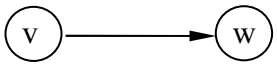
Grafos dirigidos

Definiciones básicas

Un grafo dirigido (o digrafo) G consiste de un conjunto de vértices y un conjunto de arcos E. A los vértices se les llama también nodos o puntos y a los arcos aristas dirigidas o líneas dirigidas. Un arco es un par ordenado de vértices (v, w) donde v es la cola y w la cabeza del arco. Un arco (v, w) se expresa también como $v \rightarrow w$ y se dibuja como



Se dice que un arco $v \rightarrow w$ va de v a w y que w es adyacente a v.

Los vértices de un digrafo pueden usarse para representar objetos, y los arcos relaciones entre los objetos.

Un camino en un digrafo es una secuencia de vértices v_1, v_2, …, v_n, tal que $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3$;…., $v_{n-1} \rightarrow v_n$ son arcos. El camino es de v_1 a v_n y pasa a través de los vértices v_2, v_3,…, v_{n-1} y termina en el vértice v_n. La longitud de un camino es el número de arcos del camino, en este caso, n−1.

Un camino en simple si todos los vértices del camino son distintos. Un ciclo simple es un camino simple de longitud uno como mínimo, que empieza y termina en el mismo vértice. Un grafo etiquetado es un digrafo en el que cada arco y/o vértice puede tener una etiqueta asociada. Una etiqueta puede ser un nombre, un costo o un valor de algún tipo de dato dado.

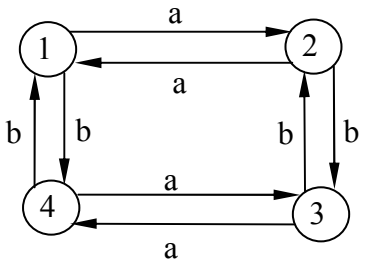
Representaciones de Grafos dirigidos

Pueden usarse varias estructuras de datos para representar un digrafo, dependiendo su selección de las operaciones que se aplicarán a los vértices y arcos del digrafo.

Una representación común para un digrafo G={V,E} es la matriz de adyacencia. Suponiendo V={1,2,…,n}, la matriz de adyacencia de G es una matriz A de nxn de valores binarios donde A[i][j] es 1 si y sólo si hay un arco del vértice i a j, y 0 si no lo hay.

Otra representación reñacionada es la matriz de adyacencia estiquetada donde A[i][j] es la etiqueta del arco que va de i a j. Si tal arco no existe puede usarse una etiqueta especial para ese caso.

Ejemplo: A continuación se muestra un digrafo y su correspondiente matriz de adyacencia.

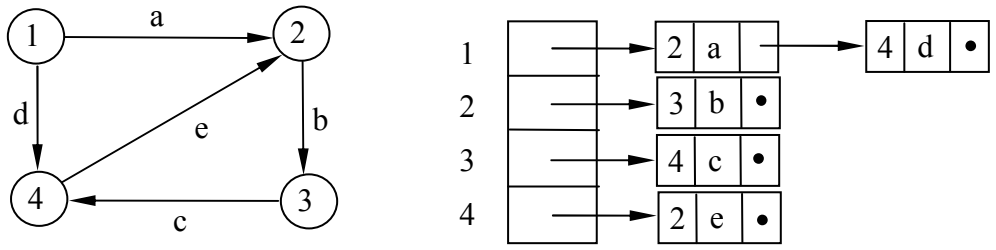


	1	2	3	4
1		a		b
2	a		b	
3		b		a
4	b		a	

La principal desventaja de usar la matriz de adyacencia es que requiere $\Omega(n^2)$ de memoria, áun cuando se tenga un digrafo con menos de n^2 arcos. Para leer o examinar la matriz se requiere un tiempo de $O(n^2)$.

