

# Biblioteca de Grafos Genérica en Eiffel

Diego Lis

20 de junio de 2004

## Índice

<b>1. Resumen</b>	<b>3</b>
<b>2. Introducción</b>	<b>3</b>
2.1. Problema . . . . .	3
2.2. Decisiones . . . . .	3
2.3. Notación . . . . .	3
2.4. Código . . . . .	4
2.5. Gráficos . . . . .	4
<b>3. Genericidad vs. eficiencia</b>	<b>4</b>
3.1. Vértices numerados . . . . .	5
3.2. Vértices genéricos . . . . .	5
3.3. Vértices hashables . . . . .	5
3.4. Vértices encapsulados . . . . .	5
<b>4. Vértices</b>	<b>5</b>
4.1. Vértice . . . . .	5
4.2. Colección de vértices . . . . .	6
<b>5. Aristas</b>	<b>6</b>
5.1. Arista . . . . .	6
5.2. Colección de aristas . . . . .	7
5.2.1. Matriz de adyacencia . . . . .	7
5.2.2. Listas de adyacencia . . . . .	7
<b>6. Grafos</b>	<b>7</b>
6.1. Grafos No Dirigidos . . . . .	9
6.2. Grafos Dirigidos . . . . .	10
<b>7. Visitadores</b>	<b>10</b>
7.1. Visitadores de adyacencia . . . . .	11
7.2. Visitadores simples . . . . .	11
7.3. Visitadores hamiltonianos . . . . .	11
7.4. Esquema de Clases . . . . .	12
7.5. Cuestiones de implementación . . . . .	12

<b>8. Pesos</b>	<b>12</b>
8.1. Camino más corto . . . . .	13
8.1.1. Dijkstra . . . . .	13
8.1.2. Floyd . . . . .	15
8.2. Árboles de expansión minimales . . . . .	16
<b>9. Genericidad vs. facilidad de uso</b>	<b>17</b>
9.1. Un ejemplo . . . . .	17
9.2. Entrada - Salida . . . . .	19
<b>10. Conclusión</b>	<b>19</b>
10.1. Genericidad . . . . .	19
10.2. Eficiencia . . . . .	19
10.3. Corrección . . . . .	20
10.4. Funcionalidad . . . . .	20
10.5. Extensibilidad . . . . .	20
10.6. Facilidad de uso . . . . .	20
<b>11. Agradecimientos</b>	<b>20</b>

# 1. Resumen

Eiffel es un lenguaje orientado a objetos puro, con chequeo de tipos estático, que soporta *Diseño por Contrato*, herencia múltiple, garbage collector, con una sintaxis sencilla y transparente ([1]).

Muchos problemas de la teoría de grafos no pueden resolverse sin la ayuda de la computadora. De hecho, ha sido de ayuda significativa en la solución de problemas teóricos (como el teorema de los cuatro colores) y también para aplicaciones en otras disciplinas ([2]): ingeniería eléctrica, ingeniería industrial, física, química, biología, ingeniería civil, ciencias sociales, economía y logística, geografía, arquitectura, juegos y ciencia de la computación.

La presente biblioteca de grafos, se ha implementado en Eiffel y compatibiliza exitosamente la genericidad, la eficiencia y la facilidad de uso.

Además, hemos mejorado el algoritmo de Dijkstra para el caso en el que se realicen muchas consultas seguidas, mejorando hasta un 50 % la eficiencia total.

## 2. Introducción

### 2.1. Problema

El problema consiste en construir una biblioteca de grafos con los siguientes requerimientos:

1. Genericidad: cada concepto debe ser lo más general posible. Es decir, no se deben introducir limitaciones por cuestiones de implementación o eficiencia.
2. Eficiencia: debe implementar cada algoritmo en forma eficiente.
3. Corrección: todo lo que realice la biblioteca debe ser correcto.
4. Funcionalidad: debe implementar, al menos, los algoritmos más comúnmente usados.
5. Extensibilidad: debe permitir, en forma sencilla, el agregado de nueva funcionalidad.
6. Facilidad de uso: La interfaz de cada módulo debe ser sencilla, fácil de aprender y de recordar.

Vemos que los dos primeros requerimientos pueden ser contradictorios. La genericidad puede tener un costo en la eficiencia. El desafío será lograr ambos aspectos.

### 2.2. Decisiones

Para implementar esta biblioteca, decidimos utilizar el paradigma orientado a objetos, porque nos permite, más naturalmente, lograr los requerimientos mencionados.

El lenguaje que elegimos es Eiffel, por su elegancia, sencillez, transparencia y mecanismos de especificación y documentación.

La biblioteca está escrita en inglés, para facilitar su reutilización en otros proyectos.

### 2.3. Notación

Utilizaremos las siguientes convenciones:

- V: conjunto de vértices del grafo.
- E: conjunto de aristas del grafo.

## 2.4. Código

Se mostrará partes del código de la biblioteca. En la mayoría de los casos, consideramos suficiente mostrar la especificación de cada rutina, es decir:

- Nombre
- Parámetros, si corresponde
- Valor devuelto, si corresponde
- Comentario
- Precondición
- Postcondición

Una rutina se vería así:

```
<nombre> [lista_parametros] [: <valor_devuelto>]
    -- <comentario>
require
    <precondicion>
ensure
    <postcondicion>
```

Nota:

- [ ]: campo opcional.
- <>: reemplace por lo que corresponda.

