Algoritmos y estructuras de datos

Representación de gráficas y digráficas

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Posgrado en Optimización

31 de mayo de 2021

Citas

Proverbio sumerio

Que haya tierra sin usar adyacente a cada casa.

Ward Cunningham

Mientras mueves comentarios de un lado a otro y tienes comentarios similares adyacentes el uno al otro, con frecuencia descubres que la mitad de las palabras se pueden eliminar. Pues un enunciado lo dice todo si está en el lugar correcto.

Mr. Scott (Star Trek)

¿Cómo piensa que llegué aquí? Tuve un pequeño debate con mi profesor de física relativista y su relación al viaje subespacial. Él parecía pensar que el rango de transportación de algo como una toronja estaba limitado a unas cien millas. Yo le dije que, con el mismo sistema, no solo podía enviar una toronja de un planeta a su planeta adyacente, lo cual es muy fácil, sino que también podía hacerlo con una forma de vida.

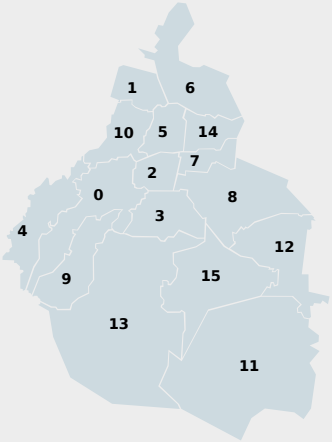
Una gráfica $G = (V, E)$ consta de n vértices y m aristas. Cada arista une a dos vértices distintos (vecinos) y cada pareja de vértices está unida por a lo más una arista.

Una digráfica $D = (V, A)$ consta de n vértices y m arcos. Cada arco va de un origen a un destino y cada pareja de vértices está unida por a lo más dos arcos opuestos.

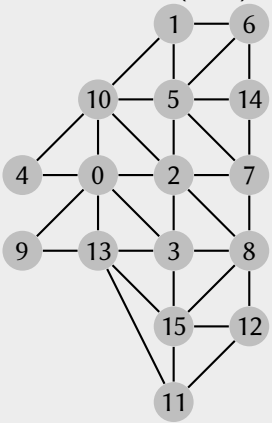
Cada vértice y arista de G (o arco de D) puede venir acompañado de cierta información. Un vértice puede tener nombre, ubicación, etc., mientras que una arista o arco puede tener nombre, costo, etc. Para simplificar, los vértices estarán numerados del 0 al $n - 1$ y el resto de la información se almacenará por separado. Por ejemplo, los nombres de los vértices se podrían almacenar usando un arreglo de apuntadores a cadenas (`char *s[n]`).

Mapa con nombres

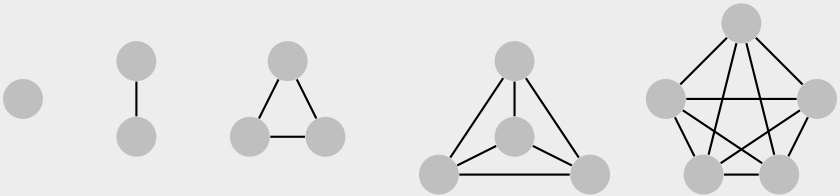
Mapa con números



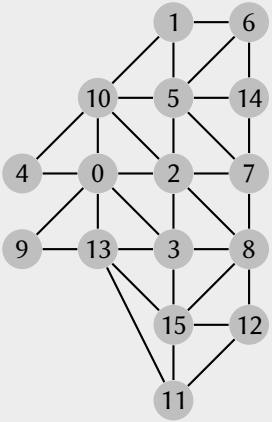
Gráfica $G = (V, E)$



Considere una gráfica $G = (V, E)$ que tiene $n = |V|$ vértices y $m = |E|$ aristas.



