Originalmente publicado en los números 155 (enunciado) y 156 (solución) de la revista Novática
(http://www.ati.es/novatica/)

Problema A: Configuración de un aeropuerto

M. Carro
mcarro@fi.upm.es

A. Herranz
aherranz@fi.upm.es

J. Mariño
jmarino@fi.upm.es

P. Sánchez
psanchez@skyrealms.org

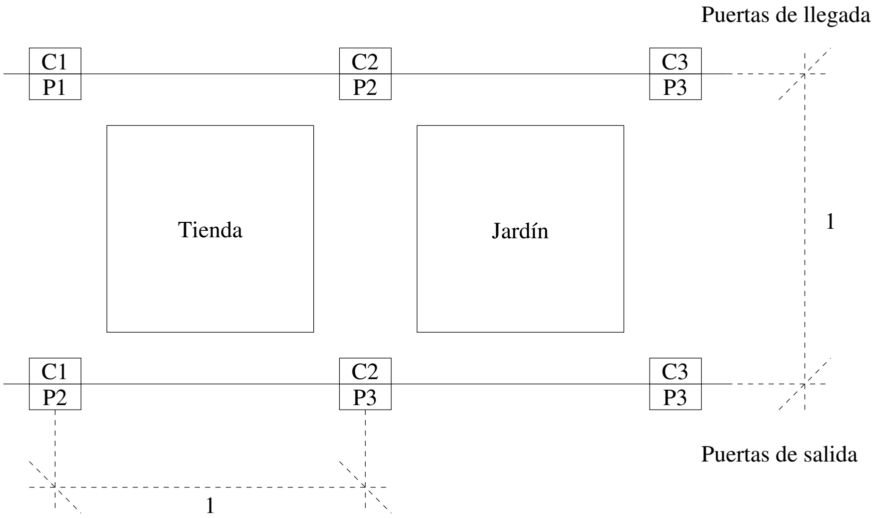
1 Descripción del problema

Aerolíneas ACM es una compañía aérea regional con sede en el aeropuerto von Neumann. Para muchos pasajeros, el aeropuerto von Neumann no es el punto de partida de su viaje ni su destino final, así que tienen lugar muchas escalas en el aeropuerto.

El aeropuerto von Neumann tiene la disposición de un pasillo. Las puertas de llegada están situadas, equidistantemente, en el lado norte de dicho pasillo; las puertas de salida están situadas en el lado sur, también de forma equidistante. La distancia entre dos puertas contiguas es igual al ancho del pasillo. A cada puerta de llegada se corresponde con una ciudad exactamente, y lo mismo ocurre con las puertas de salida. Los pasajeros entran por la puerta de llegada de su ciudad de procedencia y abandonan la terminal, o conectan con el siguiente vuelo en la puerta correspondiente a su ciudad de destino. Para este problema, sólo consideramos pasajeros con cambio de vuelo.

Las escalas generan en el pasillo un tráfico de pasajeros considerable. El número medio de personas que viajan entre ciudades se conoce de antemano. Usando esta información, debería ser posible reducir el tráfico. Si las escalas entre las ciudades Cx y Cy son muy frecuentes, puede ser útil situar las puertas correspondientes a esas ciudades cerca o incluso directamente enfrente una de la otra.

Debido a la presencia de tiendas y jardines, no es posible cruzar el pasillo oblicuamente, así que la distancia entre las puertas P1, de llegada, y P3, de salida, es $1 + 2 = 3$ (véase el diagrama).



Debes calcular la carga de tráfico total para varias configuraciones distintas. La carga de tráfico entre una puerta de origen y otra de destino se define como el número de pasajeros con esos puntos de origen y destino, multiplicado por la distancia entre las puertas de llegada y de salida. La carga de tráfico total es la suma de las cargas de tráfico para todos los pares origen-destino.

Descripción de la entrada

El archivo de entrada contiene varios casos de prueba. El último caso de prueba del archivo de entrada va seguido por una línea que contiene el número 0.

Cada caso de prueba tiene dos partes: primero los datos del tráfico y luego la configuración.

Los datos de tráfico empiezan con un entero N ($1 < N < 25$), que representa el número de ciudades. Cada una de las siguientes N líneas representa los datos de tráfico para una ciudad. Cada línea con los datos de tráfico empieza con un entero del rango $1..N$ que identifica la ciudad de origen. Esto va seguido por k pares de enteros, un par para cada ciudad de destino. Cada par identifica la ciudad de destino y el número de pasajeros (500 a lo más) que viajan de la ciudad de origen a la de destino.

