = 5$
3	4	$\{2,3\}$	$D[2] = 40 \quad D[3] = 30$	$P[2] = 4 \quad P[3] = 1$
4	3	$\{2\}$	$D[2] = 35$	$P[2] = 3$

Por ejemplo $D[4]$ pasa de un valor de 100 (en la iteración 1) a un valor de 20 en la iteración. Esto es asi ya que es más económico o más corto pasar por 5 , haciendo el camino $1 \to 5 \to 4$ que hacer $1 \to 4$. Por eso además en $P[4]$ se modifica a 5.
El código de Dijkstra sería el siguiente:

```
void Dijkstra(const Matriz &L , vector<int> &D,vector<int>&P){

 list <int > C; //conjunto de candidatos
 list <int > S; //solucion

 int n= L.num_filas();
 //Inicializacion,el vertice origen es 0
 for (int i=1;i<n;i++){
   C.insert(i);
   D[i]= L[0,i];
   P[i]=0;
 }
 for (int i=1;i<n-1;i++){
    //buscamos el vertice aun no analizado con menor
    //distancia
    int v= BuscaMinimo(D,C);
    C.erase(v);
    S.insert(v);
    list<int>::iterator it= C.begin();
    while (it!=C.end()){
      if (D[*it]>D[v]+L(v,*it)){
          D[*it]= D[v]+L(v,*it);
          P[*it]=v;
      }
      ++it;
    }
  }


}

int BuscaMinimo(const vector<int>&D,const list<int>C){
    list<int>::const_iterator it= C.begin();
    int min = numeric_limits<int>::max();
    int best_v;
    for (; it!=C.end();++it){
     if (min>D[*it]) {
      best_v=*it;
      min=D[*it];
     }
    }
}
```

Eficiencia.- Viendo la función Dijkstra podemos plantear la siguiente ecuación para obtener la eficiencia

$$\sum_{i=1}^{n-2}\sum_{i=j}^{n-1} 1 =$$

$$\sum_{i=1}^{n-2} n-j+1 =$$

$$\sum_{i=1}^{n-2} n - \sum_{i=1}^{n-2} j + \sum_{i=1}^{n-2} 1 =$$

$$n\cdot(n-2) - (n-2)\cdot\tfrac{n-1}{2} + n - 1 \in \Theta(n^2)$$

3.8.3 El viajante de Comercio

En el problema del viajante de comercio se conocen las distancias entre un cierto número de ciudades. Un viajante de comercio debe a partir de una de ellas visitar cada ciudad exactamente una vez y regresar al punto de partida, con objetivo de recorrer el menor número de kilómetros.

Este problema se puede modelizar con un grafo de la siguiente forma: Dado un grafo conexo (todas las ciudades están conectadas) y con pesos en las etiquetas (significando la distancia entre ciudades) y dado uno de sus vértices v_0 (que es donde parte el viajante de comercio) el objetivo es encontrar el ciclo Hamiltoniano (se pasa una sola vez por cada uno de los vértices, a excepción del vértice v_0). Este ciclo Hamiltoniano debe ser de coste mínimo empezando por v_0 y terminando por v_0.

Algoritmo Voraz

Sea C el camino construido hasta el momento que acaba en v y comienza en v_0: $(v_0, v_1, , v)$. Las situaciones posibles en el algoritmo voraz son:

1. C es vacío. Estamos en el proceso de inicialización y por lo tanto añadimos a C el vértice origen que es v_0.
2. C contiene todos los vértices. Estamos al final del algoritmo y lo que debemos añadir al final de C es v_0 para que el recorrido vuelva al punto de origen.
3. C no contiene todos los vértices. En este caso, si $C = (v_0, v_1, \cdots, v)$, tenemos que añadir a C la arista (v, w) de menor longitud tal que $w \notin C$.

```
1 void Viajante_Voraz(const Grafo & G, vector <int> &Solucion){
2   int n= G.num_vertices(); //numero de vertices
3   vector<bool> ya_esta(n,false);//vertices que estan en la Solucion
4
5   int vertice =0;
```

