26 | TDA Gráfica

Definiciones

Definición 26.1: Gráfica

Una *gráfica* es una tupla $G = (V, E)$ con:

V un conjunto de vértices.

E un conjunto de aristas que conectan pares de vértices en V.

- Una *gráfica pesada* también tiene una función $\omega : E \to \mathcal{R}$.
- En una gráfica *dirigida* $e = v_i \to v_j$, v_i es **vecino** de v_j.
- En una gráfica *no dirigida* $e = v_i - v_j$, v_i es **vecino** de v_j y v_j es **vecino** de v_i.

La Figura 26.1 muestra una gráfica dirigida con pesos. En esta ocasión, la misma definición del tipo de dato abstracto requiere la presencia explícita de las dos componentes: los vértices y las aristas, de lo que se desprende que sus operaciones hacen referencia a estos elementos y, por consiguiente, en esta ocasión, nuestro encapsulamiento de la estructura de datos no los ocultará.

Definición 26.2: Camino

Un *camino* $C = \{v_1 - ... - v_k\}$ en una gráfica $G = (V, E)$ es una secuencia de vértices $v_i \in V$, que se encuentran conectados entre sí mediante una sola arista $e \in E$ en la gráfica G.

Además de la creación del grafo consideraremos tres familias de operaciones que pode-

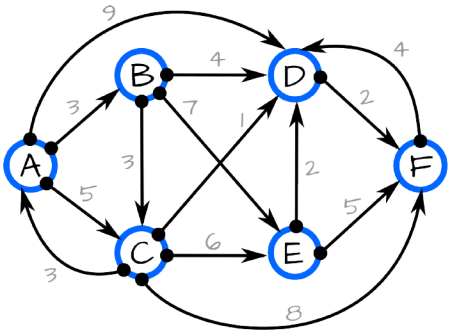


Figura 26.1 Gráfica dirigida pesada.

mos realizar sobre una gráfica: recorridos, búsqueda de caminos y generación de árboles de peso mínimo.

Recorridos

Dado que las gráficas, a diferencia de los árboles, permiten la existencia de caminos que regresen a un nodo anterior (*ciclos*), para recorrerlas, en general, es necesario contar con una estructura auxiliar que lleve registro de qué vértices ya fueron visitados. La estructura por excelencia es un tabla de dispersión, donde la llave es el vértice y el valor asociado, un Booleano, dado que esta estructura deberá permitir consultar en tiempo constante si un vértice ya fue visitado o no.

Las gráficas se pueden recorrer inicialmente en:

Amplitud También conocido como recorrido a distancia constante, los vértices son visitados a partir de un nodo inicial, luego se visitan los vértices a una arista de distancia de este nodo, luego a dos aristas, etc. Se utiliza una cola como estructura auxiliar para formar a los vértices que serán visitados.

Peso constante Semejante al recorrido en amplitud, pero la distancia al nodo inicial se mide sumando los pesos de las aristas desde dicho nodo inicial hasta el nodo en consideración, por lo que la estructura auxiliar correspondiente es una cola de prioridades, en la que la prioridad está dada por la distancia al nodo inicial.

Profundidad Partiendo de un nodo inicial se viaja de un vecino a su vecino hasta que no sea posible avanzar más por falta de vecinos no visitados, entonces se reinicia el procedimiento desde el último nodo visitado, que aún tenga vecinos sin visitar. Se implementa utilizando una pila como estructura auxiliar.

El recorrido parte de algún nodo indicado, que es agregado a la estructura auxiliar característica del tipo de recorrido. Mientras la estructura auxiliar no quede vacía se repite, a continuación, un ciclo que contiene los siguientes pasos:

- Se extrae un elemento de la estructura auxiliar y se añade al recorrido, marcándolo como visitado.
- Se toman todos sus vecinos, que no hayan sido visitados, y se agregan a la estructura auxiliar. Dado que un vértice puede ser vecino de varios vértices, aparecerán varias veces como candidatos a ser visitados, por eso es importante la tabla de vértices visitados.
  - ⋆ En el recorrido en amplitud, los vértices se marcan como visitados en cuanto se agregan a la cola.
  - ⋆ En peso constante, si un vértice ya estaba formado y se vuelve a considerar, se revisa si su distancia al vértice inicial disminuye al llegar desde este nuevo vértice; de ser el caso se actualiza su prioridad.
  - ⋆ En el recorrido en profunidad se elige entre ganar tiempo o espacio: se gastará más tiempo si, al añadir a la pila un vértice por segunda vez, se borra de la posición anterior que ocupaba, pero se ahorra un poco de espacio; se gasta más memoria si simplemente se deja ahí y al salir por segunda vez de la pila se le ignora por ya haber sido visitado.