Para evitar esta desventaja se puede usar otra representación para un digrafo G=(V,E) llamada lista de adyacencia. La lista de adyacencia de un vértice i es una lista, en algún orden, de todos los vértices adyacentes a i. Se puede representar G mediante un array HEAD donde HEAD[i] es un puntero a la lista de adyacencia del vértice i.

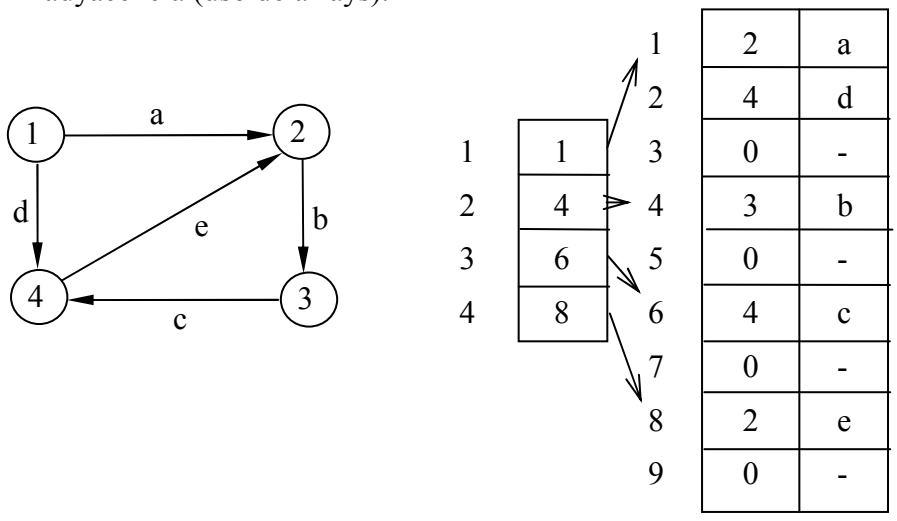
Ejemplo: A continuación se muestra un digrafo y su representación usando la lista de adyacencia. Se ha usado una lista enlazada simple.



La representación mediante lista de adyacencia requiere una memoria proprocional al número de vértices más el número de arcos.Se usa cuando el número de arcos es mucho menor a n^2. La desventaja es que podemos tomar un tiempo $O(n)$ para determinar si hay un arco del vértice i al j.

Si se espera que el grafo permanezca fijo (pocos o ningún cambio en la lista de adyacencia) puede usarse HEAD[i] como un cursor de un un array ADJ, donde ADJ[HEAD[i]] contiene los vértices adyacentes a i, hasta que se encuentre un final de lista (puede ser un 0).

Ejemplo: Para el siguientes digrafo se muestra otra representación de la lista de adyacencia (uso de arrays).



Tipo de dato abstracto para grafos dirigidos

Las principales estructuras de datos para grafos se han visto anteriormente. Las operaciones más comunes en grafos dirigidos incluyen la lectura de una etiqueta de un

vértice o arco, insertar o borrar vértices y arcos y navegar siguiendo arcos desde su cola a cabeza.

Para lograr esas operaciones, frecuentemente se usa el siguiente esquema de instrucción:

for cada vértice *w* adyacente a *v*

{ alguna acción sobre *w* }

Para implementar este paso, es necesario el concepto de tipo índice para el conjunto de vértices adyacentes a algún vértice *v*. Por ejemplo, si se usa la lista de adyacencia, entonces un índice es realmente una posición en la lista de adyacencia de *v*. Si se usa la matriz de adyacencia, un índice es un entero que representa un vértice adyacente.

Son necesarias las siguientes operaciones en grafos dirigidos:

primero(*v*) retorna el índice del primer vértice adyacente a *v*. Si no existe retorna el vértice nulo Λ.

siguiente(*v*,i) retorna el índice posterior al índice i para los vértices adyacentes a *v*. Si i es el último índice no existe retorna Λ.

vertice(*v*,i) retorna el vértice con índice i entre los vértices adyacentes a *v*.