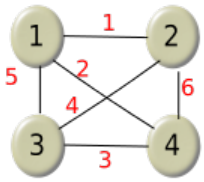
```
6     ya_esta[vertice]=true;
7     Solucion.push_back(vertice);
8     for (int i=1;i<n;i++){
9        //buscamos el mas proximo a vertice
10       vertice=Busca(g,vertice,ya_esta);
11       Solucion.push_back(vertice);
12       ya_esta[vertice]=true;
13    }
14    Solucion.push_back(0);
15 }
```

A continuación vamos a especificar la función Busca que obtiene el vértice más próximo al último insertado en Solucion

```
1  int Busca(const Grafo &G,int vertice,const vector<bool> & ya_esta){
2    int n= G.num_vertices();
3    int mejorvertice=0;
4    int min= numeric_limits<int>::max();
5    for (int i=0;i<n;i++){
6      if (i!=vertice)
7       if (!ya_esta[i] && G.arista(vertice,i)<min){
8         min= G.arsita(vertice,i);
9         mejorvertice=i;
10      }
11   }
12   return mejorvertice;
13
14 }
```

Eficiencia.- Como vemos en la función *Viajante_Voraz* se recorre todos los vértices n, y para cada vértice se invoca a la función *Busca* que recorre de nuevo los n vértices. Por lo tanto este algoritmo es $\Theta(n^2)$.

Ejemplo 3.8.2
Aplicar el Algoritmo voraz para resolver el problema del Viajante de Comercio dado el siguiente grafo.



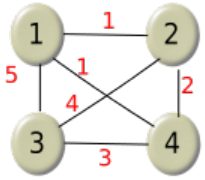
Los pasos que se ejecutan son los siguientes:

Paso	C	v	Distancia
1	(1)	1	0
2	(1,2)	2	1
3	(1,2,3)	3	5
4	(1,2,3,4)	4	8
5	(1,2,3,4,1)	1	10

En este ejemplo encontramos la solución óptima. Pero siempre no es así. En el siguiente ejemplo se muestra que este algoritmo no siempre encuentra la solución óptima.

□

Ejemplo 3.8.3
Aplicar el Algoritmo voraz para resolver el problema del Viajante de Comercio dado el siguiente grafo.



Los pasos que se ejecutan son los siguientes:

Paso	C	v	Distancia
1	(1)	1	0
2	(1,2)	2	1
3	(1,2,4)		3
4	(1,2,4,3)	4	6
5	(1,2,4,3,1)	1	11

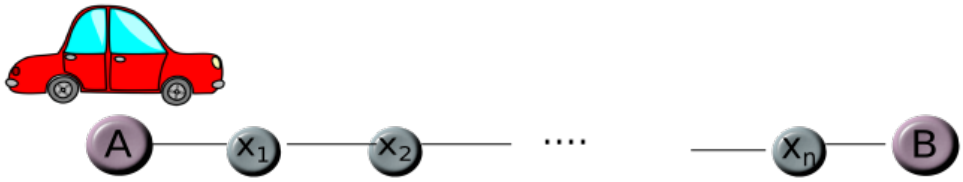
En este ejemplo se muestra que el algoritmo voraz no siempre encuentra la solución óptima. Si hubiésemos hecho el recorrido $(1,2,3,41)$ la distancia recorrida hubiese sido de 9 km.

□

3.9 Otros problemas

3.9.1 El menor número de repostajes

María ha decidido hacer un largo viaje en su coche. Parte desde la ciudad A y quiere llegar a la ciudad B. El coche de María tiene un tanque que puede viajar como máximo sin repostar L kilómetros. La distancia entre A y B es mayor que L. Tenemos a lo largo del camino n gasolineras $x_1, x_2, \cdots, x_n$. El objetivo es decidir en que gasolineras repostar haciendo el menor número de paradas. En la ciudad A María ha repostado y parte con el tanque lleno con la capacidad para hacer L kilómetros como máximo.