Los datos de configuración consisten en una o más (como mucho 20) configuraciones y termina con una línea que contiene el número 0.

Una configuración contiene 3 líneas. La primera línea contiene un número positivo que identifica la configuración. La siguiente línea contiene una permutación de las ciudades, indicando cómo se corresponden con las puertas de llegada: el primer número representa la ciudad correspondiente a la primera puerta y así sucesivamente. Del mismo modo, la siguiente línea representa cómo las ciudades se corresponden a las puertas de salida.

Descripción de la salida

Para cada caso de prueba, la salida contiene una tabla que presenta los números de configuración y la carga total de tráfico, en orden ascendente de carga. Si dos configuraciones tienen la misma carga de tráfico, debe presentarse primero la que tiene la que tiene el número de configuración más bajo. Sigue la disposición de la salida mostrada en el ejemplo siguiente.

Ejemplo de entrada

```
3
1 2 2 10 3 15
2 1 3 10
3 2 1 12 2 20
1
1 2 3
2 3 1
2
2 3 1
3 2 1
0
2
1 1 2 100
2 1 1 200
1
1 2
1 2
2
1 2
2 1
0
0
```

Salida para el ejemplo de entrada

```
Configuration  Load
    2          119
    1          122
Configuration  Load
    2          300
    1          600
```

2 Discusión preliminar

El problema propuesto en el número pasado tiene como tema central averiguar el tráfico entre puertas de un aeropuerto de paso. En un principio la intuición parece sugerir que se pide una configuración óptima de puertas de llegada y salida. Esto no es así: en realidad se limita a solicitar un cálculo de la cantidad de tráfico entre cada puerta de llegada y de salida. Este tráfico se calcula como la multiplicación del número de pasajeros que transitan entre ambas por la distancia que las separa, utilizando una *distancia de Manhattan* [Gar81] para medir esta separación.

Como inciso, no es extraño que un problema resulte a primera vista más difícil de lo que es — y también a la inversa, claro. En el caso de los concursos de programación, la experiencia dice que la presión del tiempo y el nerviosismo provocan malentendidos de los enunciados que, al menos, retrasan el llegar a una solución satisfactoria.

Volviendo al problema, los datos de entrada proponen una serie de casos diferentes. Cada uno de ellos especifica un régimen de tráfico entre ciudades y una serie de distribuciones de puertas de llegada y salida diferentes. Se pide hallar el tráfico total para cada una de las configuraciones y ordenarlas según este tráfico. Aparentemente, esta simulación ayudaría a las autoridades aeroportuarias a tener una asignación vuelos/puertas que minimizase la carga de tráfico interna. Si tenemos en cuenta que para 20 destinos de llegada y otros tantos de salida (una situación razonable para un aeropuerto de tamaño medio) hay unas $5.9 \cdot 10^{36}$ configuraciones, no debe extrañar el funcionamiento de algunos aeropuertos.

3 Análisis rápido, diseño, e implementación

La tarea que hay que realizar con cada configuración es la misma: leer los datos de tránsito de pasajeros (solo una vez) y leer la configuración de entrada y salida. La labor principal, por tanto, es calcular la suma

$$T = \sum_{i,j \in C} t_{ij} d_{ij}$$

donde C es el conjunto de ciudades conectadas, t_{ij} es el tránsito entre la ciudad i y la j y d_{ij} es la distancia entre ellas. La mayor dificultad es crear las matrices d y t. Tanto i como j son naturales menores que 25, así que una definición C como

```
int trafico[25][25];
```

basta para almacenar la información del tráfico. Los datos sobre el tráfico entre ciudades son un grafo con pesos en los arcos dado en formato de listas de adyacencia, del que se puede crear directamente la matriz de tráfico (procedimiento `LeerTrafico`):

```
for (i = 1; i <= nciudades; i ++)
  for (j = 1; j <= nciudades; j ++)
    trafico[i][j] = 0;

for (i = 1; i <= nciudades; i ++) {
  scanf("%d", &origen);              /* Origen de la adyacencia           */
```

```
  scanf("%d", &nadyacentes);          /* Número de adyacentes a esa ciudad */

  for (j = 1; j <= nadyacentes; j ++) { /* Adyacentes y tráfico con ellos  */
    scanf("%d %d", &destino, &pasajeros);

    trafico[origen][destino] = pasajeros; /* El peso del arco son los      */
                                          /* pasajeros que transitan       */
  }
}
```