Ejemplo: A continuación se muestra la implementación de las operaciones anteriores que usan la representación de la matriz de orden $n_x n$:

```
#include <stdio.h>
#define MAX_NODOS 180
#define NULO 0

typedef unsigned short int VERTICE;
VERTICE primero(VERTICE v);
VERTICE siguiente(VERTICE v,VERTICE i);
VERTICE vertice(VERTICE v,VERTICE i);
void lee_grafo_fichero();
void imprime_grafo();
void operacion(VERTICE v);
int n;
VERTICE grafo[MAX_NODOS][MAX_NODOS];

VERTICE primero(VERTICE v)
{
  VERTICE i;

  for(i=0;i<n;i++)
    if (grafo[v][i]==1)
      return(i);
  return (NULO);
}

VERTICE siguiente(VERTICE v,VERTICE i)
```

```
{
 VERTICE j;

 for(j=i+1;j<n;j++)
   if (grafo[v][j]==1)
     return(j);
 return (NULO);
}

VERTICE vertice(VERTICE v,VERTICE i)
{
 if (grafo[v][i]==1)
  return(i);
 else
  return(NULO);
}

void operacion(VERTICE v)
{
 VERTICE i,w;

 i=primero(v);
 while(i!=NULO)
 {
  w=vertice(v,i);
  printf("accion sobre %d\n",w);
  i=siguiente(v,i);
 }
}

void lee_grafo_fichero()
{
 FILE *fp;
 int i,j;

 fp=fopen("grafo.dat","r");
 fscanf(fp,"%d",&n);
 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   fscanf(fp,"%d",&grafo[i][j]);
}

void imprime_grafo()
{
 int i,j;

 printf("Numero de nodos: ");
   printf("%d \n",n);
 printf("Matriz de adyacencia:\n");
```

```
  for(i=0;i<n;i++)
  {
   for(j=0;j<n;j++)
    printf("%d ",grafo[i][j]);
   printf("\n");
  }
}
```

El problema de los caminos más cortos desde un vértice

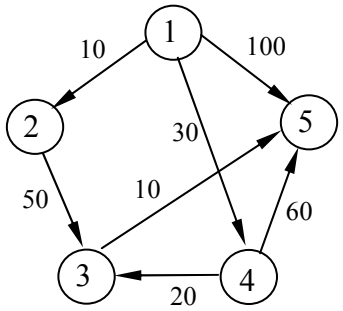
Si en un grafo dirigido G=(V,E) cada arista se especifica por un valor no negativo que representa un costo o distancia y se selecciona un vértice como fuente. El problema consiste en determinar la distancia más corta desde el vértice fuente a cada vértice distinto en V, donde la distancia de un camino es la suma de las distancias de las aristas del camino.

Una solución a éste problema es usar el algoritmo de Dijkstra. Este algoritmo trabaja manteniendo un conjunto de vértices S cuya menor distancia desde la fuente es conocida. Inicialmente S contiene sólo el vértice fuente. En cada iteración, se añade a S un vértice remanente *v* cuya distancia desde la fuente sea tan pequeña como sea posible. Asumiendo que todos los arcos tienen longitudes no-negativas, siempre es posible encontrar un camino más corto desde la fuente *v* que pasa sólo por vértices en S, al cual llamamos especial. Para registrar la longitud del camino especial a cada vértice se usa un array *d* que se actualiza en cada iteración del algoritmo. Cuando el conjunto S incluye todos los vértices del grafo todos los caminos son especiales por lo que el array *d* mantendrá la distancia más corta desde la fuente a cada vértice del grafo.