Algoritmo Voraz. La estrategia voraz o greedy hará una elección sobre donde repostar y reduce tu problema a un problema mas pequeño, se acorta la distancia que hay que completar. Las posibles elecciones sobre la gasolinera donde repostar puede ser:

- La gasolinera más próxima. Esta elección nos llevaría a repostar en todas las gasolineras, lo cual en vez de minimizar el número de paradas lo maximizamos.
- La gasolinera más lejana alcanzable con el contenido del depósito. Esta elección nos llevará a lograr el menor número de paradas.

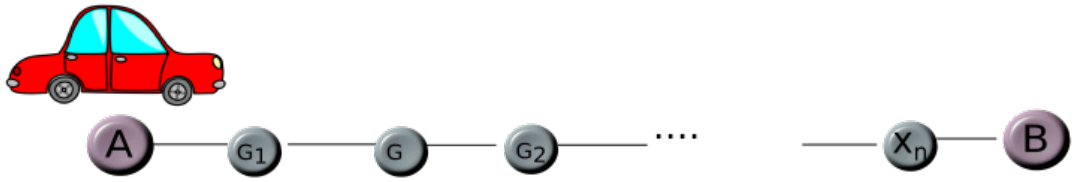
Así siguiendo la segunda estrategia en la que paramos en la gasolinera más lejos posible, partiendo de A alcanzamos la gasolinera *G* más lejana, llegando desde A a *G* con un tanque. Ahora reducimos nuestro problema a un problema más pequeño. Ahora *G* actuará como el punto de inicio *A*, con objetivo llegar a *B*. El algoritmo voraz tiene tres parámetros:

- x: es un vector donde x[0] es la ciudad origen A y x[n+1]es la ciudad destino B. El resto de elementos indica a cuantos kms de A se encuentra las gasolineras.
- n: numero de gasolineras.
- L: kilómetros que puede hacer el tanque lleno.

```
1  int Numero_Repostajes (const vector<int> &x, int n, int L){
2    int numRepostajes=0;
3    int puntoActual=0;
4    int anteriorPunto;
5    while (puntoActual<=n){
6      anteriorPunto= puntoActual;
7      //buscamos el siguiente punto de repostaje
8      while (puntoActual <=n && x[puntoActual+1]-x[anteriorPunto]<=L)
9        puntoActual++;
10
11     //Si no hemos avanzado es que la distancia era mayor que
12     //L entre dos puntos consecutivos
13     if (puntoActual==anteriorPunto)
14      return numeric_limits<int>::max();//IMPOSIBLE
15
16     if (puntoActual<=n)
17         numRepostajes++;
18   }
19   return numRepostajes;
20 }
```

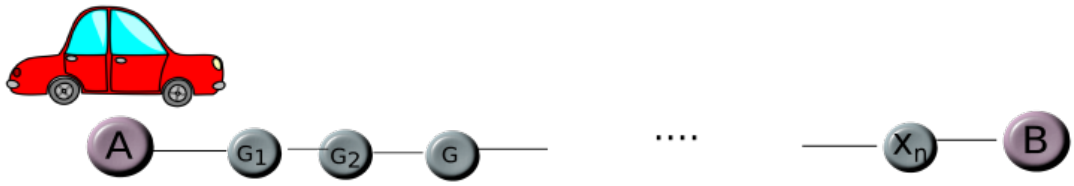
Optimalidad.Supongamos que tenemos una ruta óptima entre A y B. Sea G_1 el primer punto que se para a repostar en esa ruta óptima. Sea G la gasolinera más distante de A. Entonces si $G = G_1$ nuestra ruta es la óptima. Pero que pasa si $G1 \neq G$. Para ver esto supongamos que G_2 es la siguiente gasolinera en la ruta óptima. Y se puede dar los siguientes casos:

- Caso 1.- G esta más cerca de A que G_2.



En este caso podríamos repostar en G, en vez de G_1 y a continuación repostar en G_2, obteniendo el mismo numero de respostajes que en la ruta óptima.

- Caso 2.- G_2 esta más cerca de A que G.