Camino de peso mínimo

Definición 26.3: Camino de peso mínimo

El peso $w(C)$ de un camino $C = \{v_1 - ... - v_k\}$ es la suma de los pesos de las aristas que lo consituyen. Un *camino de peso mínimo* del vértice u al vértice v es cualquier camino cuyo peso es menor o igual que el peso para cualquier otro camino de u a v, si alguno existe.

Para calcular el camino de peso mínimo desde un vértice hasta cualquier otro alcanzable en la gráfica se puede utilizar el algoritmo de Dijkstra [Algoritmo 5]. Para implementarlo se pueden utilizar tres tablas de dispersión con el vértice como llave o una sola que contenga objetos con los tres valores indicados a continuación, más una cola de prioridades.

- Mantener tablas de dispersión con los vértices como llaves:
  - $k(v)$ Terminado. Ya se conoce la distancia mínima a v.
  - $d(v)$ Distancia. Distancia del vértice inicial a v.
  - $P(v)$ Predecesor. Vértice que antecede a v en el camino de peso mínimo.
- Q cola de prioridades **mínima** según $d(v)$, con $k(v) = \texttt{False}$

La estructura $P(v)$ contendrá la descripción de un árbol, pues desde cada nodo es posible reconstruir, en reversa, un camino de peso mínimo de la raíz hasta v.

Algoritmo 5 Dijkstra

```
1:  k(v) ← Falso, d(v) ← ∞, P(v) ← ∅
2:  Q ← V
3:  for |V| veces do
4:      Elegir de Q al vértice v con menor d(v).
5:      k(v) ← Verdadero
6:      for all vecino μ de v con k(μ) = Falso do
7:          if d(μ) > d(v) + ω(v, μ) then
8:              d(μ) ← d(v) + ω(v, μ)                ▷ Actualizar Q
9:              P(μ) ← v
10:         end if
11:     end for
12: end for
```

Árbol generador de peso mínimo

Definición 26.4: Árbol de expansión mínima

Un *árbol generador de peso mínimo* o *árbol de expansión mínima* es un grafo acíclico, cuyas aristas $T \subseteq E$ conectan a todos los vértices V y cuyo peso total

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

es mínimo.

Es posible obtener el árbol generador de peso mínimo de un grafo utilizando el algoritmo de Prim, que resulta ser el algoritmo de Dijkstra con una pequeña variante: la prioridad en la estructura auxiliar Q está dada por el peso de la arista que conecta al vértice v con su predecesor en $P(v)$, en lugar de ser el peso de todo el camino desde el vértice inicial. La estructura $P(v)$ contiene la descripción del árbol de peso mínimo. Dicho esto, una vez programado Dijkstra, prácticamente ya se tiene Prim.

27 | Gráfica con listas de adyacencia

Meta

Que el alumno implemente una *gráfica* haciendo uso de *listas de adyacencia* y los principales algoritmos para trabajar con ellas.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Aplicar las características de las estructuras de datos vistas anteriormente para implementar una estructura más compleja.
- Hacer uso de estructuras auxiliares para la implementación de algoritmos sobre gráficas.

Código Auxiliar 27.1: Gráfica con listas de adyacencia

https://github.com/computacion-ciencias/ed-grafica-adyacencia

Antecedentes

Estructura

La implementación de una gráfica utilizando listas de adyacencia elabora en la metodología utilizada para representar a los árboles binarios haciendo uso de vértices y referencias, pero con el ingrediente adicional de aristas que pueden tener un peso asociado. Para esta práctica trabajarás con gráficas dirigidas pesadas. Cada vértice almacenará en su interior un dato, que también le servirá como etiqueta para distinguirlo de los otros vértices. La Figura 27.1 muestra las clases que usarás, con sus atributos y métodos. En

