Análisis y diseño de algoritmos

5. Algoritmos voraces

Juan Morales García,
Victor M. Sánchez Cartagena,
Jose Luis Verdú Mas

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

12 de enero de 2025

Índice

Problema de la Mochila continuo

- Sean n objetos con valores v_i y pesos w_i y una mochila con capacidad máxima de transporte de peso W.
- Seleccionar un conjunto de objetos de forma que:
  - no sobrepase el peso W (restricción)
  - el valor transportado sea máximo (función objetivo)
  - se permite fraccionar los objetos, es decir, $x_i \in [0, 1]$
- El problema se reduce a:
  - Seleccionar un subconjunto de (fracciones de) los objetos disponibles,
  - que cumpla las restricciones, y
  - que maximice la función objetivo.
- ¿Cómo resolverlo mediante un algoritmo voraz?
  - Se necesita un criterio de **selección voraz** que decida qué objeto tomar en cada momento.

Posibles criterios voraces

- Supongamos el siguiente ejemplo:

$$W = 12 \quad w = (6, 5, 2) \quad v = (49, 40, 20)$$

Criterios	**Solución**	**Peso** W	**Valor** v
valor decreciente	$(1, 1, \frac{1}{2})$	12	99
peso creciente	$(\frac{5}{6}, 1, 1)$	12	100.8
valor especifico decreciente $(\frac{v_i}{w_i})$	$(1, \frac{4}{5}, 1)$	12	**101**

Formalización

- Solución: $X = (x_1, x_2, \ldots, x_n)$, $x_i \in [0, 1]$
  - $x_i = 0$: no se selecciona el objeto i
  - $0 < x_i < 1$: fracción seleccionada del objeto i
  - $x_i = 1$: se selecciona el objeto i completo

- Función objetivo:

$$\text{máx} \left(\sum_{i=1}^{n} x_i v_i \right) \qquad \text{(valor transportado)}$$

- Restricción:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

algoritmo voraz (valor óptimo)

```
1  double knapsack(
2      const vector<double> &v, // values
3      const vector<double> &w, // weights
4      double W                 // knapsack weight limit
5  ){
6      vector<unsigned> idx(w.size());  // objects sorted by value density
7      for( unsigned i = 0; i < idx.size(); i++) idx[i] = i;
8
9      sort( idx.begin(), idx.end(),
10         [&v,&w]( unsigned x, unsigned y ){
11             return v[x]/w[x] > v[y]/w[y];
12         }
13     );
14     double acc_v = 0.0;
15     for( unsigned i = 0; i < idx.size(); i++ ) {
16         if( w[ idx[i] ] >= W ) {
17             acc_v += W/w[ idx[i] ] * v[ idx[i] ];
18             break;
19         }
20         acc_v += v[ idx[i] ];
21         W -= w[ idx[i] ];
22     }
23     return acc_v;
24 }
```

algoritmo voraz (vector óptimo)

```
1  vector<double> knapsack_W(
2      const vector<double> &v, // values
3      const vector<double> &w, // weights
4      double W                 // knapsack weight limit
5  ){
6      vector<unsigned> idx(w.size());
7      for( unsigned i = 0; i < idx.size(); i++) idx[i] = i;
8      sort( idx.begin(), idx.end(), [&v,&w]( unsigned x, unsigned y ){
9          return v[x]/w[x] > v[y]/w[y]; } );
10
11     vector<double> x(w.size());
12     double acc_v = 0.0;
13     for( unsigned i = 0; i < idx.size(); i++ ) {
14         if( w[ idx[i] ] >= W ) {
15             acc_v += W/w[ idx[i] ] * v[ idx[i] ];
16             x[ idx[i] ] = W/w[ idx[i] ];
17             break;
18         } else {
19             acc_v += v[ idx[i] ];
20             W -= w[ idx[i] ];
21             x[ idx[i] ] = 1.0;
22         }
23     }
24     return x;
25 }
```

Corrección del algoritmo /1

Teorema: El algoritmo encuentra la solución óptima

Sea $X = (x_1, x_2, \ldots, x_n)$ la solución del algoritmo $(\sum_{i=1}^{n} x_i w_i = W)$
Sea $Y = (y_1, y_2, \ldots, y_n)$ otra solución factible $(\sum_{i=1}^{n} y_i w_i = Q \leq W)$