En este caso podríamos repostar en G, evitando repostar en G_1 y en G_2, y así reduciendo en 1 el número de repostajes. Lo cual contradice que nuestra ruta era óptima. Por lo tanto elegir la gasolinera más lejana da el número óptimo de repostajes.

3.9.2 Compañía Eléctrica

Una compañía eléctrica debe elegir la ruta que deben seguir sus técnicos para arreglar una serie de averías que se han producido en la ciudad. Las averías están situadas en 5 puntos B, C, D, E y F. La central eléctrica está en A. La distancia entre los distintos puntos viene dada por la siguiente tabla

A					
B	6				
C	8.07	6			
D	17.55	12.70	15		
E	16.52	12.05	15.32	4	
F	19	15.32	19.38	8.6	5
	A	B	C	D	E

Encontrar un algoritmo voraz que obtenga la mejor ruta posible para minimizar la distancia recorrida por los técnicos.

Elementos del Algoritmos.

- **Candidatos C**. Todas los puntos donde hay avería B, C, D, E, F
- **Solucion S**. Un vector $S = (A, a_1, a_2, \cdots, a_n)$ donde se especifica que partiendo de A el camino a seguir.
- **Matriz de Distancias M**. $M(i, j)$ contiene la distancia del punto i al punto j.
- **posicionacutal** . Nos indica el último punto insertado en S.
- **Función de Selección**. Escoger el punto más cercano al punto indicado por *posicionactual*.
- **Función Objetivo**. *minimizar* $\sum_{i=0}^{n} M(S(i), S(i+1))$ siendo S el vector solución.

A continuación especificamos el algoritmo:

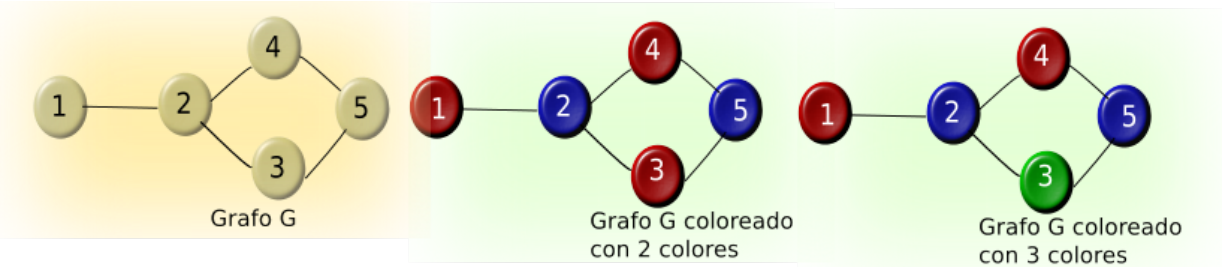
```
/**
 @brief Obtiene la mejor ruta para ir por todos los puntos donde
        existe averias partiendo del primer punto que es donde esta la electrica
 @param M : matriz de distancias entre los diferentes puntos. Es simetrica.
 @param S: vector solucion. Define el camino a seguir para pasar por todos
           los puntos
 @param C: todos los puntos que son averia y la sede de la electrica en 0.
 @return la distancia total recorrida
 */
float Ruta(const Matriz &M, vector<int> &S, const vector<int> &C){
   vector<int> Caux(C);
   S.push_back(Caux[0]);
   Caux.erase(Caux[0]);
   float dis_total=0.0;
   int posicionactual=0;
   while (Caux.size()>0){
    float min_dis= numeric_limits<float>::max();
    int posicion_min;
    //Seleccionamos el siguiente punto de averia
    for (int i=0;i<Caux.size();i++){
     if (M(S[posicionactual],Caux[i])<min_dis){
       min_dis =M(S[posicionactual],Caux[i]);
       posicion_min=i;
     }
    }

    S.push_back(Caux[posicion_min]);
    posicionacutal++;
    dis_total+=min_dis;
    Caux.erase(Caux[posicion_min]);
   }
   return dis_total;
}
```

Ejercicio 3.2
Obtener la eficiencia del problema de la compañía eléctrica.