Para un digrafo G=(V,E) donde V={1,2,3,…,*n*} se asume que se selecciona el vértice *f* como fuente. La matriz de costos C es un array bidimensional donde C[i][j] es el costo de ir del vértice *i* al vértice *j*. Si no existe un arco entre los vértices *i* y *j* se asume que C[i][j] es un valor muy grande (infinito). En cada paso d[i] contiene la longitud del camino más corto al vértice *i*. El pseudocódigo del algoritmo es el siguiente:

```
/* Calcula el costo del camino más corto desde el vértice f a cada vértice del
    digrafo definido por la matriz de costo G de orden n*/
Dijkstra(f, n, G)
{
 S={f};  /* asignación del nodo fuente */
 /* inicialización del array d */
 for (i=1; i<n; i++)
   d[i]=C[f][i];
 /* selección de los caminos más cortos */
 for (i=0; i<n-1; i++)
 {
  seleccionar un vértice w en V−S tal que d[w] es mínimo;
  añadir w a S;
  for cada vértice v en V-S
    d[v]=MIN(d[v], d[w] + C[w][v]);
 }
}
```

Ejemplo: Aplicar el algoritmo de Dijkstra al siguiente digrafo considerando como vértice inicial el 1:



Inicialmente:
S = 1
d[2]=10, d[3]=∞, d[4]=30, d[5]=100
la primera iteración del algoritmo produce w=2 como vértice seleccionado por tener un valor mínimo en D.
Por tanto:
d[3]=min(∞,10+50)=60
d[4], d[5] no cambian debido a que ir directamente desde 1 es más corto que a través del vértice 2.

La secuencia de los valores de d a través de las iteraciones del algoritmo producen los siguientes resultados:

Iteración	S	w	d[2]	d[3]	d[4]	d[5]
Inicial	{1}	–	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Si se desea reconstruir el camino más corto desde la fuente a cada vértice, podemos usar un array p de vértices, tal que p[*v*] contiene el vértice anterior a *v* en el camino. Inicializando p[*v*] a 1 para todo *v*=1. El array p puede actualizarse después de la última línea del algoritmo, siempre que d[*w*]+c[*w*][*v*]<d[*v*] entonces p[*v*]=*w*. Cuando termina el algoritmo, el camino a cada vértice se halla mediante una traza hacia atrás en el array p.

Ejemplo: Para el digrafo anterior el array p tiene los siguientes valores:

p[2]=1 p[3]=4 p[4]=1 p[5]=3

para obtener el camino de 1 a 5 se traza en orden inverso los predecesores empezando por el vértice 5. Del array p se determina que 3 es el predecesor, 4 es el predecesor de 3 y 1 es el predecesor de 4, por lo que el camino más corto del vértice 1 al 5 es 1,4,3,5.

El tiempo de ejecución del algoritmo, suponiendo que se usa la matriz de adyacencia para representar el digrafo y que el digrafo tiene *n* vértices y *e* aristas, se calcula considerando que el ciclo interno toma un tiempo O(*n*) y es ejecutado *n*−1 veces, lo que da un tiempo total de $O(n^2)$.

Si e es mucho menor que n^2, es mejor usar la representación de la lista de adyacencia y usar una cola de prioridad para los vértices en V−S. En este caso se demuestra que el tiempo requerido es de O(*e* log*n*).