Otras veces, sin embargo, fue necesario incluir el cuerpo de la rutina, aunque decidimos hacerlo en pseudocódigo. La sintaxis es muy similar a la de Eiffel, con algunos agregados para una mayor claridad de este informe.

## 2.5. Gráficos

También necesitaremos mostrar grafos de dependencias entre las clases del sistema. La convención utilizada aquí es:

- Las flechas sólidas indican “es un caso particular de” o “hereda de”.
- Las flechas punteadas indican “utiliza a” o “es cliente de”.

## 3. Genericidad vs. eficiencia

Como hemos mencionado, queremos lograr genericidad sin perder eficiencia. Aquí describiremos diferentes soluciones a este problema, y finalmente, cuál elegimos. Para hacer una comparación entre las alternativas, trabajaremos con el siguiente ejemplo: agregar una arista a un grafo representado por matriz de adyacencia.

### 3.1. Vértices numerados

Normalmente, los vertices de un grafo se representan con un número entero entre 0 y  $n - 1$ , o algo similar. Esto tiene la ventaja de ser eficiente. En nuestro ejemplo, agregar una arista es  $O(1)$ . Sin embargo, posee dos desventajas que atentan contra la genericidad:

- Impide la representación de otro tipo de vértices, como letras, cadenas, ciudades o personas.
- Se estaría dando un orden explícito a los vértices. En general, un grafo contiene un conjunto no ordenado de vértices, más allá de que en la implementación se los pueda ordenar por cuestiones de eficiencia.

### 3.2. Vértices genéricos

La alternativa opuesta a la anterior, es permitir cualquier tipo de vértices. Esto aumenta la genericidad, pero disminuye notablemente la eficiencia. En nuestro ejemplo, agregar una arista es  $O(|V|)$ .

### 3.3. Vértices hashables

Una solución intermedia es permitir que los vértices sean cualquier entidad hashable, es decir, que tenga un código de hash. Así, internamente se accedería a los vértices a través de dicho código. La eficiencia podría verse un poco afectada, pero en el caso promedio, seguirá siendo  $O(1)$ .

### 3.4. Vértices encapsulados

Esta solución consiste en crear una estructura de datos que contenga el vértice en sí (que puede ser una letra, cadena, ciudad, persona, etc) y toda otra información necesaria para acceder al mismo en tiempo constante. Posiblemente, esa información consista en la posición que ocupa en el grafo. De esta forma, agregar una arista en nuestro ejemplo será  $O(1)$ . Este dato estará oculto, pues el orden que le damos a los vértices es una cuestión de implementación.

Esta es la solución que elegimos, pues brinda la máxima genericidad y una óptima eficiencia.

## 4. Vértices

En esta sección veremos dos estructuras de datos: el vértice y la colección de vértices.

### 4.1. Vértice

En la sección anterior vimos la solución que elegimos para representar los vértices. Ahora vamos a verla más en detalle. Un *vértice* tiene los siguientes elementos:

- **item**: el contenido del vértice
- **index**: posición en el grafo

Queremos que **item** pueda ser de cualquier tipo. Una solución es declararlo de tipo **ANY**. Sin embargo, esto tiene dos problemas:

- El tipo **ANY** no tiene ninguna operación, así que no podremos hacer mucho con los vértices del grafo.
- Generalmente uno utiliza grafos de ciudades, grafos de personas, etc. Es decir, uno limita el tipo de los vértices.

Para resolver ambos problemas, aprovechamos el mecanismo de clases genéricas de Eiffel para que `item` sea de cualquier tipo, pero que el tipo sea declarado en tiempo de compilación.

Así, un grafo `g` podrá ser declarado de tipo `GRAPH[INTEGER]` o `GRAPH[STRING]`, los cuales tendrán vértices de tipo `VERTEX[INTEGER]` o `VERTEX[STRING]`, respectivamente.

El elemento `index` sólo se exporta a la clase `GRAPH`. Es decir, se oculta este detalle de implementación a todas las otras clases.

## 4.2. Colección de vértices

Una *colección de vértices* tiene las siguientes operaciones:

```
count: INTEGER
    -- cantidad de vertices

is_empty: BOOLEAN
    -- no hay vertices

add (k: K)
    -- crea un vertex con el contenido 'k' y lo agrega al grafo
require
    k /= Void
ensure
    last_vertex_added.item = k

last_vertex_added: VERTEX[K]
    -- ultimo vertice agregado a la coleccion

get_new_iterator: ITERATOR[VERTEX[K]]
    -- itera sobre los vertices en la coleccion

has_vertex (v: VERTEX[K]): BOOLEAN
    -- esta 'v' en el grafo?
```

Hemos implementado esta estructura con un `SET`, pues tanto `add` como `has_vertex` son  $O(1)$ .

Veremos más adelante que los grafos serán casos particulares de colecciones de vértices, y por lo tanto, contarán con todas estas operaciones.