□

3.9.3 Problema: El coloreo de Grafos

En este problema dado un grafo, el objetivo es asignar un color a cada nodo de forma que dos nodos unidos por una arista tengan siempre diferente color. El objetivo es colorear el grafo con el menor número de colores.



En esta imagen se puede observar que el menor número de colores para colorear todo el grafo es 2. Si el grafo esta completamente desconectado (el conjunto de aristas es vacío), necesitaremos 1 color solamente. Y el otro caso extremo es cuando el grafo esté completamente conectado (para n nodos tiene $n \cdot \frac{n-1}{2}$ aristas) necesitaremos n colores. Este problema es NP-completo. No se resuelve en tiempo polinomial, por lo tanto la solución que damos con voraces va a ser una solución aproximada.

Elementos del Algoritmos.

- **Candidatos C**. Todas los vértices del grafo, siendo n.
- **Solucion S**. Un vector $S = (c_1, c_2, c_3, \cdots, c_n)$ en el que cada c_i define el color asignado al nodo i. La solución es válida si para toda arista $a = (v_i, v_j)$ se cumple que $c_{v_i} \neq c_{v_j}$, se le asignan colores diferentes a los vértices de la arista.
- **Función de Selección**. Escoger cualesquiera de los nodos sin colorear
- **Función Factible(S,x)**.Es true si el color asignado a x no coincide con los colores asignados a ninguno de sus adyacentes.
- **Función Objetivo**. Minimizar el numero de colores usados.

La pasos generales del algoritmo son:

1. Inicialmente ningún vértice tiene color asignado
2. Tomamos un color colorActual=1
3. Para cada uno de los nodos sin colorear
   a) Comprobar si es posible asignarle el color colorActual
   b) Si se puede se le asigna ese color, en otro caso dejar sin colorear
4. Si quedan nodos sin colorear hacer colorActual=colorActual+1 y volver a analizar los vértices no coloreados.

La implementación en C++ sería como sigue:

```
1 /**
2  *@brief Aplicamos el algoritmo de coloreo a un grafo usando la tecnica Voraz
```

```
 * @param A: grafo de entrada
   @param vertice: vector con los vertices a colorear
 * @param result: un vector indicando por cada vertice el color asignado.
 *  No asignado ==-1
 * @return el numero de colores necesarios
 *
 */
int Voraz_Coloreo(Graph & A,vector<info_vertice> &vertices, vector<int >&result){
  int ncolores=0;
  unsigned int nv=result.size();
  vector<bool>colores_usados(nv,false);
  vector<bool>libre_color(result.size(),true);

  for (unsigned int u=0; u<nv;++u){
    Graph::vertex_set out=GetAdyacentes(A,vertices[u].v);
  //Para cada uno de los adyacentes de u
  for (Graph::vertex_set::const_iterator q= out.begin(); q != out.end(); q++){
       unsigned int v= *q;

       v--;//indexa desde 1 graph
       //saber que colores son usados
       if (result[v]!=-1)//activamos los colores usados
         libre_color[result[v]]=false;
    }
    //buscamos el primer color libre_color
    unsigned int cr;
    bool find=false;
    for (cr=0;cr<nv && !find; cr++)
       if (libre_color[cr]){
         find=true;
         result[vertices[u].v-1]=cr;
       }

    if (colores_usados[cr]==false){//color a usar
      ncolores++;//incrementamos el numero de colores
      colores_usados[cr]=true;
    }
    for (Graph::vertex_set::const_iterator q= out.begin(); q != out.end(); q++){
       unsigned int v= *q;

       v--;
       if (result[v]!=-1) // restauramos libre_color
         libre_color[result[v]]=true;
```

```
46      }
47
48    }
49    return ncolores;
50  }
```

3.9.4 Codificación Huffman

La códigos Huffman es una técnica de compresión de datos muy eficiente, llegando a ahorrar entre 20 % a 90 % del espacio. El algoritmo voraz de codificación Huffman usa una tabla de las frecuencias de ocurrencias del alfabeto que se usa para expresar las palabras. Dado un fichero con palabras podemos contabilizar para cada carácter de nuestro alfabeto con que frecuencia aparece. Por ejemplo supongamos que nuestro archivo tiene 100000 caracteres que deseamos comprimir. La frecuencia con que aparece cada carácter del alfabeto es:

	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5
Código de longitud fija	000	001	010	011	100	101
Código de longitud variable	0	101	100	111	1101	1100

Según la tabla solamente aparecen seis caracteres y la frecuencia del carácter **a** es 45000. Si usamos un código binario para expresar cada carácter de longitud fija necesitamos 3 bits como mínimo ($log_2(6)$. Usando este código nuestro fichero requiere 300000 bits para almacenarse. La cuestión es i podemos hacerlo mejor.

La respuesta es sí. Para ello podemos usar un código de longitud variable asignando códigos más cortos a los caracteres más frecuentes y códigos binarios más largos a los caracteres menos frecuentes.En este caso para representar el carácter **a** usamos el código de **0**, mientras que para el carácter **e**, que tiene una frecuencia de 9000, usamos el código **1101**. Por lo tanto el número de bits necesarios para almacenar nuestro fichero es:

$$((45\cdot 1 + 13\cdot 3 + 12\cdot 3 + 16\cdot 3 + 9\cdot 4 + 5\cdot 4) * 1000) = 224000 \text{ bits}$$

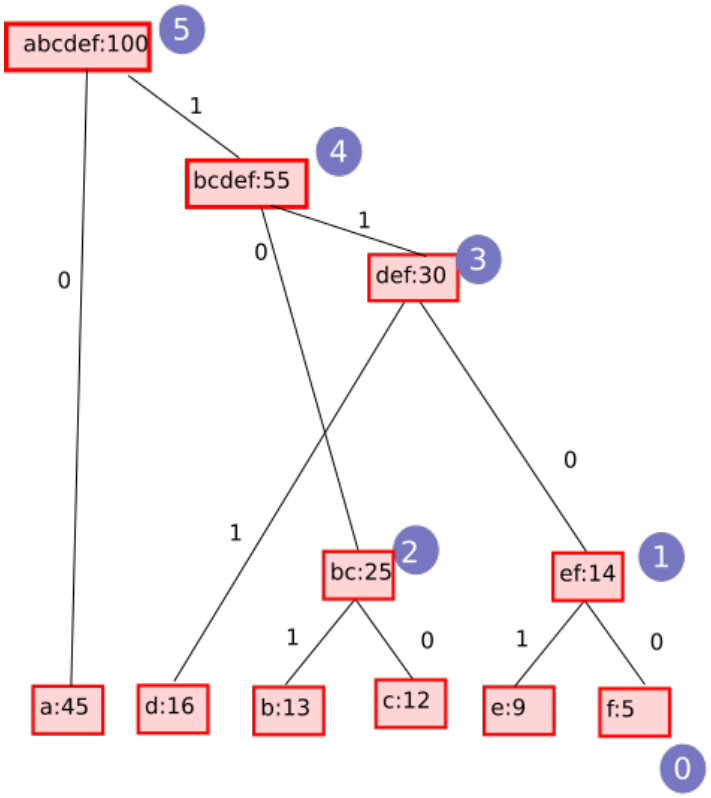
Con este sistema hemos ahorrado un 25 %.