Una implementación del algoritmo de Dijkstra se muestra a continuación y en la que puede apreciarse que ha sido necesario implementar el tipo de datos y operaciones sobre conjuntos:

```
#include <stdio.h>
```

```
#define MAX_NODOS 50
#define INFINITO 1e6
#define LIMITE   10000
#define MIN(x,y) (x<y)?x:y

main()
{
  void lee_grafo_fichero(int *n, float g[][MAX_NODOS]);
  void imprime_grafo(int n, float g[][MAX_NODOS]);
  void dijkstra(int e,int n, float g[][MAX_NODOS]);
  float grafo[MAX_NODOS][MAX_NODOS];
  int n;

  lee_grafo_fichero(&n,grafo);
  imprime_grafo(n,grafo);
  dijkstra(0,n,grafo);
  return(0);
}

void lee_grafo_fichero(int *n, float grafo[][MAX_NODOS])
{
  FILE *fp;
  int i,j;
  float a;

  fp=fopen("dijks.dat","r");
  fscanf(fp,"%d",n);
  for(i=0;i<*n;i++)
   for(j=0;j<*n;j++)
    {
     fscanf(fp,"%f",&a);
     if (a<LIMITE)
       grafo[i][j]=a;
     else
       grafo[i][j]=INFINITO;
    }
}

void imprime_grafo(int n, float grafo[][MAX_NODOS])
{
  int i,j;

  printf("Numero de nodos: ");
    printf("%d \n",n);
  printf("Matriz de adyacencia:\n");
  for(i=0;i<n;i++)
  {
   for(j=0;j<n;j++)
    if(grafo[i][j]!=INFINITO)
```

```
      printf("%6.2f ",grafo[i][j]);
     else
      printf("     ~ ");
    printf("\n");
   }
}

#define MAX_ELEM 255
typedef unsigned short int ELEMENTO;

ELEMENTO s[MAX_ELEM];
ELEMENTO v[MAX_ELEM];
ELEMENTO diff[MAX_ELEM];
void escribe_camino(int o,int c);
void inicia_conjunto(ELEMENTO c[]);
void inserta_elemento(ELEMENTO c[],int i);
void inserta_elementos(ELEMENTO c[],int n_elem);
int en_conjunto(ELEMENTO c[],int elem);
void diferencia_conjuntos(ELEMENTO dif[],ELEMENTO a[],ELEMENTO b[]);
int p[MAX_NODOS];

void dijkstra(int o,int n, float grafo[][MAX_NODOS])
{
 int i,j,j_min;
 float min,tmp;
 float d[MAX_NODOS];

 inicia_conjunto(s);
 inicia_conjunto(v);
 inicia_conjunto(diff);

 inserta_elemento(s,o);
 inserta_elementos(v,n);
 for(i=0;i<n;i++)
 {
  d[i]=grafo[o][i];
  p[i]=o;
 }
 diferencia_conjuntos(diff,v,s);
 for(i=0;i<n-1;i++)
 {
  min=INFINITO;
  for(j=0;j<n;j++)
   if(d[j]<min && en_conjunto(diff,j))
    {
     j_min=j;
     min=d[j];
    }
  inserta_elemento(s,j_min);
```

```
    diferencia_conjuntos(diff,v,s);
    for(j=0;j<n;j++)
     if (en_conjunto(diff,j))
     {
      tmp=d[j_min]+grafo[j_min][j];
      d[j]=MIN(d[j],tmp);
      if (d[j_min]+grafo[j_min][j]==d[j])
       p[j]=j_min;
     }
   }
   printf("Algoritmo de Dijkstra\n");
   printf("Costo de ir de %d a:\n",o+1);
   for(i=0;i<n;i++)
    if (i!=o)
     printf("%d --> %f\n",i+1,d[i]);


   printf("Ruta mas corta desde %d:\n",o+1);
   for(i=0;i<n;i++)
    if (i!=o)
    {
     printf("a %d:",i+1);
     escribe_camino(o,i);
     printf(" -> %d\n",i+1);
    }
}

void escribe_camino(int o,int c)
{
 if (p[c]==o)
   printf(" %d",o+1);
 else
 {
   escribe_camino(o,p[c]);
   printf(" -> %d",p[c]+1);
 }
}

void inicia_conjunto(ELEMENTO c[])
{
 int j;
 for (j=0;j<MAX_ELEM;j++)
   c[j]=0;
}

void inserta_elemento(ELEMENTO c[],int i)
{
   c[i]=1;
}
```

```
void inserta_elementos(ELEMENTO c[],int n_elem)
{
 int j;
 for (j=0;j<n_elem;j++)
   c[j]=1;
}

int en_conjunto(ELEMENTO c[],int elem)
{
  return (c[elem]);
}

void diferencia_conjuntos(ELEMENTO dif[],ELEMENTO a[],ELEMENTO b[])
{
 int j;
 for (j=0;j<MAX_ELEM;j++)
  dif[j]=a[j];
 for (j=0;j<MAX_ELEM;j++)
  if (b[j]==1)
   dif[j]=0;
}
```