- De la hipótesis se desprende: $W - Q = \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$
- Hay que demostrar: $V(X) - V(Y) \geq 0$, sabiendo que:
  - $V(X) - V(Y) = \sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$
- Sea j la posición del (único) objeto que puede estar fraccionado en X, es decir:
  - Sea $j : x_i = 1 \; \forall i < j$ y $x_i = 0 \; \forall i > j$
- Si podemos demostrar que $\boxed{(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j} \; \forall i}$,

  concluiremos:
  $V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i} \geq \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i = \frac{v_j}{w_j} (W - Q) \geq 0$

Corrección del algoritmo /2

¿ Es cierto que $\boxed{(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j} \ \forall i}$?

$i < j$: (parte completamente cargada en X)

- $x_i = 1,\ y_i \leq 1 \Longrightarrow x_i - y_i \geq 0$
- $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$
- Se cumple! (por la forma en la que hemos escogido j)

$i = j$: (El elemento que puede estar fraccionado)

- $\frac{v_i}{w_i} = \frac{v_j}{w_j}$
- Se cumple! (ambas partes de la inecuación son iguales).

$i > j$: (parte completamente vacía en X)

- $x_i = 0,\ y_i \geq 0 \Longrightarrow x_i - y_i \leq 0$
- $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$
- Se cumple! (puesto que el primer factor es negativo)

Índice

Algoritmos voraces: definición (1)

- Sea C un conjunto de elementos.
- **Problema de optimización por selección discreta**:[1] problema cuya solución consiste en obtener un subconjunto de C tal que:
  - Satisface unas restricciones
  - Optimiza una cierta función objetivo
- **Solución factible**: Aquella que satisface las restricciones del problema
- **Solución óptima**: Solución factible que optimiza (maximiza o minimiza) la función objetivo del problema.
- Los algoritmos voraces implementan una **estrategia** de resolución de este tipo de problemas

[1] El algoritmo de la mochila continua no pertenece a este grupo ya que se producen selecciones fraccionarias.

Algoritmos voraces: definición (2)

Definición:

Un algoritmo voraz es aquel que, para resolver un determinado problema de optimización por selección, toma decisiones siguiendo un criterio de selección voraz con la esperanza de que esta estrategia conduzca a la solución general óptima

Definición:

Un criterio de selección voraz es aquel que siempre escoge la alternativa **localmente óptima**.

Algoritmos voraces: definición (3)

- Los algoritmos voraces implementan una **estrategia** concreta para obtener la solución óptima al problema
- Pasos:
  - Se construye la solución por etapas
  - En cada etapa:
    - Se elige un elemento y del conjunto C para incluirlo en el subconjunto solución.
    - El elemento y será aquel que produce un óptimo local para esta etapa (según el estado actual de la solución y sin considerar decisiones futuras)
    - Se comprueba si la solución sigue siendo factible al añadir y a la solución. Si lo es, el candidato y se incluye en la solución. Si no lo es, se descarta para siempre.
    - La decisión es irreversible. El elemento y no se vuelve a reconsiderar nunca más.

Algoritmos voraces: esquema

Esquema voraz

```
1  t_conjuntoElementos VORAZ(t_problema dp)
2  {
3     t_conjuntoElementos y, solucion;
4     elemento decision;
5     y=prepararDatos(dp);          // preparacion de datos para facilitar seleccion
6     while(noVacio(y) || !esSolucion(solucion)) { // quedan datos por seleccionar
7                                                  // y aun no se ha llegado a la solucion
8        decision=selecciona(y);
9        if (esFactible(decision,solucion))
10          solucion=anadeElemento(decision,solucion);
11       y=quitaElemento(decision,y);   // descartar en cualquier caso
12    }
13    return solucion;
14 }
15
```

Alg. voraces: características y ámbito de aplicación (1)