## 5. Aristas

En esta sección veremos dos estructuras de datos: la arista y la colección de aristas.

### 5.1. Arista

Una *arista* es un par ordenado de vértices. En otras palabras, contiene dos vértices, uno de los cuales es el origen (`source`) y el otro el destino (`target`).

Una *arista con peso* es una arista que además tiene un valor numérico asociado: `weight`.

En realidad, estas estructuras no se usan intensivamente en la biblioteca. Como veremos, la colección de aristas y los grafos trabajan con pares de vértices, más que con aristas para lograr una mayor facilidad de uso.

Las aristas van a aparecer recién cuando implementemos los árboles de expansión minimales.

## 5.2. Colección de aristas

Una *colección de aristas* tiene las siguientes operaciones:

```
add (v1, v2: VERTEX[K]) is
    -- agrega una arista entre 'v1' y 'v2'
require
    not has (v1, v2)
ensure
    has (v1, v2)

remove (v1, v2: VERTEX[K]) is
    -- quita la arista entre 'v1' y 'v2'
require
    has (v1, v2)
ensure
    not has (v1, v2)

has(v1, v2: VERTEX[K]): BOOLEAN is
    -- hay una arista entre 'v1' y 'v2'?

at (v: VERTEX[K]): ITERATOR[VERTEX[K]] is
    -- {w: has('v', w)}
```

### 5.2.1. Matriz de adyacencia

Una de las implementaciones de la colección de aristas es la *matriz de adyacencia*. Esta consiste en una matriz booleana de  $|V| \times |V|$ . La celda  $(i, j)$  indica si hay una arista entre  $i$  y  $j$ .

Agregar, quitar y buscar una arista son  $O(1)$ , y recorrer los vecinos de un vértice es  $O(|V|)$ . El almacenamiento es  $O(|V|^2)$ .

### 5.2.2. Listas de adyacencia

Otra implementación de la colección de aristas es mediante *listas de adyacencia*. Para cada vértice, hay asociada una lista de sus vecinos.

Agregar y quitar una arista son  $O(1)$ , buscar una arista es  $O(k)$  y recorrer los vecinos de un vértice es  $O(k)$ , donde  $k$  es el grado del vértice. El almacenamiento es  $O(|E|)$ .

## 6. Grafos

Un *grafo* es una colección de vértices que además tiene las siguientes operaciones:

```
add_edge(v1, v2: VERTEX[K])
    -- agrega una arista entre 'v1' y 'v2'. Si ya existe, no hace nada.
require
    is_valid_edge (v1, v2)
ensure
    are_adjacents(v1, v2)
```

```

remove_edge(v1, v2: VERTEX[K])
    -- quita la arista entre 'v1' y 'v2'
ensure
    not are_adjacents(v1, v2)

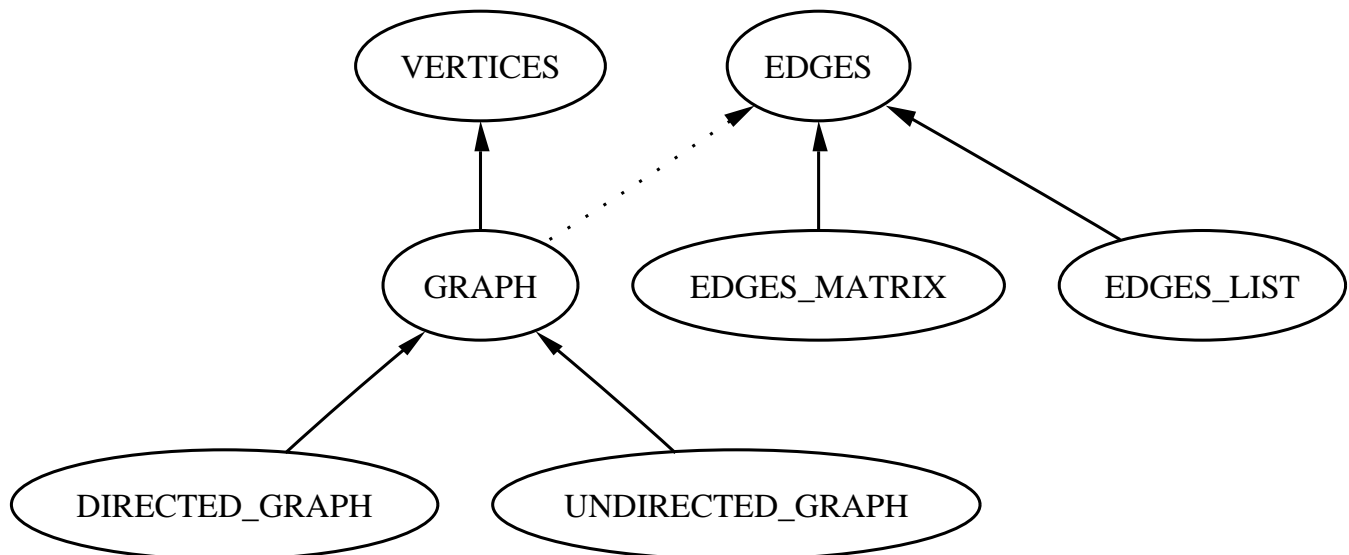
are_adjacents(v1, v2: VERTEX[K]): BOOLEAN
    -- hay una arista entre 'v1' y 'v2'?
ensure
    Result implies is_valid_edge (v1, v2)

adjacents(v: VERTEX[K]): ITERATOR[VERTEX[K]]
    -- { w : are_adjacents (v, w) }
require
    has_vertex(v)

is_valid_edge (v1, v2: VERTEX[K]): BOOLEAN
    -- es ('v1', 'v2') una arista valida? Esto permite establecer restricciones.
    -- Por ejemplo, en un grafo bipartito no se pueden unir dos vertices de
    -- la misma parte. En un arbol, no se puede agregar una arista si genera
    -- un ciclo.
ensure
    Result implies has_vertex (v1) and has_vertex (v2)