El problema de todos los pares de caminos más cortos

Dado un digrafo G=(V,E) en el que cada arista $v_1 \rightarrow v_2$ tiene un costo (longitud) no negativo C[*v*][*w*], el problema consiste en encontrar para cada par ordenado de vértices (*v*,*w*) el menor costo (longitud) del camino de *v* a *w*.

Este problema puede resolverse usando el algoritmo de Dijkstra tomando cada vértice como fuente. Existe, sin embargo, una manera más directa de resolver este problema usando el algoritmo de Floyd. Asumiendo que los vértices V se numeran 1,2,….,*n*, el algoritmo de Floyd usa una matriz A de $n \times n$ en el que se calculan las longitudes de los caminos más cortos para cada iteración. Inicialmente se asigna A[*i*][*j*]=C[*i*][*j*] para todo $i \neq j$. Si no hay una arista de *i* a *j* se considera C[*i*][*j*]= ∞ y los elementos diagonales se ponen a 0.

A continuación se realizan *n* iteraciones sobre la matriz A. Después de la *k*-ésima iteración, A[*i*][*j*] contendrá el valor de la longitud más corta de cualquier camino del vértice *i* al vértice *j* que no pasa a través de un vértice numerado mayor que *k*. Otra forma de ver esto es que *i* y *j*, los vértices finales del camino, los vértices finales del camino, pueden ser cualquier vértice pero los vértices intermedios del camino deben ser menores o iguales a *k*.

En la *k*-ésima iteración se usa la siguiente fórmula para calcular A:

$$A_k[i][j] = \min \begin{cases} A_{k-1}[i][j] \\ A_{k-1}[i][k] + A_{k-1}[k][j] \end{cases}$$

El subíndice *k* indica el valor de la matriz A después de la *k*-ésima iteración y no debe entenderse como que hay *n* matrices diferentes.

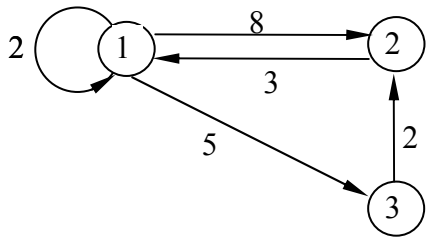
Ejemplo: Para el siguientes digrafo calcular la matriz A y su contenido después de tres iteraciones.

$A_0[i][j]$	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_1[i][j]$	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_2[i][j]$	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_3[i][j]$	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0



Como $A_k[i][k] = A_{k-1}[i][k]$ y $A_k[k][j] = A_{k-1}[k][j]$ ninguna entrada con subíndice igual a k cambia durante la k-ésima iteración. Por ello, es posible usar una sola copia de la matriz A para llevar a cabo el cómputo. Un programa que realiza este cálculo se da a continuación:

```
void floyd(int n, float c_inicial[][MAX_NODOS],float c_final[][MAX_NODOS])
{
 int i,j,k;

 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    c_final[i][j]=c_inicial[i][j];
 for(i=0;i<n;i++)
   c_final[i][i]=0;
 for(k=0;k<n;k++)
  for(i=0;i<n;i++)
   for(j=0;j<n;j++)
    if (c_final[i][k]+c_final[k][j]<c_final[i][j])
     c_final[i][j]= c_final[i][k]+c_final[k][j];
}
```

Este algoritmo toma un tiempo $O(n^3)$. Para digrafos esparsos es mejor usar la versión de Dijkstra con lista de adyacencia, que toma un tiempo $O(ne\log n)$.