- Se obtienen algoritmos eficientes y fáciles de implementar
  - Costes normalmente polinómicos y a menudo $O(n \log(n))$
  - Debido principalmente al carácter irreversible de las decisiones.
- Por qué no se usa siempre?
  - No todos los problemas admiten esta estrategia.
  - La selección de óptimos locales no conduce siempre a óptimos globales (depende del problema).
  - Se tiene que encontrar un criterio de selección que encuentre siempre la solución óptima **ante cualquier instancia del problema** (no siempre es posible; depende del problema).
    - En el problema general de la mochila discreta, este criterio no se conoce (tampoco se ha podido demostrar que no existe)
  - Es decir, tenemos que demostrar formalmente que el criterio de selección escogido consigue encontrar óptimos globales para cualquier entrada del algoritmo.

Alg. voraces: características y ámbito de aplicación (2)

- Se aplican mucho, aún en problemas para los que se sabe que no necesariamente va a encuentrar la solución óptima:
  - cuando es preferible una solución aproximada a tiempo a una óptima más costosa, o
  - cuando se busca un equilibrio entre eficiencia (complejidad reducida) y eficacia (solución aproximada que depende del criterio de selección).
  - Como punto de partida (solución aproximada) para facilitar la búsqueda de la óptima usando otros algoritmos más costosos.
- Pero **Atención!!**. En estos casos:
  - Podemos obtener una solución no óptima a un problema que la tiene o incluso podemos no encontrar ninguna solución.
  - Para cada problema particular, hay que analizar si se pueden dar estas circunstancias y sobre todo, valorar como afectarán a la toma de decisiones derivada del uso del algoritmo.

Índice

Problema de la mochila discreta (sin fraccionamiento)

- Sean n objetos con valores v_i y pesos w_i y una mochila con capacidad máxima de transporte peso W. Seleccionar un conjunto de objetos de forma que:
  - no sobrepase el peso W
  - el valor transportado sea máximo
- Formulación del problema:
  - Expresaremos la solución mediante un vector $(x_1, x_2, \ldots, x_n)$ donde x_i representa la decisión tomada con respecto al elemento i.
  - Función objetivo:

$$\text{máx}\left(\sum_{i=1}^{n} x_i v_i\right) \qquad \text{(valor transportado)}$$

  - Restricciones

$$\sum_{i=1}^{n} x_i w_i \leq W \qquad x_i \in \{0,1\} \begin{cases} x_i = 0 & \text{no se selecciona el objeto } i \\ x_i = 1 & \text{sí se selecciona el objeto } i \end{cases}$$

problema de la mochila discreta (sin fraccionamiento)

- En este caso el método voraz no resuelve el problema.
- Ejemplo:

$$W = 120 \quad w = (60, 60, 20) \quad v = (300, 300, 200) \quad v/w = (5, 5, 10)$$

  - solución voraz: $(0, 1, 1) \rightarrow$ valor total $= 500$
  - solución óptima: $(1, 1, 0) \rightarrow$ valor total $= 600$

$\Rightarrow$ La selección por valor específico no conduce al óptimo.

- No se conoce ningún criterio de selección (voraz o no) que conduzca al óptimo ante cualquier instancia de este problema.
- Aún así, esta solución se utiliza mucho como aproximación al óptimo (en esto consisten las heurísticas voraces).

Heurística voraz para resolver 'la mochila discreta'

```
1  double knapsack_d(
2      const vector<double> &v,
3      const vector<double> &w,
4      double W
5  ) {
6      vector<size_t> idx( w.size() );
7      for( size_t i = 0; i < idx.size(); i++) idx[i] = i;
8
9      sort( idx.begin(), idx.end(), [&w,&v]( size_t x, size_t y ){
10         return v[x]/w[x] > v[y]/w[y];
11     } );
12
13     double acc_v = 0.0;
14
15      for( auto i : idx ) {
16
17         if( w[i] <= W ) {
18             acc_v += v[i];
19             W -= w[i];
20         }
21     }
22
23     return acc_v;
24 }
```

El problema del cambio

- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C:
  - Una solución es una secuencia de decisiones

$$S = (s_1, s_2, \ldots, s_n)$$

  - La función objetivo es

$$\text{mín } |S|$$

  - La restricción es

$$\sum_{i=1}^{n} \text{valor}(s_i) = M$$

- La solución voraz es tomar en cada momento la moneda de mayor valor posible.