Con esto ya tenemos creada la matriz t_{ij}. El formato de entrada para la matriz d_{ij} es algo distinto: en cada configuración, dos líneas de números

$$\begin{array}{cccc} a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \end{array}$$

según las cuales $d_{a_i b_j} = |i - j| + 1$. Es posible, por tanto, utilizar los identificadores de ciudades como índice de la matriz. Esto, naturalmente, concuerda con el tipo de la matriz de tráfico. Un programador avezado puede no crear la matriz explícita, sino almacenar cada valor $i, (1 \leq i \leq n)$ en un vector indexado por a_i. En cada lectura de un b_j sólo es necesario recorrer, para esa ciudad b_j, el vector a, extraer la posición i para cada una de las ciudades de ese vector, y calcular la distancia entre ambas puertas como $|i - j| + 1$. Esa distancia se multiplica por el valor de tráfico correspondiente entre dos ciudades, que se va acumulando. El código correspondiente (procedimiento `ProcesarConfiguracion`) sería:

```
int i, origen, destino;
int origenes[25];                    /* La posición de la puerta de origen */
                                     /* dentro de la permutación           */
int carga = 0;                       /* Carga de la configuración          */

/* Se almacena la permutación de destinos, pero almacenando para cada      */
/* destino, su posición en la permutación (para obtener un acceso directo) */

for (i = 1; i <= nciudades; i ++) {
  scanf("%d", &origen);
  origenes[origen] = i;
}

for (i = 1; i <= nciudades; i ++) {
  scanf("%d", &destino);                                     /* El origen */

  for (origen = 1; origen <= nciudades; origen ++)
    /* El tráfico que se añade a la configuración resulta:               */
    /*  distancia = diferencia horizontal + 1                            */
    /*                                   ^                               */
    /*                         (diferencia vertical)                     */
    /* sumar: distancia * tráfico entre las puertas                      */
    carga += (abs(i - origenes[origen]) + 1) * trafico[origen][destino];
}
```

Por último, cada configuración correspondiente a unos mismos datos de tráfico genera una carga que es necesario almacenar para posteriormente ordenar las configuraciones según su carga (y, en caso de empate, según un identificador de configuración contenido en los datos de entrada).

```
typedef struct {
```

```
  int id;                               /* Identificador de la configuración */
  int carga;                            /* Carga de la configuración         */
} TConfiguracion;

int nciudades;                          /* Número de ciudades en la entrada  */
int trafico[25][25];                    /* Tráfico entre dos ciudades        */
                                        /* (es una matriz de adyacencia)     */

int nconfs;                             /* Número de configuraciones         */
TConfiguracion confs[25];               /* Vector de configuraciones         */
```

Lo más cómodo para la ordenación es utilizar una función de biblioteca, como `qsort`:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

que requiere datos acerca del vector que se va a ordenar y una función de comparación. En este caso la función de comparación tiene en cuenta primero la carga y después el identificador asociado.

```
int Comparacion (const void *a, const void *b)
{
  TConfiguracion conf1 = *((const TConfiguracion *) a);
  TConfiguracion conf2 = *((const TConfiguracion *) b);

  return (conf1.carga == conf2.carga ?
          conf1.id - conf2.id :
          conf1.carga - conf2.carga);
}
```

Por último, el bucle principal tendría que leer tráfico y configuraciones para cada uno de los casos de entrada:

```
LeerTrafico();      /* Lectura de la adyacencia (pasajeros entre puertas) */
                           /* Proceso de las diferentes configuraciones */
nconfs = 0;
scanf("%d", &configuracion);         /* Identificador de la configuración */
while (configuracion) {
  confs[nconfs].id = configuracion;
  confs[nconfs].carga = ProcesarConfiguracion();
  nconfs ++;
  scanf("%d", &configuracion);
}

                                     /* Ordenación de las configuraciones */
qsort(confs, nconfs, sizeof(TConfiguracion), Comparacion);

                    /* Impresión del resultado ordenado según se requiere */
printf("Configuration  Load\n");
for (i = 0; i < nconfs; i ++) {
  printf("%5d%10s%d\n", confs[i].id, "", confs[i].carga);
}
```

En definitiva, el mayor contratiempo de este problema es entender exactamente qué se está pidiendo y cómo extraer esa información de los datos de entrada.