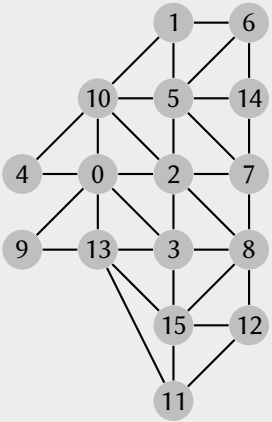
1. La gráfica K_n que tiene todas las aristas posibles $m = \frac{1}{2}n(n-1)$ se llama completa.
2. Una gráfica G que tiene relativamente muchas aristas ($m \propto n^2$) se llama densa.
3. Una gráfica G que tiene relativamente pocas aristas ($m \propto n$) se llama dispersa.
4. Ejemplos: gráficas planas ($m \leq 3n$), cuadrículas ($m \leq 2n$), árboles ($m \leq n$).



La forma más sencilla de representar una gráfica es a través de su matriz de adyacencia A, la cual es una matriz cuadrada simétrica de $n \times n$ en la cual

$$A_{uv} = \begin{cases} 1 & \text{si } u \text{ y } v \text{ están unidos por una arista} \\ 0 & \text{en caso contrario.} \end{cases}$$

La matriz de adyacencia ocupa espacio proporcional a n^2. Esto la hace ideal para representar gráficas completas o densas. Por otro lado, la hace completamente inadecuada para representar gráficas dispersas.



A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	1	1	0	0	0	0	1	1	0	0	1	0	0
1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0
2	1	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0
3	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1
4	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
5	0	1	1	0	0	0	1	1	0	0	1	0	0	0	1	0
6	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0
7	0	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0
8	0	0	1	1	0	0	0	1	0	0	0	0	1	0	0	1
9	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
10	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
12	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1
13	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	1
14	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
15	0	0	0	1	0	0	0	0	1	0	0	1	1	1	0	0

Definiremos un tipo estructurado `grafica` para representar la matriz de adyacencia de una gráfica. Este tipo consiste de la cantidad `n` de vértices y de una matriz `a` de $n \times n$.

```
typedef struct {
  int n;     // cantidad de vertices
  int **a; // matriz de adyacencia
} grafica;
```

En realidad `a` será un apuntador a un vector de `n` apuntadores a vectores de tamaño `n` (cada uno el mapa de bits de los vecinos de un vértice) pero posteriormente lo usaremos simplemente como una matriz. De ser necesario, esta estructura se podría aumentar con un vector para los nombres de los vértices, otra matriz para los costos, etc.

Creación de una matriz de adyacencia vacía

Abajo usamos `g.a` como un arreglo de renglones, donde el renglón `g.a[u]` corresponde con el vértice `u`, mientras que `g.a[u][v]` corresponde con la adyacencia de `u` y `v` (que fué inicializada en `0` por la llamada a `calloc`).

```
grafica creaGrafica(int n) {
  grafica g;
  g.n = n;
  g.a = (int **) malloc(g.n*sizeof(int *));
  for (int u = 0; u < g.n; u++)
    g.a[u] = (int *) calloc(g.n, sizeof(int));
  return g;
}
```

Creación de una matriz de adyacencia desde la entrada

En la entrada se espera primero `n` y luego los `n` renglones de la matriz de adyacencia.

```
grafica leeGrafica(void) {
  int a, n;
  scanf("%d", &n);
  grafica g = creaGrafica(n);
  for (int u = 0; u < g.n; u++)
    for (int v = 0; v < g.n; v++) {
      scanf("%d", &a);
      g.a[u][v] = a;
    }
  return g;
}
```

Primero liberamos cada uno de los renglones de la matriz `a` y luego liberamos el vector de apuntadores a los renglones.

```
void destruyeGrafica(grafica *g) {
  for (int u = 0; u < g->n; u++)
    free(g->a[u]);
  free(g->a);
  g->a = NULL;
  g->n = 0;
}
```

En ambos casos revisamos si los vértices están fuera de rango (arista ilegal).

```
void agregaArista(grafica g, int u, int v) {
  if (0 <= u && u < g.n && 0 <= v && v < g.n)
    g.a[u][v] = g.a[v][u] = 1;
}

void eliminaArista(grafica g, int u, int v) {
  if (0 <= u && u < g.n && 0 <= v && v < g.n)
    g.a[u][v] = g.a[v][u] = 0;
}
```