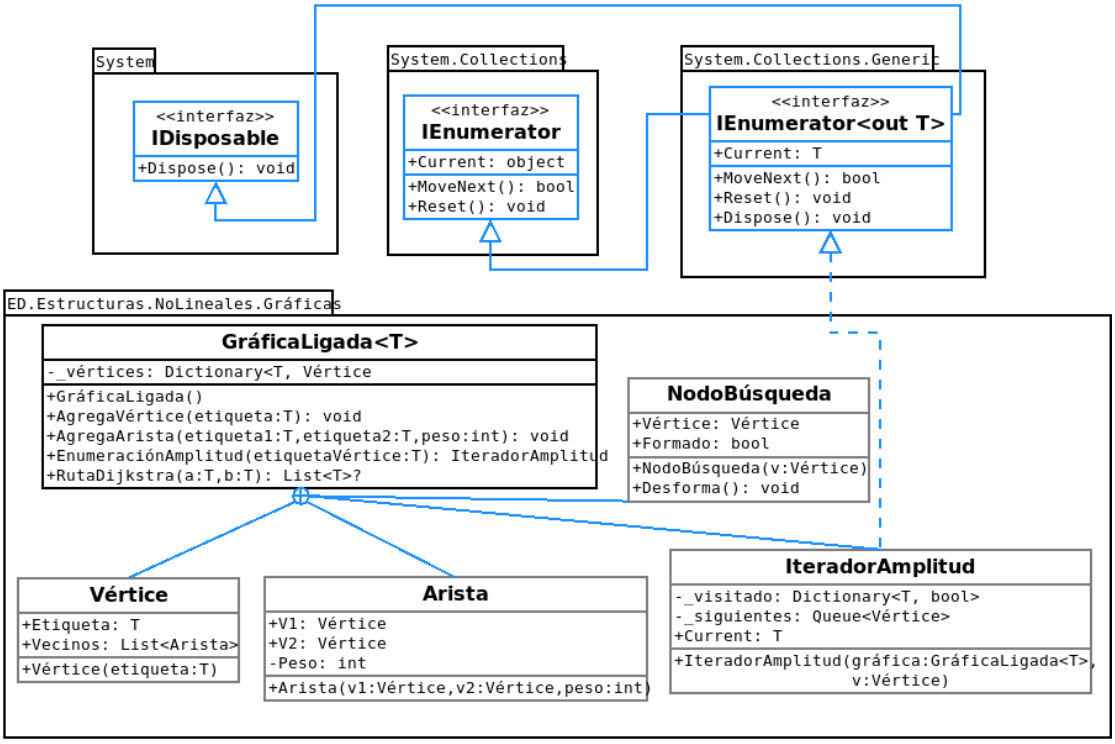


Figura 27.1 La gráfica ligada contiene cuatro clases internas.

esta ocasión harás uso de cuatro clases internas, tres de ellas se utilizan para almacenar a la gráfica y operar con ella: `Vértice`, `Arista` e `IteradorAmplitud`; la cuarta, `NodoBúsqueda`, te servirá como auxiliar para implementar el algoritmo para encontrar la ruta más corta entre dos nodos. El ser internas permitirá a estas clases acceder al tipo genérico `T`.

La clase `Arista` contiene referencias al vértice donde incia `V1` y al vértice hacia el que apunta `V2`; se implementarán como atributos sólo con permiso de lectura, pues su valor sólo se asignará en el constructor. Mientras que cada vértice guardará en su lista `Vecinos` a aquellas aristas que salgan de él. Sería posible tener también una lista de aristas que apunten al vértice, pero no la necesitarás en esta práctica.

La clase `GráficaLigada<T>` tendrá como atributo una tabla de dispersión que indexe a los vértices según su etiqueta. Entonces, dada la etiqueta de un vértice podrás acceder al vértice y, a partir de él, a las aristas que emergen de él.

Observa que en esta ocasión `GráficaLigada<T>` no implementa `IEnumerable<T>`. Esto es porque cualquier recorrido requiere que se indique el vértice inicial, elegir uno al azar sería demasiado arbitrario y el método `GetEnumerator()` no recibe argumentos. Pero sí programarás un método `EnumeraciónAmplitud()`, que reciba la etiqueta del vértice inicial como parámetro, y devuelva un iterador que recorra la gráfica en amplitud a partir de él.

Cuando un usuario haga uso de tu clase, su código se verá como el siguiente:

```
public class UsoGráfica
{
  public static void Main()
  {
    GráficaLigada<char> g = new();
    g.AgregaVértice('A');
    g.AgregaVértice('B');
    g.AgregaArista('A', 'B', 10);
    IEnumerator<char> it = g.EnumeradorAmplitud('A');
    while(it.MoveNext())
    {
      WriteLine("Vértice␣" + it.Current + "␣visitado.");
    }
  }
}
```

Observa que, para que una clase interna pueda acceder a los atributos de la clase que la contiene, necesita tener una referencia al objeto que los contiene. Por ello IteradorAmplitud solicita esta referencia en su constructor.