```

No confundir *grafo* con *colección de aristas*. Pese a que poseen operaciones similares, un grafo es un caso particular de colección de vértices, y se implementa usando una colección de aristas. Este diseño permite una mayor genericidad y extensibilidad, pues la colección de aristas puede estar implementada con matriz o listas de adyacencia. En el futuro puede implementarse una nueva colección de aristas (por ejemplo, con lista de aristas), y puede usarse en un grafo sin necesidad de modificar ningún código. A continuación se muestra la relación entre ambas clases:



A partir de las operaciones básicas se pueden definir nuevas operaciones para un grafo:

```

valency (v: VERTEX[K]): INTEGER
    -- cantidad de adjacents a 'v'

```



```

require
    has_vertex(v)

is_connected: BOOLEAN
    -- es un grafo conexo ?

has_cycle: BOOLEAN
    -- contiene un ciclo ?

is_tree: BOOLEAN
    -- es un arbol?

is_complete: BOOLEAN
    -- es un grafo completo?

is_regular: BOOLEAN
    -- es un grafo regular?

coloring_number_greedy: INTEGER
    -- corre 'greedy' sobre el grafo

is_bipartite: BOOLEAN
    -- es un grafo bipartito?

exists_path(v1, v2: VERTEX[K]): BOOLEAN
    -- hay un camino desde 'v1' a 'v2'
require
    has_vertex(v1)

distance (v1, v2: VERTEX[K]): INTEGER
    -- longitud del camino minimo desde 'v1' a 'v2'
require
    exists_path (v1, v2)

eulerian_path (v1, v2: VERTEX[K]): BOOLEAN is
    -- hay un recorrido euleriano de 'v1' a 'v2'
require
    has_vertex (v1)
    has_vertex (v1)
ensure
    Result implies is_connected
    Result and v1 = v2 implies valency (v1) \ 2 = 0
    Result and v1 /= v2 implies valency (v1) \ 2 = 1 and valency (v2) \ 2 = 1

```

## 6.1. Grafos No Dirigidos

Un *grafo no dirigido* es un grafo que tiene además el siguiente invariante:

- $(v1, v2) \in E \Rightarrow (v2, v1) \in E$

## 6.2. Grafos Dirigidos

Un *grafo dirigido* es un grafo que tiene además:

```
backward_adjacents(v: VERTEX[K]): ITERATOR[VERTEX[K]] is
  -- { w : are_adjacents (w, v) }
require
  has_vertex(v)
```

Ahora puede definirse:

```
topological_order: ITERATOR[VERTEX[K]]
  -- un orden topologico
require
  not has_cycle
```

## 7. Visitadores

Hay muchos algoritmos de grafos que consisten en recorrer los vértices en algún orden y hacer algo con ellos. Por ejemplo, para ver si un grafo es conexo, se recorre los vértices en modo BFS (Breath First Search) o DFS (Depth First Search) y se cuentan los vértices visitados. Si es igual a  $|V|$  entonces el grafo es conexo. Como este hay muchos algoritmos que utilizan BFS, DFS o algún otro recorrido.

En ese caso, fue necesario recorrer todos los vértices hasta terminar. Sin embargo, en otros, esto no será necesario. Por ejemplo, para ver si dos vértices están conectados entre sí, se recorre en modo BFS partiendo de uno de los vértices, y el algoritmo para cuando encuentra el otro. En este caso, no fue necesario recorrer todos los vértices.

Por esta razón, decidimos implementar BFS como un iterador sobre los vértices. Luego generalizando, definimos el concepto de visitador. Un *visitador* es un iterador sobre los vértices de un grafo.

Los elementos que definen un visitador son:

- **graph**: es el grafo que se quiere visitar.
- **first**: es el primer vértice que se va a visitar.
- **dispenser**: es una estructura que tiene las siguientes operaciones:
  - **add**: agregar un elemento
  - **item**: elemento actual
  - **remove**: quitar el elemento actual
  - **is\_empty**: esta vacío

Como puede observarse, dispenser es una generalización de pila y de cola. Vemos que la mayoría de los visitadores necesitan de un dispenser donde ir almacenando los vertices recorridos y realizar alguna operación con cada uno de ellos.

- **childs**: es una función que toma un vértice y devuelve un conjunto de vértices
- **action**: es una rutina que toma como parámetro un vértice.

- **test**: es un predicado sobre un vértice.