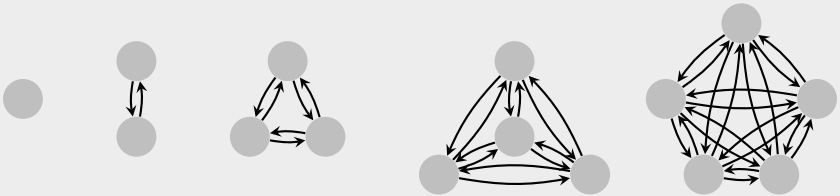
Determinar si u y v son vecinos toma un paso, pues solo se revisa g.a[u][v].

```
int sonVecinos(grafica g, int u, int v) {
  return g.a[u][v];
}
```

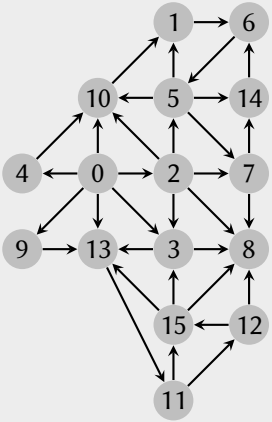
Determinar el grado de u toma *n* pasos, pues hay que revisar todo el renglón g.a[u].

```
int grado(grafica g, int u) {
  int v, d = 0;
  for (v = 0; v < g.n; v++)
    d += g.a[u][v];
  return d;
}
```

Considere una digráfica $D = (V, A)$ que tiene $n = |V|$ vértices y $m = |A|$ arcos.



1. La digráfica $\vec{K}_n$ que tiene todos los arcos posibles $m = n(n-1)$ se llama completa.
2. Una digráfica D que tiene relativamente muchos arcos ($m \propto n^2$) se llama densa.
3. Una digráfica D que tiene relativamente pocos arcos ($m \propto n$) se llama dispersa.
4. Ejemplos: digráficas planas ($m \leq 6n$), cuadrículas ($m \leq 4n$), árboles ($m \leq 2n$).



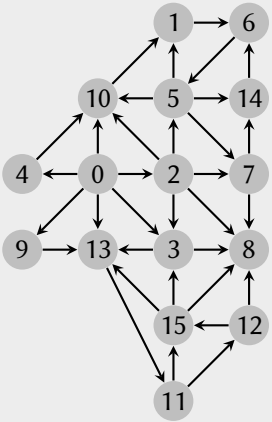
La forma más sencilla de representar una digráfica es a través de su matriz de adyacencia A, la cual es una matriz cuadrada de $n \times n$ (posiblemente asimétrica) en la cual

$$A_{uv} = \begin{cases} 1 & \text{si hay un arco de } u \text{ a } v \\ 0 & \text{en caso contrario.} \end{cases}$$

La matriz de adyacencia ocupa espacio proporcional a n^2. Esto la hace ideal para representar digráficas completas o densas. Por otro lado, la hace completamente inadecuada para representar digráficas dispersas.

Digráficas

Representación con matriz de adyacencia



A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	1	1	0	0	0	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	1	0	1	1	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
5	0	1	0	0	0	0	0	1	0	0	1	0	0	0	1	0
6	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
10	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
14	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
15	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0

Usaremos el mismo tipo estructurado `grafica` para representar la matriz de adyacencia de una digráfica.

```
typedef struct {
  int n;     // cantidad de vertices
  int **a; // matriz de adyacencia
} grafica;
```

Las operaciones de creación de una digráfica vacía o desde la entrada, así como la destrucción de una digráfica, también son las mismas que antes.

En ambos casos revisamos si los vértices están fuera de rango (arco ilegal).

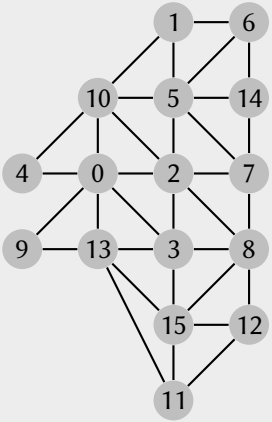
```
void agregaArco(grafica g, int u, int v) {
  if (0 <= u && u < g.n && 0 <= v && v < g.n)
    g.a[u][v] = 1;
}

void eliminaArco(grafica g, int u, int v) {
  if (0 <= u && u < g.n && 0 <= v && v < g.n)
    g.a[u][v] = 0;
}
```

Determinar el grado exterior toma n pasos, pues hay que revisar todo el renglón `g.a[u]`.

```
int gradoExterior(grafica g, int u) {
  int v, d = 0;
  for (v = 0; v < g.n; v++)
    d += g.a[u][v]; // arcos que salen de u
  return d;
}
```

Determinar el grado interior toma n pasos, pues hay que revisar toda la columna de `u`.