Si se desea imprimir el camino más corto de un vértice a otro, una forma de lograrlo es usar una matriz p, donde $p[i][j]$ almacena el vértice k que permite al algoritmo de Floyd encontrar el menor valor de $A[i][j]$. Si $p[i][j]=0$ entonces el camino más corto entre i y j es directo, según el arco $i \rightarrow j$.

Se puede implementar un procedimiento recursivo para imprimir los vértices intermedios del camino más corto entre i y j.

Ejemplo: la matriz final p para el digrafo del ejemplo anterior es:

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

Una implementación del algoritmo junto con la función que imprime las rutas se muestran a continuación:

```
#include <stdio.h>
#define MAX_NODOS 50
#define INFINITO 1e6
#define LIMITE   10000

main()
{
 void lee_costo_fichero(int *n, float g[][MAX_NODOS]);
 void imprime_grafo(int n, float g[][MAX_NODOS]);
 void floyd(int n, float c[][MAX_NODOS],float d[][MAX_NODOS]);
 float costo_inicial[MAX_NODOS][MAX_NODOS],
   costo_final[MAX_NODOS][MAX_NODOS];
 int n;

 lee_costo_fichero(&n,costo_inicial);
 imprime_grafo(n,costo_inicial);
 floyd(n,costo_inicial,costo_final);
 return(0);
}

void lee_costo_fichero(int *n, float grafo[][MAX_NODOS])
{
 FILE *fp;
 int i,j;
 float a;

 fp=fopen("floyd.dat","r");
 fscanf(fp,"%d",n);
 for(i=0;i<*n;i++)
  for(j=0;j<*n;j++)
  {
   fscanf(fp,"%f",&a);
   if (a<LIMITE)
     grafo[i][j]=a;
   else
     grafo[i][j]=INFINITO;
  }
}

void imprime_grafo(int n, float grafo[][MAX_NODOS])
{
 int i,j;

 printf("Numero de nodos: ");
   printf("%d \n",n);
```

```
  printf("Matriz de adyacencia:\n");
  for(i=0;i<n;i++)
  {
   for(j=0;j<n;j++)
    if(grafo[i][j]!=INFINITO)
     printf("%6.2f ",grafo[i][j]);
    else
     printf("     ~ ");
   printf("\n");
  }
}

int p[MAX_NODOS][MAX_NODOS];

void floyd(int n, float c_inicial[][MAX_NODOS],float c_final[][MAX_NODOS])
{
  int i,j,k;
  void escribe_rutas(int n,float c[][MAX_NODOS]);

  for(i=0;i<n;i++)
   for(j=0;j<n;j++)
   {
    c_final[i][j]=c_inicial[i][j];
    p[i][j]=0;
   }
  for(i=0;i<n;i++)
    c_final[i][i]=0;
  for(k=0;k<n;k++)
   for(i=0;i<n;i++)
    for(j=0;j<n;j++)
     if (c_final[i][k]+c_final[k][j]<c_final[i][j])
     {
      c_final[i][j]= c_final[i][k]+c_final[k][j];
      p[i][j]=k+1;
     }
  imprime_grafo(n,c_final);
  escribe_rutas(n,c_final);
}

void escribe_rutas(int n,float costo[][MAX_NODOS])
{
  int i,j;
  void ruta(int o,int c);

  printf("Algoritmo de Floyd\n");
  printf("Costo Rutas mas corta:\n");
  for(i=0;i<n;i++)
   for(j=0;j<n;j++)
   if (i!=j && costo[i][j]!=INFINITO)
```

```
        {
         printf(" %.2f ",costo[i][j]);
         printf(" %d ->",i+1);
         ruta(i,j);
         printf(" %d\n",j+1);
        }
      }

      void ruta(int o,int c)
      {
       int k;

       k=p[o][c];
       k--;
       if (k>0)
       {
         ruta(o,k);
         printf(" %d ->",k+1);
         ruta(k,c);
       }
      }
```