Definidos estos elementos del visitador, su comportamiento es el siguiente:

1. Se agrega **first** a **dispenser**
2. Mientras **dispenser** no esté vacío:
  - a) Sea **v** el vértice actual en **dispenser**
  - b) Para todos los vértices de **childs (v)** que cumplan el predicado **test**, se agregan al **dispenser** y se ejecuta **action**.
  - c) Se quita **v** de **dispenser**

En pseudocódigo, es:

```

from
    dispenser.add (first)
until
    dispenser.is_empty
loop
    v := dispenser.item
    dispenser.remove
    if test (v) then
        for w in childs (v) do
            dispenser.add (w)
        end
        action (v)
    end
end
end

```

## 7.1. Visitadores de adyacencia

Para la mayoría de los visitadores, **childs (v)** son los vecinos de **v**. A estos los llamaremos *visitadores de adyacencia*.

## 7.2. Visitadores simples

Cuando un visitador pasa a lo sumo una vez por cada vértice, decimos que es un *visitador simple*. Un visitador simple es un visitador que tiene además:

- **visited**: conjunto de vértices ya visitados.

Luego definimos:

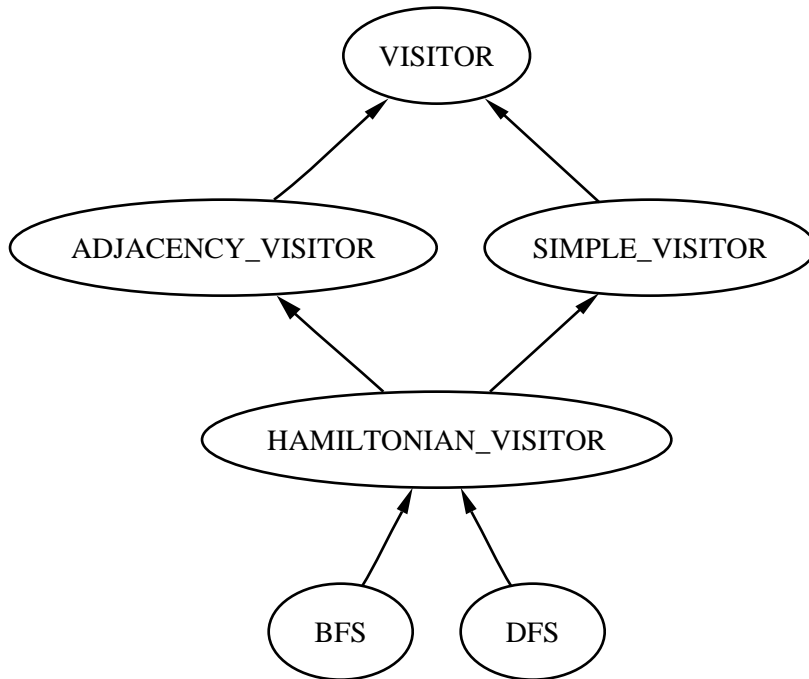
- **action (v)**: agregar **v** a **visited**
- **test (v)**: está **v** en **visited**?

## 7.3. Visitadores hamiltonianos

BFS y DFS son visitadores de adyacencia simples. A estos los llamaremos *visitadores hamiltonianos*.

Ahora, BFS y DFS son visitadores hamiltonianos cuyos **dispenser** son una cola y una pila respectivamente.

## 7.4. Esquema de Clases



## 7.5. Cuestiones de implementación

Hemos implementado la mayoría de las operaciones de *grafo* utilizando visitantes:

- **is\_connected**: corre BFS y cuenta la cantidad de vértices visitados. Es conexo si es igual a  $|V|$ .
- **has\_cycle**: corre un visitador de adyacencia. Cuando encuentra un vértice ya visitado, es porque hay un ciclo.
- **is\_bipartite**: similar a **has\_cycle**, pero detecta ciclos de longitud impar.
- **exists\_path**: corre BFS desde uno de los vértices y se detiene al llegar al otro.
- **distance**: igual que en **exists\_path**, pero contando la cantidad de pasos.

## 8. Pesos

Hasta ahora hemos hablado de grafos sin pesos. Vamos a definir un *grafo con pesos* como un grafo que tiene asociado a cada arista un valor numérico. La genericidad provista por Eiffel nos ha permitido que el tipo de este valor pueda ser muy general: enteros, punto flotante y en general cualquier anillo con un orden total.

Un grafo con pesos tiene las siguientes operaciones:

```
add_edge (v1, v2: VERTEX[K]; w: W)
  -- agrega una arista entre 'v1' y 'v2' de peso 'w'
ensure
  weight (v1, v2) = w

weight(v1, v2: VERTEX[K]): W
```

```

    -- peso de la arista entre 'v1' y 'v2'.
require
    has_vertex (v1)
    has_vertex (v2)
    are_adjacents (v1, v2)

shortest_path (v1, v2: VERTEX[K]): W
    -- Calcula el camino mas corto de 'v1' a 'v2'
    -- utilizando los calculos previamente realizados
    -- para mayor eficiencia
require
    exists_path (v1, v2)

all_paths
    -- Calcula el camino mas corto entre cada par de vertices

all_paths_from_vertex (v: VERTEX[K])
    -- Calcula el camino mas corto de 'v' a todos los vertices

kruscal: like Current
    -- obtiene un arbol de expansion minimal usando Kruscal
require
    is_connected
ensure
    Result.count = count
    Result.is_tree

prim: like Current
    -- obtiene un arbol de expansion minimal usando Prim
require
    is_connected
ensure
    Result.count = count
    Result.is_tree

```

Hemos implementado el grafo con pesos con una matriz de  $W$ , donde  $W$  es el tipo de los pesos. Así, la celda  $(i, j)$  es el peso de la arista de  $i$  a  $j$ . Esto tiene una óptima eficiencia en tiempo para acceder y cambiar los pesos.