Ejemplo

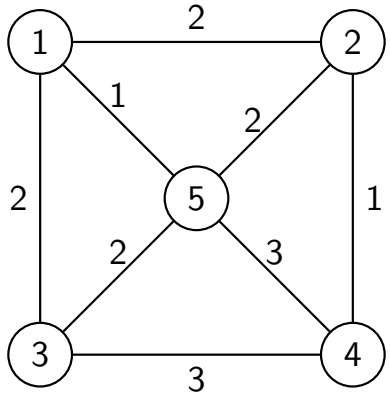
- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C:
- Sea $M = 65$

C	S	n	**Solución**
$\{1, 5, 25, 50\}$	$(50, 5, 5, 5)$	4	óptima
$\{1, 5, 7, 25, 50\}$	$(50, 7, 7, 1)$	4	óptima
	$(50, 5, 5, 5)$	4	no voraz
$\{1, 5, 11, 25, 50\}$	$(50, 11, 1, 1, 1, 1)$	6	factible pero no óptima
$\{5, 11, 25, 50\}$	$(50, 11, ?)$	???	no encuentra solución

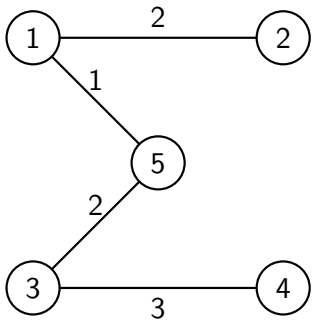
Árbol de recubrimiento de coste mínimo

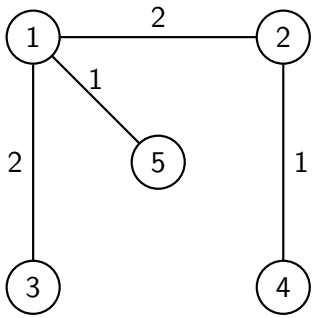
- Partimos de un grafo $g = (V, A)$:
  - conexo
  - ponderado
  - no dirigido
  - con arcos positivos
- Queremos el árbol de recubrimiento de g de coste mínimo:
  - subgrafo de g
  - con todos los vértices (recubrimiento)
  - sin ciclos (árbol)
  - conexo (árbol)
  - coste mínimo

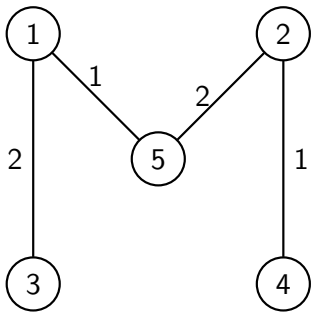
Ejemplos de árboles de recubrimiento

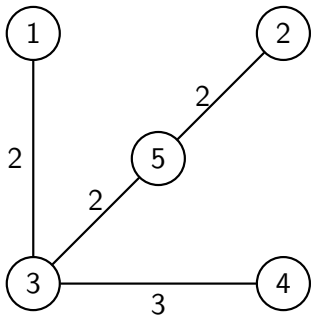


- Algunos árboles de recubrimiento (hay más):









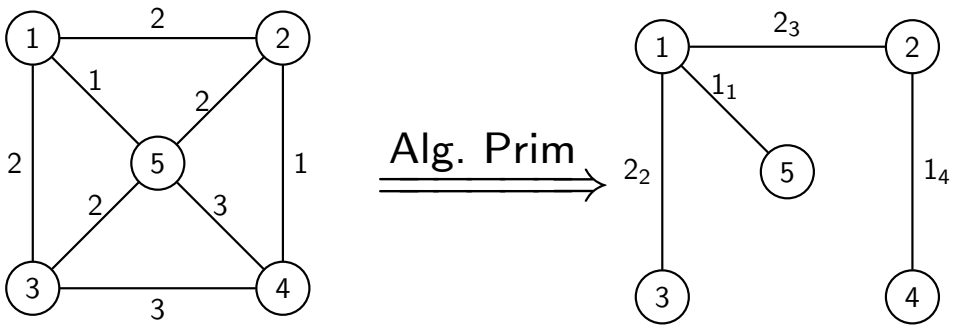
  - Cada uno es una selección de $n-1$ aristas ($n = |V|$) de manera que el subgrafo ha de seguir siendo conexo (y por tanto libre de ciclos)