Iteradores

Programa sólo el iterador en amplitud. Utiliza una cola como atributo del iterador para guardar los vértices a visitar. En cada llamada a `MoveNext()` marca como `Current` al nodo extraído de la cola. El recorrido termina cuando la cola esté vacía.

Desarrollo

1. Crea la clase `GráficaLigada<T>` en el espacio de nombres

   `ED.Estructuras.NoLineales.Gráficas`

2. Programa las clases internas `Vértice` y `Arista` según el diagrama UML. Cuida que la etiqueta del vértice no puede ser `null`, ni los vértices conectados por la arista.

3. Agrega la tabla de dispersión a `GráficaLigada<T>` y un constructor que inicialice una gráfica vacía.

4. Agrega los métodos:

   - `public void AgregaVértice(T etiqueta)`. Crea un vértice con la etiqueta indicada y lo almacena en la tabla de dispersión.

- `public void AgregaArista(T etiqueta1, T etiqueta2, int peso)`. Crea la arista conectando a los vértices representados por las etiquetas y la almacena en la lista del vértice inicial.
- Añade el iterador en amplitud.
- Programa la clase `NodoBúsqueda`, lo necesitarás para modificar las prioridades en la cola de prioridades que requiere el algoritmo de Dijkstra.
- Programa el método `List<T> RutaDijkstra(T a, T b)` que reciba las etiquetas de dos vértices en la gráfica y devuelva la lista de etiquetas de vértices que correspondan a un camino de peso mínimo entre los vértices con etiquetas `a` y `b`.

  Observa que las estructuras auxiliares necesarias para implementar este algoritmo serán sólo variables locales a este método, no necesitas añadir atributos a la clase.

  Cuando necesites meter un vértice a la cola de prioridades, envuélvelo primero en un objeto nuevo tipo `NodoBúsqueda`. La propiedad `Formado` debe ser `true`, pues estará formado en la cola. También debes meter el nodo búsqueda a un diccionario usando como llave la etiqueta del vértice. De este modo tendrás dos referencias al nodo: desde la cola de prioridades y desde el diccionario. Cuando el algoritmo te requiera modificar la prioridad de algún vértice harás las operaciones siguientes:

  a) Accede al nodo búsqueda del vértice utilizando el diccionario.
  b) Usa el método `Desforma` para marcar que ya no está en la cola, aunque en realidad no podrás este nodo de la cola, tendrá que quedarse donde esté.
  c) Remueve ese nodo del diccionario.
  d) Crea un nuevo nodo búsqueda con la nueva prioridad del vértice y repite el procedimiento para insertar un nodo.

  Ahora, cada vez que solicites el siguiente nodo de la cola de prioridades, ignora a todos los que estén marcados como desformados.
- Prueba y documenta tu código.

Preguntas

1. Dado el diseño de esta clase es posible tener una gráfica donde un vértice esté desconectado de todos los demás. Si quisieras impedirlo ¿Cómo lo harías?
2. ¿Qué necesitarías para implementar el algoritmo de Prim en otra función?

28 | Gráfica con matriz de adyacencia

Meta

Que el alumno utilice la representación de una *gráfica* con *matriz de adyacencia* para implementar el algoritmo de *Floyd-Warshall*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Utilizar la representación con matriz de adyacencia para una gráfica.
- Implementar un algoritmo eficiente para el cálculo de las rutas de menor peso de cualquier vértice a cualquier otro vértice en una gráfica dirigida.

Código Auxiliar 28.1: Gráfica con matriz de adyacencia

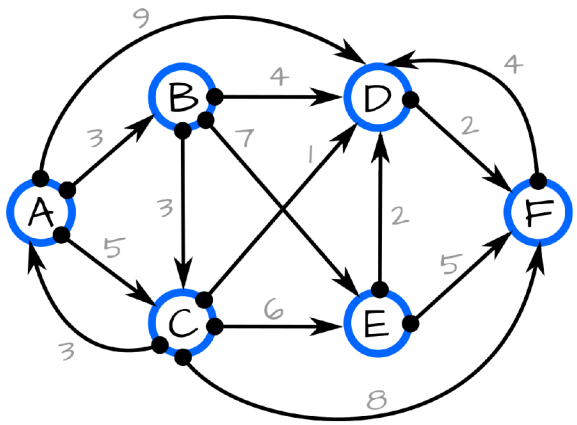
https://github.com/computacion-ciencias/ed-grafica-matriz