## 8.1. Camino más corto

### 8.1.1. Dijkstra

Para calcular el camino más corto entre dos vértices  $x$  e  $y$ , utilizamos el algoritmo de Dijkstra. En pseudocódigo:

```

dijkstra (x, y: VERTEX[K]): W is
    -- W es el tipo de los pesos. Puede ser INTEGER, FLOAT, DOUBLE, etc.
require
    exists_path (x, y)
local

```

```

closest: SET[VERTEX[K]] -- vertices cuya distancia minima desde 'x' es conocida
distances: ARRAY2[W] -- distancias ya calculadas
min: W
min_vertex: VERTEX[K]
do
  create closest.make_empty
  create distances.make (1, count, 1, count)
from
  distances.put (0, x, x)
  closest.add (x)
until
  closest.has (y)
loop
  min := infinity
  for u in closest do
    for v in adjacents (u) do
      if not closest.has (v) then
        -- para todo par (u, v) tal que
        -- (u, v) es una arista
        -- 'u' en 'closest' y 'v' fuera de 'closest'
        if min > distances.item (x, u) + weight (u, v) then
          min := distances.item (x, u) + weight (u, v)
          min_vertex := v
        end
      end
    end
  end
  closest.add (min_vertex)
  distances.put (min, x, v)
end
Result := distances.item (x, y)
end

```

Vemos que la eficiencia de este algoritmo puede mejorarse si utilizamos los cálculos que hemos realizado anteriormente. Por ejemplo, si hemos calculado el camino más corto de  $x$  a  $y$ , el cual pasa por  $z$ , habremos calculado el camino más corto de  $x$  a  $z$  y de  $z$  a  $y$ . Por otra parte, podremos inicializar **distances** con todas las distancias ya calculadas.

De hecho, **distances** será un atributo (oculto) de grafo con pesos. De esta manera, luego de ejecutar **dijkstra** para un par de vértices, permanecerá hasta la próxima vez que se ejecute.

Nos servirá también para inicializar **closest**. Agregaremos a este conjunto, todos los vértices cuyas distancias ya están calculadas.

El pseudocódigo quedaría modificado de la siguiente manera:

```

distances: ARRAY2[W] -- distancias ya calculadas (ahora como atributo)

dijkstra_mejorado (x, y: VERTEX[K]): W is
  -- W es el tipo de los pesos. Puede ser INTEGER, FLOAT, DOUBLE, etc.
require
  exists_path (x, y)
local

```

```

closest: SET[VERTEX[K]] -- vertices cuya distancia minima desde 'x' es conocida
min: W
min_vertex: VERTEX[K]
do
  create closest.make_empty
  from
    for u in vertices do
      if distances.item (x, u) < infinity then
        -- agrego a 'closest' todos los vertices
        -- cuya distancia ya conozco
        closest.add (u)
      end
    end
  until
    closest.has (y)
  loop
    min := infinity
    for u in closest do
      for v in adjacents (u) do
        if not closest.has (v) then
          -- para todo par (u, v) tal que
          -- (u, v) es una arista
          -- 'u' en 'closest' y 'v' fuera de 'closest'
          if min > distances.item (x, u) + weight (u, v) then
            min := distances.item (x, u) + weight (u, v)
            min_vertex := v
          end
        end
      end
    end
    closest.add (min_vertex)
    distances.put (min, x, v)
  end
  Result := distances.item (x, y)
end

```

Esta es la versión que hemos utilizado. Para finalizar, debemos observar que cada vez que se realice algún cambio en los pesos del grafo, debemos reinicializar **distances**. Hemos comprobado en nuestras pruebas que el ahorro de tiempo puede ser muy significativo (hasta un 50 %).

También implementamos una rutina **all\_paths\_from\_vertex**, que toma un vértice y corre **dijkstra\_mejorado**, con la diferencia de que no se detiene, si no que calcula la distancia a todos los otros vértices. Esto puede ser útil cuando se va a calcular la distancia de un vértice a muchos otros.

### 8.1.2. Floyd

Lo anterior nos sugiere la posibilidad de implementar una rutina **all\_paths** que calcule la distancia entre todo par de vértice, usando el algoritmo de Floyd.

Esto será un poco más eficiente que correr Dijkstra  $|V|$  veces.