Algoritmos de Prim y Kruskal

- El problema consiste en encontrar el árbol de recubrimiento de mínimo coste (puede haber más de uno).
- Existen al menos dos algoritmos voraces que lo resuelven:
  - algortimo de Prim
  - algortimo de Kruskal
- En ambos se van añadiendo arcos de uno en uno a la solución. La diferencia está en la forma de elegir los arcos a añadir.

Algoritmo de Prim

- Se mantiene un conjunto de vértices explorados (V')
- Se coge un vértice al azar y se añade al conjunto de explorados
- En cada paso:
  - buscar el arco de mínimo peso que va de un vértice explorado a uno que no lo está ($V - V'$)
    - Así se garantiza que no se incorporarán ciclos.
  - añadir el arco a la solución y el vértice a los explorados
- Termina cuando se han explorado todos los vértices ($V' = V$)



Un posible resultado partiendo del vértice 1. La arista (1,3) se ha escogido antes que la (1,2) y esto hace que la (2,4) se escoja en último lugar.

Proceso:

1. Parte del vértice 1. Se marca como visitado: $V' = \{1\}$
2. Selecciona la arista (1,5). $V' = \{1, 5\}$
3. Selecciona la arista (1,3) aunque también podría haber seleccionado la (1,2). $V' = \{1, 3, 5\}$. -la arista (2,4) no puede seleccionarse pues no cumple la condición-
4. Selecciona la arista (1,2). $V' = \{1, 2, 3, 5\}$
5. Selecciona la arista (2,4). Termina ($V' = V$)

Algoritmo de Prim

Algoritmo de Prim básico

1: Datos: $G = (V, A)$ ▷ Grafo no dirigido, conexo y ponderado
2: Resultado: $Sol \subseteq A$ ▷ Árbol de expansión mínimo: subconjunto de aristas
3: Auxiliar: $V' \subseteq V$ ▷ Conjunto de vértices visitados
4: Auxiliar: $(v, v') \in A$ ▷ Arista seleccionada
5: $Sol \leftarrow \emptyset$ ▷ Inicialmente el árbol está vacío
6: $v \leftarrow$ **elementoAleatorio**(V) ▷ Partimos de cualquier vértice de V
7: $V' \leftarrow \{v\}$ ▷ Se visita ese vértice
8: **while** $V' \neq V$ **do** ▷ Mientras no se hayan visitado todos los vértices
9: ▷ Selección: Arista de menor peso con un vértice visitado y el otro no
10: $(v, v') \leftarrow\in$ **aristaMenorPeso**(A) **con** $v \in V - V'$ **y** $v' \in V'$
11: $Sol \leftarrow Sol \cup \{(v, v')\}$ ▷ Se añade esa arista a la solución
12: $V' \leftarrow V' \cup \{v\}$ ▷ Se visita el vértice v
13: **end while**

Algoritmo de Prim

Algoritmo de Prim (ineficiente)

```
1  list<edge> prim( const Graph &g )
       {
2      int n = g.size();
3      vector<bool> visited( n, false
       );
4      list<edge> r;
5
6      edge e{-1,0};
7      for( int i = 0; i < n-1; i++ )
         {
8          visited[e.d] = true;
9
10         e = min_edge( g, visited )
       ;
11
12         r.push_back(e);
13
14     }
15     return r;
```

Estructuras de datos

```
1  using Graph = vector<vector<unsigned>>;
2
3  struct edge {
4      unsigned s;
5      unsigned d;
6  };
7
8  // Graph instantiation example
9  const unsigned INF
10     = numeric_limits<unsigned>::max();
11
12 Graph g{
13    { INF,   3,   1,   6, INF, INF },
14    {   3, INF,   5, INF,   3, INF },
15    {   1,   5, INF,   5,   6,   4 },
16    {   6, INF,   5, INF, INF,   2 },
17    { INF,   3,   6, INF, INF,   6 },
18    { INF, INF,   4,   2,   6, INF }
19 };
```