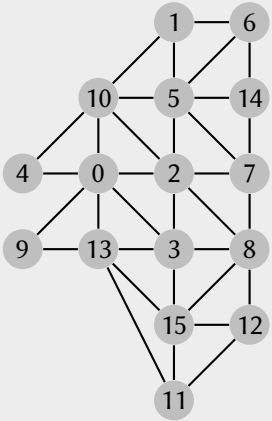
```
int gradoInterior(grafica g, int u) {
  int v, d = 0;
  for (v = 0; v < g.n; v++)
    d += g.a[v][u]; // arcos que entran a u
  return d;
}
```



Otra forma de representar una gráfica es a través de sus listas de adyacencia A, las cuales son representaciones de los conjuntos A_u de los vecinos de cada vértice u. Observe que una arista que une a u y v queda representada dos veces: como $v \in A_u$ y como $u \in A_v$.

Las listas de adyacencia ocupan espacio proporcional a $n + m$. Esto las hace ideales para representar gráficas dispersas (y ligeramente inferiores para gráficas densas).

Representación con listas de adyacencia



$$
\begin{aligned}
A_0 &= \{2,3,4,9,10,13\} & A_8 &= \{2,3,7,12,15\}\\
A_1 &= \{5,6,10\} & A_9 &= \{0,13\}\\
A_2 &= \{0,3,5,7,8,10\} & A_{10} &= \{0,1,2,4,5\}\\
A_3 &= \{0,2,8,13,15\} & A_{11} &= \{12,13,15\}\\
A_4 &= \{0,10\} & A_{12} &= \{8,11,15\}\\
A_5 &= \{1,2,6,7,10,14\} & A_{13} &= \{0,3,9,11,15\}\\
A_6 &= \{1,5,14\} & A_{14} &= \{5,6,7\}\\
A_7 &= \{2,5,8,14\} & A_{15} &= \{3,8,11,12,13\}
\end{aligned}
$$

Como cada lista de adyacencia es un conjunto, se puede representar de cualquiera de las formas presentadas antes. Las formas más comunes son:

Listas enlazadas A sería un arreglo de n listas enlazadas. Al inicio estas listas estarían vacías.

Vectores dinámicos A sería un arreglo de n vectores dinámicos. Al inicio cada vector estaría vacío (aunque posiblemente tendría tamaño positivo).

Vamos a suponer que nunca se intenta agregar la misma arista dos veces y que nunca se intenta agregar una arista ilegal (vértices fuera de rango o iguales). Esto significa que las listas enlazadas o los vectores dinámicos pueden permanecer o estar desordenados. Esto hará que muchas operaciones sean muy rápidas y no nos causará mayores problemas.

Definiremos un tipo estructurado `graficaLA` para representar las listas de adyacencia de una gráfica. Este tipo consiste de la cantidad `n` de vértices y de un vector `a` de `n` listas.

```
typedef struct {
  int    n; // cantidad de vertices
  lista *a; // listas de adyacencia
} graficaLA;
```

De ser necesario, esta estructura se podría aumentar con un vector para los nombres de los vértices, los nodos de las listas podrían almacenar los costos de las aristas, etc.

Abajo usamos `g.a` como un arreglo de listas, donde `g.a[u]` es el apuntador al inicio de la lista del vértice `u`, que al principio está vacía.

```
graficaLA creaGraficaLA(int n) {
  graficaLA g;
  g.n = n;
  g.a = (lista *) malloc(g.n*sizeof(lista));
  for (int u = 0; u < g.n; u++)
    g.a[u] = NULL;
  return g;
}
```