4 Otra visión de la implementación

Veamos una implementación en otro lenguaje siguiendo la misma filosofía que la anterior. El lenguaje elegido ha sido Prolog, pero no se utiliza ninguna característica específica del lenguaje (*backtracking*, uso de variables lógicas, llamadas de orden superior...) de modo que debe ser rápidamente extrapolable a cualquier otro lenguaje lógico (Mercury, Gödel...), funcional (Haskell, Hope, ML...) o híbrido (Babel, Curry...).

Asumiremos que los datos ya han sido leídos, pero no que haya habido ninguna transformación esencial de los mismos, de modo que el programa trabaja sobre los mismos datos presentes a la entrada. Una definición de tráfico, como

```
3
1 2 2 10 3 15
2 1 3 10
3 2 1 12 2 20
```

se transforma de forma inmediata, mediante una sencilla lectura, en una lista de adyacencia como:

```
[lst(1,[(2,10),(3,15)]), lst(2,[(3,10)]), lst(3,[(1,12),(2,20)])]
```

Y una configuración como

```
1 2 3
2 3 1
```

se traduce a un par de listas de ciudades como

```
[1, 2, 3]
[2, 3, 1]
```

Con esto, la tarea se *reduce* a programar el predicado `carga(Trafico, Entradas, Salidas, Coste)`, que será llamado como

```
carga([lst(1,[(2,10),(3,15)]), lst(2,[(3,10)]), lst(3,[(1,12),(2,20)])],
      [1, 2, 3], [2, 3, 1], C).
```

En la implementación se ha optado por abstraer la matriz de adyacencia, y generar los valores de tráfico entre las ciudades dinámicamente a partir de la descripción inicial:

```
cant_trafico(Trafico, Entr, Sal, Coste):-
        member(lst(Entr, Lst), Trafico),
        member((Sal, Coste), Lst),
        !.
cant_trafico(_Trafico, _Entr, _Sal, 0).
```

Si bien esto penaliza la complejidad del programa, no es difícil realizar una traducción previa a *funtores* que daría un acceso con complejidad $O(1)$ o una basada en árboles 2-3 que daría una en $O(\log n)$.

El resto del programa, una vez visto el análisis inicial, es bastante sencillo: dos bucles, uno para recorrer las ciudades de entrada, y, para cada una de ellas, otro para recorrer las ciudades de salida. En aras de una mayor claridad, se ha obviado el uso de parámetros de acumulación, que usualmente disminuye el consumo de memoria e incrementa la velocidad del programa.

```
%% Coste se calcula para una relacion de transbordo dada por Trafico
%% entre las ciudades ordenadas en Entradas y Salidas
carga(Trafico, Entradas, Salidas, Coste):-
        procesa(Entradas, 1, Salidas, Trafico, Coste).
```

```
%% Para cada ciudad en posición IndEnt, se calcula el Coste con respecto a
%% todas las ciudades en Salidas, y se continúa con el resto de las ciudades
%% IndEnt es la posición de la ciudad Entr
procesa([], _, _, _, 0).
procesa([Entr|Entradas], IndEnt, Salidas, Trafico, Coste):-
        proc_salidas(Salidas, 1, Entr, IndEnt, Trafico, C1),
        IndEnt1 is IndEnt + 1,
        procesa(Entradas, IndEnt1, Salidas, Trafico, C2),
        Coste is C1 + C2.

%% Para cada ciudad de salida en posición IndSal, se calcula su coste
%% con respecto a la ciudad en posición IndEnt.
%% IndSal es la posición de la ciudad Sal; igual con IndEnd y Entr
proc_salidas([], _, _, _, _, 0).
proc_salidas([Sal|Salidas], IndSal, Entr, IndEnt, Trafico, Coste):-
        cant_trafico(Trafico, Entr, Sal, Tr),
        C1 is Tr*(abs(IndEnt-IndSal)+1),
        IndSal1 is IndSal + 1,
        proc_salidas(Salidas, IndSal1, Entr, IndEnt, Trafico, C2),
        Coste is C1 + C2.
```

Referencias

[Gar81] Martin Gardner. Juegos matemáticos. *Investigación y Ciencia*, páginas 114–118, Enero 1981.