Algoritmo de Prim

Algoritmo de Prim (ineficiente)

```
1  edge min_edge(
2      const Graph& g,
3      const vector<bool> &visited
4
5  ) {
6      int n = g.size();
7      int min = numeric_limits<int>::max();
8      edge e;
9      for( int i = 0; i < n; i++ )
10         for( int j = 0; j < n; j++ )
11             if( visited[i] && !visited[j] )
12                 if( g[i][j] < min ) {
13                     min = g[i][j];
14                     e.s = i;
15                     e.d = j;
16                 }
17
18     return e;
19 }
```

- complejidad `min-edge`: $O(V^2)$
- complejidad de `Prim`: $O(V^3)$
- ¿se puede mejorar?

Algoritmo de Prim

Mejora:

- No hace falta incurrir en el coste cuadrático de recorrer todos los arcos cada vez.
- **Si cambia el mínimo a seleccionar es a causa del último vértice añadido**
- Hay que guardarse, para cada vértice no visitado, La arista mínima que incide en él desde un vértice visitado:
  - Se hace mediante un vector w de mínimos ya calculados que se actualiza cada vez que se visita un vértice,
  - **en w_i está el peso de la mejor arista que conecta un vértice visitado con el vértice i (aún sin visitar)**,
  - Además, hay que utilizar otro vector f:
    - El vector f es necesario para saber de qué arista se trata,
    - en f_i está el vértice origen de la arista representada por w_i.

Algoritmo de Prim

Algoritmo de Prim (con índices)

```
1  list<edge> prim( const Graph &g ) {
2      int n = g.size();
3      vector< bool > visited( n, false);                  // visited vertex
4      vector<int> w(n, numeric_limits<int>::max() );  // previous min's
5      vector<int> f(n);                                 // father of each min
6      list<edge> r;
7
8      edge e{-1,0};
9      for( int i = 0; i < n-1; i++ ) {
10
11         visited[e.d] = true;
12         update_idx( g, w, f, e.d );                  // update index
13
14         e = min_edge( g, w, f, visited );
15
16         r.push_back(e);
17     }
18     return r;
19 };
```

Algoritmo de Prim

Actualizar índices

```
1  void update_idx(
2      const  Graph &g,
3      vector<int>  &w,
4      vector<int>  &f,
5      int          nv
6  ) {
7
8      int n = g.size();
9      for( int j = 0; j < n; j++ )
10         if( w[j] > g[nv][j] ) {
11             w[j] = g[nv][j];
12             f[j] = nv;
13         }
14 }
```

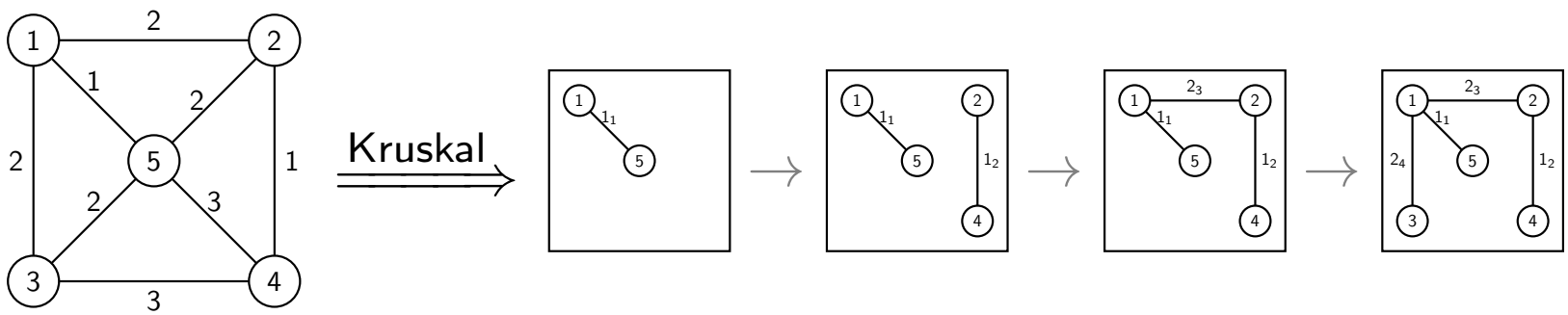
Buscar mejor arista

```
1  edge min_edge(
2      const Graph         &g,
3      const vector<int>   &w,
4      vector<int>         &f,
5      const vector<bool> &visited
6  ) {
7      int n = g.size();
8
9      int min = numeric_limits<int>::max();
10     edge e;
11     for( int j = 0; j < n; j++ ) {
12         if( !visited[j] && w[j] < min ) {
13             min = w[j];
14             e.s = f[j];
15             e.d = j;
16         }
17     }
18     return e;
19 }
```