Creación de listas de adyacencia desde la entrada

En la entrada se espera primero `n` y `m` y luego `m` renglones con parejas de vértices `u` y `v` unidos por una arista. Las funciones que terminan en `Nodo` o `Lista` son de listas enlazadas.

```
graficaLA leeGraficaLA(void) {
  int n, m, u, v;
  scanf("%d %d", &n, &m);
  graficaLA g = creaGraficaLA(n);
  for (int j = 0; j < m; j++) {
    scanf("%d %d", &u, &v);
    insertaNodo(&g.a[u], v);
    insertaNodo(&g.a[v], u);
  }
  return g;
}
```

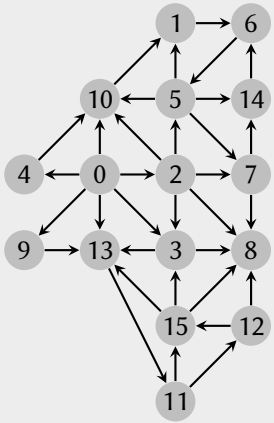
Primero liberamos cada una de las listas en a y luego liberamos el vector de listas.

```
void destruyeGraficaLA(graficaLA *g) {
  for (int u = 0; u < g->n; u++)
    destruyeLista(&g.a[u]);
  free(g->a);
  g->a = NULL;
  g->n = 0;
}
```

Determinar vecindad y determinar el grado d toman d pasos recorriendo la lista.

```
int sonVecinosLA(graficaLA g, int u, int v) {
  return buscaLista(g.a[u], v) != NULL;
}

int gradoLA(graficaLA g, int u) {
  return longitudLista(g.a[u]);
}
```

El grado se podría encontrar en 1 paso si se almacena un contador por vértice.

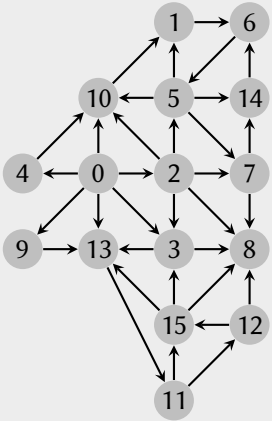


Otra forma de representar una digráfica es a través de sus listas de adyacencia A, las cuales son representaciones de los conjuntos A_u de los destinos de cada arco con origen u. Observe que un arco que va de u a v queda representado solo una vez como $v \in A_u$.

Las listas de adyacencia ocupan espacio proporcional a $n + m$. Esto las hace ideales para representar digráficas dispersas (y ligeramente inferiores para digráficas densas).

Digráficas

Representación con listas de adyacencia



$$
\begin{aligned}
A_0 &= \{2, 3, 4, 9, 10, 13\} \\
A_1 &= \{6\} \\
A_2 &= \{3, 5, 7, 8, 10\} \\
A_3 &= \{8, 13\} \\
A_4 &= \{10\} \\
A_5 &= \{1, 7, 10, 14\} \\
A_6 &= \{5\} \\
A_7 &= \{8, 14\}
\end{aligned}
\qquad
\begin{aligned}
A_8 &= \emptyset \\
A_9 &= \{13\} \\
A_{10} &= \{1\} \\
A_{11} &= \{12, 15\} \\
A_{12} &= \{8, 15\} \\
A_{13} &= \{11\} \\
A_{14} &= \{6\} \\
A_{15} &= \{3, 8, 13\}
\end{aligned}
$$

Usaremos el mismo tipo estructurado `graficaLA` para representar las listas de adyacencia de una digráfica.

```
typedef struct {
  int    n; // cantidad de vertices
  lista *a; // listas de adyacencia
} graficaLA;
```

Las operaciones de creación de una digráfica vacía, así como la destrucción de una digráfica, también son las mismas que antes. Saber si hay un arco de u a v se hace con `sonVecinosLA(g, u, v)`, mientras que el grado exterior de u se obtiene de `gradoLA(g, u)`. Calcular el grado interior de u requeriría recorrer todas las listas de adyacencia.

Distinto a `leeGraficaLA` en que el arco de u a v solo se inserta en la lista de u.

```
graficaLA leeGraficaLAD(void) {
  int n, m, u, v;
  scanf("%d %d", &n, &m);
  graficaLA g = creaGraficaLA(n);
  for (int j = 0; j < m; j++) {
    scanf("%d %d", &u, &v);
    insertaNodo(&g.a[u], v);
  }
  return g;
}
```