En pseudocódigo es:

**all\_paths** is

```

local
  i, j, k: VERTEX[K]
do
  for k in vertices do
    for i in vertices do
      for j in vertices do
        if distances.item (i, j) > distances.item (i, k) + distances.item (k, j) then
          distances.item (i, j) := distances.item (i, k) + distances.item (k, j)
        end
      end
    end
  end
end
end

```

## 8.2. Árboles de expansión minimales

Un árbol de expansión minimal de un grafo es un subgrafo que contiene todos los nodos del grafo, es conexo y no tiene ciclos.

Implementamos un algoritmo que, dado un grafo, calcula un árbol de expansión minimal: Kruscal. En Kruscal, utilizamos dos estructuras de datos:

- FOREST: para mantener las componentes conexas (implementado con *union-find*).
- PRIORITY\_QUEUE: para iterar sobre las aristas en orden.

Mostramos el códigos respectivo:

```

kruscal: like Current is
  -- obtiene un arbol de expansion minimal usando Kruscal
require
  is_connected
local
  ii: ITERATOR[WEIGHTED_EDGE[K, W]]
  forest: FOREST
  v1, v2: VERTEX[K]
do
  create Result.from_collection (Current)
  from
    create forest.make (count)
    ii := get_new_iterator_on_ordered_edges
  until
    ii.is_off or forest.count = 1
  loop
    v1 := ii.item.source
    v2 := ii.item.target
    if not forest.are_connected (v1.index, v2.index) then
      Result.add_edge (v1, v2, ii.item.weight)
      forest.union (v1.index, v2.index)
    end
    ii.next
  end
end

```



```

    check forest.count = 1 end
ensure
    Result.count = count
    Result.is_tree
end

```

## 9. Genericidad vs. facilidad de uso

Hasta aquí hemos logrado una gran genericidad. En esta sección, aumentaremos la facilidad de uso sin perder esa genericidad.

### 9.1. Un ejemplo

Vamos a trabajar con el siguiente ejemplo. Queremos utilizar esta biblioteca para realizar las siguientes operaciones:

- Contruir un grafo no dirigido con pesos con los siguientes parámetros:
  - Tipo de los nodos: STRING
  - Representación: Matriz de adyacencia
  - Tipo de los pesos: INTEGER
- Agregar vértices a ese grafo
- Unir algunos de ellos con aristas
- Preguntar si es conexo
- Preguntar el camino más corto entre algunos pares de vértices
- Pedir el árbol de expansión minimal

Para ver evaluar la sencillez de la biblioteca, mostraremos el código necesario para ejecutar estas operaciones:

```

local
    -- grafo no dirigido con pesos, nodos=STRING, aristas=MATRIX, pesos=INTEGER
    g: UNDIRECTED_WEIGHTED_GRAPH[STRING, EDGES_MATRIX[STRING], INTEGER]
    -- vertices
    v1, v2, v3, v4: VERTEX[STRING]
do
    -- crear grafo
    create g.make
    -- agregar nodos
    g.add ("Cordoba")
    v1 := g.last_vertex_added
    g.add ("Rosario")
    v2 := g.last_vertex_added
    g.add ("Buenos Aires")
    v3 := g.last_vertex_added
    g.add ("Parana")
    v4 := g.last_vertex_added
    -- agregar aristas: (origen, destino, peso)

```

```

g.add_edge (v1, v2, 10)
g.add_edge (v1, v3, 15)
g.add_edge (v2, v3, 5)
g.add_edge (v3, v4, 20)
  -- consultas
<variable_booleana> := g.is_connected
<variable_entera> := g.shortest_path (v1, v3)
<variable_entera> := g.shortest_path (v2, v3)
<grafo> := g.kruscal
end

```

Dada la genericidad logrado con este mecanismo, pensamos que la facilidad de uso es razonable. Sin embargo, puede mejorarse aún más.

Hemos agregado a vértice un nuevo constructor:

```

from_parent (g: VERTICES[K]; k: K)
  -- crea el vertice con contenido 'k' y lo agrega a 'g'
ensure
  item = k

```

El código para agregar los nodos quedaría así:

```

  -- agregar nodos
create v1.from_parent (g, "Cordoba")
create v2.from_parent (g, "Rosario")
create v3.from_parent (g, "Buenos Aires")
create v4.from_parent (g, "Parana")

```

A veces sólo necesitamos grafos cuyos nodos sean los enteros del 1 al  $n$ , y con pesos enteros. En este caso, la facilidad de uso debería ser mucho mayor. Para lograr esto, introducimos el concepto de *grafo simple*. Vamos a ver como queda nuestro código:

```

local
  -- grafo simple
  g: UNDIRECTED_WEIGHTED_SIMPLE_GRAPH[SIMPLE_EDGES_MATRIX]
do
  -- crear grafo con nodos: 1, 2, 3, 4
  create g.simple_make (1, 4)
  -- agregar aristas
  g.simple_add_edge (1, 2, 10)
  g.simple_add_edge (1, 3, 15)
  g.simple_add_edge (2, 3, 5)
  g.simple_add_edge (3, 4, 20)
  -- consultas
  <variable_booleana> := g.is_connected
  <variable_entera> := g.shortest_path (1, 3)
  <variable_entera> := g.shortest_path (2, 3)
  <grafo> := g.kruscal
end

```

Este ejemplo muestra que hemos logrado un alto grado de facilidad en el uso de esta biblioteca.