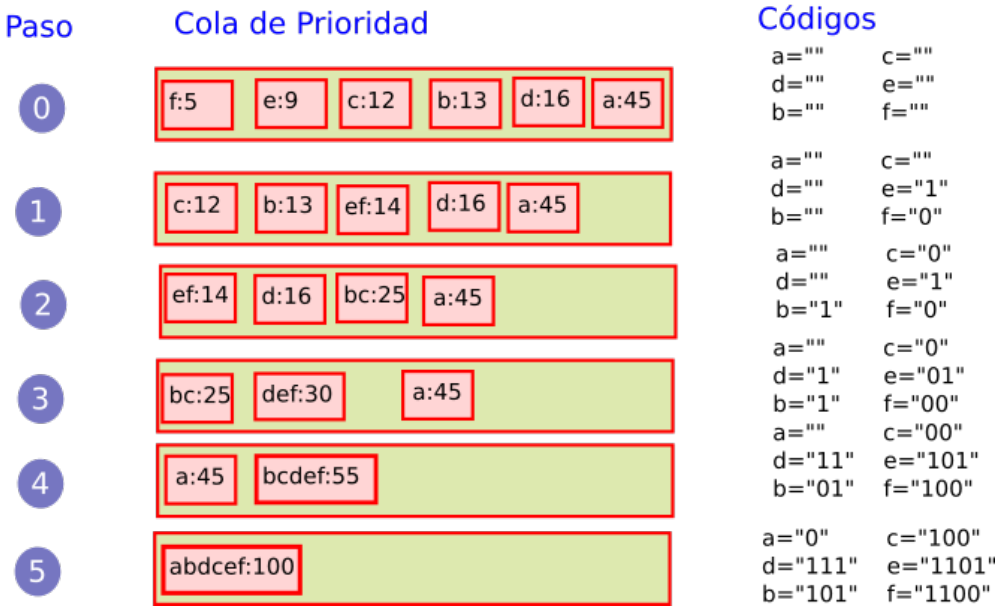
Los pasos para obtener el código Huffman de un conjunto de caracteres, con las frecuencias de aparición de cada carácter son:

- Definir las parejas: frecuencia, carácter asociado.
- Las parejas frecuencia-carácter insertarlas en una cola de prioridad. La mas prioritaria debe ser aquel par de menor frecuencia y carácter asociado.
- Sacar los dos nodos más prioritario de la cola. Sumar sus frecuencias y añadir al código de los caracteres correspondientes a uno 0 y al otro 1. Insertar un nuevo nodo en la cola con prioridad con frecuencia la suma de las frecuencias de los nodos, y caracteres asociados los sacados.
- Si la cola es vacía terminar en otro caso volver a sacar las dos nodos más prioritarios y repetir los anteriores pasos.

En realidad el algoritmo construye un árbol correspondiente al código óptimo partiendo desde las hojas hasta la raíz. En las siguientes imágenes se muestra como se construye el árbol subyacente y

como se implementan estos pasos usando una cola de prioridad.





El código C++ podría ser el siguiente:

```
/**
  @brief Obtiene la codificacion Huffman de un conjunto de caracteres
  @param frecuencias_car: vector con parejas frecuencias,caracter.
        Esta ordenado por frecuencia de menor a mayor
  @param codigos: parejas de caracter codigo. Es Modificado
  @note en codigos debe quedar los codigos por cada caracter
*/
void Huffman_Voraz(const vector<pair<int,char> > &frecuencias_car,
                   map<char,string>  &codigos){
  //cola de prioridad con frecuencia y caracteres asociados
 priority_queue<pair<int,vector<char> >pq;
 //Inicializamos la cola de prioridad
 for (int i=0;i<frecuencias_car.size();i++){
    pair<int,vector<char> >aux;
    aux.first =frecuencias_car[i].first;
    aux.second.push_back(frecuencias_car[i].second);
    pq.push(aux);
  }
  //Mientras que la cola no se vacia.
  while (!pq.empty()){
    //sacamos los dos nodos que dan una suma de frecuencias menor
    pair<int,vector<char> >left=pq.top();
    pq.pop();
    pair<int,vector<char> >right=pq.top();
    pq.pop();
    left .first+=right.first; //sumamos las frecuencias
    //a los caracteres en left le anhadimos al codigo un 0
    for (int i=0;i<left.second.size();i++)
       codigos[left.second[i]].insert(0,''0'');
    // a los caracteres en right le anhadimos al codigo un 1
    for (int i=0;i<right.second.size();i++)
       codigos[right.second[i]].insert(0,''1'');
    //ponemos todo en en left para insertarlo en la cola
    for (int i=0;i<right.second.size();i++)
       left.second.push_back(right.second[i]);
    if (!pq.empty())
       pq.push(left);
   }
}



}
```