- `min_edge` y `update_idx` $\in O(V)$
- `Alg. Prim` $\in O(V^2)$

Algoritmo de Kruskal

- Se seleccionan aristas por orden creciente de peso
  - Conviene realizar una ordenación previa de las aristas
- Una arista se incorpora a la solución si no forma ciclos.
- Termina cuando se han seleccionado $n-1$ aristas ($n = |V|$).



- Notar que en este algoritmo, a diferencia del alg. de Prim, su pueden ir formando distintos subárboles que inicialmente no están conexos pero, al final del proceso, el resultado será una única componente conexa.

Algoritmo de Kruskal

Algoritmo de Kruskal básico

1: Datos: $G = (V, A)$ ▷ Grafo no dirigido, conexo y ponderado
2: Resultado: $Sol \subseteq A$ ▷ Árbol de expansión mínimo
3: Auxiliar: $A' \subseteq A$ ▷ Conjunto ordenado de aristas
4: Auxiliar: $(u, v) \in A$ ▷ Arista seleccionada
5: $Sol \leftarrow \emptyset$ ▷ Comenzamos con un bosque vacío
6: $A' \leftarrow$ **ordenarPesosCreciente**(**A**) ▷ Preparar conjunto de aristas
7: **while** $|Sol| < |V| - 1 \wedge A' \neq \emptyset$ **do**
8: $(u, v) \leftarrow$ **primero**(A') ▷ Selección: Arista de mínimo peso aún sin considerar
9: **if noCreaCiclo**$((u, v), Sol)$ **then** ▷ Ha de conectar dos árboles existentes,
10: ▷ o bien, ser un nuevo árbol (nueva componente conexa)
11: $Sol \leftarrow Sol \cup \{(u, v)\}$ ▷ Se añade a la solución
12: **end if**
13: $A' \leftarrow A' - \{(u, v)\}$ ▷ En cualquier caso se descarta la arista
14: **end while**
15: **if** $|Sol| = |V| - 1$ **then**
16: **return** Sol
17: **else**
18: **return** $\emptyset$ ▷ Grafo no conexo: no existe árbol de recubrimiento
19: **end if**

Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

¿Cómo implementar la función **noCreaCiclo**$(\cdots)$?

- **Marcar los vértices visitados**, como se hace en el algoritmo de Prim, **no sirve** pues, aunque descarta ciclos, también descartaría aristas que conectan dos componentes conexas independientes.

Solución:

- Hacer uso de una estructura de datos denominada "conjuntos disjuntos" o "unión-búsqueda" (*Disjoint-set*, *Union-find*)

Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

Conjuntos disjuntos:

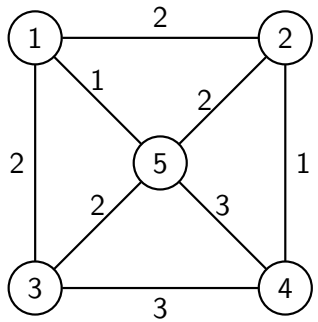
- También llamado TAD unión-búsqueda (*union-find*) por las operaciones que comprende.
- Se utiliza para mantener particiones de un conjunto de datos (en este caso los vértices).
- Operaciones que comprende el TAD:
  - Inicializar la partición: cada elemento en un bloque distinto
  - Poder unir dos bloques de la partición (`union`)
  - Saber a qué bloque pertenece un elemento (`find`)

Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

Aplicándolo al algoritmo de Kruskal:

1. Mantener una partición de los vértices.
   - Se puede hacer con un vector de etiquetas. Una por cada vértice.
   - Incialmente las n etiquetas son distintas lo que indicaría que cada vértice está en una componente conexas independiente al resto.
2. Mientras queda más de una componente conexa:
   - **Buscar** la arista de menor peso que una dos componentes distintas,
     - (Esto asegura que no habrán ciclos)
   - Añadir la arista a la solución
   - **Unir** las dos componentes conexas en una única componente conexa (poner a todos los vértices implicados la misma etiqueta)
3. El proceso termina cuando todos los vértices están en la misma componente conexa.

Algoritmo de Kruskal. Conjuntos disjuntos. Ejemplo



1. Mantener un vector s de etiquetas. Inicialmente todas distintas:
   $s = (1, 2, 3, 4, 5)$
   Esto indica que cada vértice está en una partición distinta, es decir, hay 5 componentes conexas (independientes)
2. Seleccionar la arista de mínimo peso, con la condición de que los extremos estén en particiones distintas. Incorporar la arista a la solución y fusionar ambas particiones (colocarles la misma etiqueta)
   1. arista $(1, 5)$; $s = (1, 2, 3, 4, 1)$
   2. arista $(2, 4)$; $s = (1, 2, 3, 2, 1)$
   3. arista $(1, 2)$; $s = (1, 1, 3, 1, 1)$
   4. arista $(1, 3)$; $s = (1, 1, 1, 1, 1)$
3. La selección de aristas se repite hasta que solo haya una componente conexa en s (se habrán seleccionado $n - 1$ aristas).

Algoritmo Kruskal. Complejidad temporal

- TAD *union-find*:
  - `union`: Reetiquetar todos los elementos de una de las particiones: $O(n)$
    - Con una implementación eficiente (con árboles) se consigue $O(\log n)$
  - `find`: Consultar la etiqueta de un vértice: $O(1)$

- Algoritmo de Kruskal:
  - Ordenar el conjunto E de aristas: $\Theta(E \log E)$
  - Realizar $|V| - 1$ operaciones `union`: $O(V \log V)$
  - Realizar $|V| - 1$ operaciones `find`: $O(V)$
  - Por lo tanto: $O(E \log E + V \log V)$

Algoritmo de Kruskal

```
1  list<edge> kruskal( const Graph &g ) {
2      struct node { int  w; edge e; };
3
4      int n = g.size();
5      list<edge> r;
6      disjoint_set s(n);
7
8      vector<node> v;
9      for( int i = 1; i < n; i++ )
10         for( int j = 0; j < i; j++ )
11             v.push_back({ g[i][j], {i, j} });
12
13     sort( v.begin(), v.end(), []( const node &n1, const node &n2) {
14         return n1.w < n2.w;
15     });
16
17     for( auto n : v ) {
18         if( s.find(n.e.s) != s.find(n.e.d) ) {
19             r.push_back(n.e);
20             s.merge( n.e.s, n.e.d );
21         }
22     }
23     return r;
24 }
```

Comparativa Prim *versus* Kruskal

- El algoritmo de Prim tiene una complejidad $O(V^2)$
- El algoritmo del Kruskal tiene una complejidad $O(E \log E)$
  - En el peor caso el número de aristas de un grafo es $E \in O(V^2)$
  - Por lo tanto, el algoritmo de Kruskal es:

$$O(E \log E) = O(V^2 \log V^2) = O(V^2 \log V)$$

¿Cuándo usar cada algoritmo?

- Si el grafo tiende a estar completo el algoritmo de Prim es mejor.
- El algoritmo de Kruskal es mejor cuando el grafo es disperso (pocas aristas).

El fontanero diligente

- Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.
- En otras palabras, si llamamos E_i a lo que espera el cliente i-ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E(n) = \sum_{i=1}^{n} E_i$$

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i.
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí.
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.
- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} b_{ij}$$

- Diremos que una asignación es óptima si es de mínimo coste.