## 9.2. Entrada - Salida

Nuestra biblioteca deberá brindar un mecanismo automático para leer y escribir grafos por entrada y salida estándar. El formato que hemos elegido es el provisto por Graphviz ([7]).

Los ejemplos anteriores podrían haberse construido a partir de las siguientes entradas:

```
graph G {
  1 [label = "Cordoba"];
  2 [label = "Rosario"];
  3 [label = "Buenos Aires"];
  4 [label = "Parana"];
  1 -> 2 [label = 10];
  1 -> 3 [label = 15];
  2 -> 3 [label = 15];
  3 -> 4 [label = 20];
}

graph G2 {
  1 -> 2 [label = 10];
  1 -> 3 [label = 15];
  2 -> 3 [label = 15];
  3 -> 4 [label = 20];
}
```

Por otra parte, al permitir este formato de salida, puede utilizarse para graficar automáticamente con Graphviz.

## 10. Conclusión

En esta sección veremos si hemos cumplido con los requerimientos planteados al comienzo de este trabajo.

### 10.1. Genericidad

Este tema ha sido parcialmente tratado en secciones anteriores. Ahora ampliaremos un poco.

Ya vimos que para los vértices tenemos la máxima generalidad posible. Pueden ser de cualquier tipo, y han sido encapsulados para agregar información de implementación para mayor eficiencia.

Los grafos pueden utilizarse con cualquier implementación de *colección de aristas* que se halla escrito.

La genericidad del concepto de *visitador* permite sencillas implementaciones de nuevos algoritmos.

La genericidad del concepto de *dispenser* permite sencillas implementaciones de diversas estructuras de datos como pila, cola, cola de prioridades, para usar en los visitadores.

La genericidad del tipo de los pesos en un grafo, nos permite trabajar con enteros, punto flotante, y en general con cualquier anillo con un orden total.

### 10.2. Eficiencia

Cada algoritmo se ha implementado pensando en su eficiencia teórica. Es decir, si se espera que BFS implementado con listas de adyacencia tenga una eficiencia  $O(|E|)$ , a eso hemos apuntado.

Además, los algoritmos de cálculo de caminos más cortos (shortest path), se han implementado buscando mejorar la eficiencia utilizando los cálculos previamente realizados.

### 10.3. Corrección

Las aserciones de Eiffel nos han permitido chequear en alto grado la corrección de los algoritmos. Por otra parte, una buena modularización nos ha permitido mantener interfaces claras y cortas, lo cual se traduce en una mayor claridad de código, y esto se refleja en la corrección.

### 10.4. Funcionalidad

Hemos implementado algunos de los algoritmos más comunes en el área de grafos:

- Propiedades intrínsecas (`is_connected`, `exists_path`, etc)
- Visitadores
- Caminos más cortos.
- Árboles de expansión minimales.
- Greedy para coloreo.

Han quedado para futuras extensiones:

- Flujos
- Matching

### 10.5. Extensibilidad

La extensibilidad se ha logrado principalmente en los siguientes aspectos:

- Nuevas implementaciones de *colección de vértices*: con arreglos, listas, árboles u otra estructura de datos.
- Nuevas implementaciones de *colección de aristas*: con lista de aristas, etc.
- Nuevos tipos de grafos: network, árboles, etc.
- Nuevos visitadores: la estructura de un visitador es muy flexible.
- Nuevos algoritmos: se pueden agregar dentro de las clases ya escritas o como nuevas clases.

Todo esto puede agregarse sin realizar ningún cambio en el resto del código, lo cual muestra claramente el poder de extensión de la biblioteca.

### 10.6. Facilidad de uso

Se han elegido nombres claros para los nombres de los tipos (`GRAPH`, `VERTEX`, `EDGE`, etc) y para los algoritmos (`is_bipartite`, `shortest_path`, etc).

Se han provisto interfaces sencillas y cortas. Además se provee de interfaces aún más simples para usar grafos cuyos nodos sean números enteros.

Se logró una gran genericidad y extensibilidad.

Todo esto contribuye a que la biblioteca sea fácilmente reusable.

## 11. Agradecimientos

Agradezco a los profesores Héctor Gramaglia, Daniel Friedlender y Daniel Penazzi por enseñarme los algoritmos utilizados, a Daniel Moisset y Antonio Lenton por brindarme algunas ideas útiles, a Carlos de la Torre y a Matthias Gallé por impulsar este proyecto y a Agustín Bartó por facilitarme unos compiladores.

## Referencias

- [1] Bertrand, Meyer : “Object Oriented Software Construction”, segunda edición.
- [2] Foulds, L. R.: “Graph Theory Applications”.
- [3] [www.boost.org/libs/graphs/doc](http://www.boost.org/libs/graphs/doc)
- [4] [http://www.osl.iu.edu/research/comparing/eiffel\\_readme.html](http://www.osl.iu.edu/research/comparing/eiffel_readme.html)
- [5] Roberts, Fred S.: “Applied Combinatorics”.
- [6] Biggs, Norman L.: “Discrete Mathematics”.
- [7] <http://www.research.att.com/sw/tools/graphviz/>