Antecedentes

Estructura

La implementación más sencilla y directa de una gráfica dirigida es sobre un arreglo bidimensional o matriz. Las etiquetas de las columnas y renglones corresponden a los vértices V, de modo que el contenido de la casilla (v_i, v_j) sea el peso del arista $e = v_i \to v_j$, si no hay conexión se asocia el valor ∞ y sobre la diagonal se colocan ceros. [Figura 28.1]

Para gráficas no pesadas basta con utilizar un sistema binario para indicar *presencia* o *ausencia* de conexión. Obsérvese que para una gráfica no dirigida basta con almacenar



	A	B	C	D	E	F
A	0	3	5	9	∞	∞
B	∞	0	3	4	7	∞
C	3	∞	0	1	6	8
D	∞	∞	∞	0	∞	2
E	∞	∞	∞	2	0	5
F	∞	∞	∞	4	∞	0

Figura 28.1 **Izquierda:** gráfica. **Derecha:** Matriz de adyacencia correspondiente.

las conexiones en un arreglo triangular, pues los pesos se obtienen por reflexión sobre la diagonal.

Si bien sería sencillo implementar el TDA Gráfica utilizando esta representación subyacente, también es posible utilizarla como un atributo dentro de una clase para realizar procedimientos más complejos, como implementar el algoritmo de Floyd-Warshall para encontrar las rutas de peso mínimo entre todos los pares de vértices.

Floyd-Warshall

El algoritmo de Floyd Warshall utiliza dos matrices en todo tiempo para llevar registro de los caminos de peso mínimo computados hasta el momento. La primer matriz D_i contiene el peso del mejor camino conocido hasta el paso i del vértice en el renglón hacia el vértice indicado por la columna; mientras que la segunda matriz Π_i indica cuál es el vértice predecesor al vértice en la columna en dicha ruta.

Las matrices iniciales son la matriz de pesos D_0 y una matriz de padres que indica los vértices a la izquierda en la definición de cada arista $v_i \to v_j$, según se muestra en la Figura 28.2, para la gráfica en la Figura 28.1. Esta inicialización y los pasos siguientes se muestran en el Algoritmo 6.

	A	B	C	D	E	F
A	0	A	A	A	NIL	NIL
B	NIL	0	B	B	B	NIL
C	C	NIL	0	C	C	C
D	NIL	NIL	NIL	0	NIL	D
E	NIL	NIL	NIL	E	0	E
F	NIL	NIL	NIL	F	NIL	0

Figura 28.2 Matriz con los predecesores de cada vértice al inicio de la ejecución de Floyd-Warshall.

Algoritmo 6 Floyd-Warshall

1: $D^k \leftarrow matriz(n \times n)$, $\Pi^k \leftarrow matriz(n \times n)$ ▷ Costos, Padres
2: $D^0 \leftarrow W$ ▷ Matriz de pesos (adyacencia)
3: **for** $i \in [1, n]$ **do**
4: **for** $j \in [1, n]$ **do**
5: $$\Pi^0_{i,j} \begin{cases} NIL & \text{si } i = j \text{ ó } \omega(i,j) = \infty \\ i & \text{si } i \neq j \text{ y } \omega(i,j) < \infty \end{cases}$$
6: **end for**
7: **end for**
8: **for** $k \in [1, n]$ **do** ▷ Agrega vértices
9: **for** $i \in [1, n]$ **do**
10: **for** $j \in [1, n]$ **do**
11: $d_{ikj} \leftarrow d^{k-1}_{i,k} + d^{k-1}_{k,j}$
12: $D^k_{i,j} \leftarrow mín(d^{k-1}_{i,j}, d_{ikj})$
13: $$\Pi^k_{i,j} \leftarrow \begin{cases} \pi^{k-1}_{ij} & \text{si } d^{k-1}_{ij} \leqslant d_{ikj} \text{ ▷ No pasa por } v_k \\ \pi^{k-1}_{kj} & \text{si } d^{k-1}_{ij} > d_{ikj} \text{ ▷ Pasa por } v_k \end{cases}$$
14: **end for**
15: **end for**
16: **end for**

Desarrollo

1. Para esta práctica crea una clase llamada `FloydWarshall` dentro del paquete

   `ed.estructuras.nolineales.gráficas`.