novática
UPGRADE
Revista de la Asociación de Técnicos de Informática
Edición Digital
MARZO - ABRIL 2002
156

Novática, revista fundada en 1975, es el órgano oficial de expresión y formación continua de ATI (Asociación de Técnicos de Informática)

ATI es miembro de CEPIS *(Council of European Professional Informatics Societies)* y tiene un acuerdo de colaboración con ACM (Association for Computing Machinery). Tiene asimismo acuerdos de vinculación o colaboración con AdaSpain, AI^2 y ASTIC

<http://www.ati.es/novatica/>

CONSEJO EDITORIAL

Antoni Carbonell Nogueras, Francisco López Crespo, Julián Marcelo Cocho, Celestino Martín Alonso, Josep Molas i Bertrán, Roberto Moya Quiles, Gloria Nistal Rosique (Presidenta del Consejo), César Pérez Chirinos, Mario Piattini Velthuis, Fernando Piera Gómez, Miquel Sàrries Griñó, Carmen Ugarte García, Asunción Yturbe Herranz

Coordinación Editorial
Rafael Fernández Calvo <*rfcalvo@ati.es*>

Composición y autoedición
Jorge Llácer

Administración
Tomás Brunete, María José Fernández

SECCIONES TECNICAS: COORDINADORES

Arquitecturas
Antonio Gonzalez Colás (DAC-UPC) <*antonio@ac.upc.es*>
Bases de Datos
Coral Calero Muñoz, Mario G. Piattini Velthuis
(Escuela Superior de Informática, UCLM)
<*Coral.Calero@uclm.es*>, <*mpiattin@inf-cr.uclm.es*>
Calidad del Software
Juan Carlos Granja (Universidad de Granada) <*jcgranja@goliat.ugr.es*>
Derecho y Tecnologías
Isabel Hernando Collazos (Fac. Derecho de Donostia, UPV)
<*ihernando@legaltek.net*>
Enseñanza Universitaria de la Informática
Cristóbal Pareja Flores (Dep. Sistemas Informáticos y Programación-UCM)
<*cpareja@sip.ucm.es*>
Informática Gráfica
Roberto Vivó (Eurographics, sección española) <*rvivo@dsic.upv.es*>
Ingeniería del Software
Luis Fernández (PRIS-E.I./UEM) <*lufern@dpris.esi.uem.es*>
Inteligencia Artificial
Federico Barber,Vicente Botti (DSIC-UPV)
<*{vbotti, fbarber}@dsic.upv.es*>
Interacción Persona-Computador
Julio Abascal González (FI-UPV) <*julio@si.ehu.es*>
Internet
Alonso Álvarez García (TID) <*alonso@ati.es*>
Llorenç Pagés Casas (Atlante) <*pages@ati.es*>
Lengua e Informática
M. del Carmen Ugarte (IBM) <*cugarte@ati.es*>
Lenguajes informáticos
Andrés Marín López (Univ. Carlos III) <*amarin@it.uc3m.es*>
J. Ángel Velázquez (ESCET-URJC) <*a.velazquez@escet.urjc.es*>
Libertades e Informática
Alfonso Escolano (FIR-Univ. de La Laguna) <*aescolan@ull.es*>
Lingüística computacional
Xavier Gómez Guinovart (Univ. de Vigo) <*jgomez@uvigo.es*>
Manuel Palomar (Univ. de Alicante) <*mpalomar@dlsi.ua.es*>
Profesión informática
Rafael Fernández Calvo (ATI) <*rfcalvo@ati.es*>
Miquel Sarries Grinyó (Ayto. de Barcelona) <*msarries@ati.es*>
Seguridad
Javier Areitio (Redes y Sistemas, Bilbao) <*jareitio@orion.deusto.es*>
Sistemas de Tiempo Real
Alejandro Alonso, Juan Antonio de la Puente (DIT-UPM)
<*{aalonso,jpuente}@dit.upm.es*>
Software libre
Jesús M. González Barahona, Pedro de las Heras Quirós
(GSYC, URJC) <*{jgb,pheras}@gsyc.escet.urjc.es*>
Tecnología de Objetos
Esperanza Marcos (URJC) <*e.marcos@escet.urjc.es*>
Gustavo Rossi (LIFIA-UNLP, Argentina) <*gustavo@sol.info.unpl.edu.ae*>
Tecnologías para la Educación
Benita Compostela (F. CC. PP.- UCM) <*benita@dial.eunet.es*>
Josep Sales Rufí (ESPIRAL) <*jsales@pie.xtec.es*>
Tecnologías y Empresa
Pablo Hernández Medrano <*phmedrano@terra.es*>
TIC para la Sanidad
Valentín Masero Vargas (DI-UNEX) <*vmasero@unex.es*>

Las opiniones expresadas por los autores son responsabilidad exclusiva de los mismos. Novática permite la reproducción de todos los artículos, salvo los marcados con © o copyright, debiéndose en todo caso citar su procedencia y enviar a Novática un ejemplar de la publicación.

Coordinación Editorial y Redacción Central (ATI Madrid)
Padilla 66, 3º, dcha., 28006 Madrid
Tlf.914029391; fax.913093685 <*novatica@ati.es*>
Composición, Edición y Redacción ATI Valencia
Palomino 14, 2ª, 46003 Valencia
Tlf./fax 963918531 <*secreval@ati.es*>
Administración, Subscripciones y Redacción ATI Cataluña
Via Laietana 41, 1º, 1ª, 08003 Barcelona
Tlf.934125235; fax 934127713 <*secregen@ati.es*>
Redacción ATI Andalucía
Isaac Newton, s/n, Ed. Sadiel, Isla Cartuja 41092 Sevilla
Tlf./fax 954460779 <*secreand@ati.es*>
Redacción ATI Aragón
Lagasca 9, 3-B, 50006 Zaragoza
Tlf./fax 976235181 <*secreara@ati.es*>
Redacción ATI Asturias-Cantabria <*gp-astucant@ati.es*>
Redacción ATI Castilla-La Mancha <*gp-clmancha@ati.es*>
Redacción ATI Galicia
Recinto Ferial s/n, 36540 Silleda (Pontevedra)
Tlf.986581413; fax 986580162 <*secregal@ati.es*>

Publicidad: Padilla 66, 3º, dcha., 28006 Madrid
Tlf.914029391; fax.913093685 <novatica.publicidad@ati.es>

Imprenta: 9 Impressió S.A., Juan de Austria 66, 08005 Barcelona.
Depósito Legal: B 15.154-1975
ISBN: 0211-2124; CODEN NOVAEC

Portada: Antonio Crespo Foix / © ATI 2002

SUMARIO

En resumen: **¿eXtremismo? Sí, gracias** **3**
Rafael Fernández Calvo

Monografía: «*eXtreme Programming* / Programación eXtrema»
(En colaboración con ***Informatik/Informatique*** y ***Upgrade***)
Editor invitado: *Luis Fernández Sanz*

Presentación: *eXtreme Programming* y la mejora en el desarrollo del software **5**
Luis Fernández Sanz

Referencias útiles sobre *eXtreme Programming* **7**
Luis Fernández Sanz

***eXtreme Programming* (XP): un nuevo método de desarrollo de software** **8**
César F. Acebal, Juan M. Cueva Lovelle

La necesidad de velocidad: automatización de las pruebas de aceptación en un entorno de Programación Extrema **13**
Lisa Crispin, Tip House, con la contribución de *Carol Wade*

Estudios cualitativos sobre XP en una empresa de tamaño mediano **22**
Robert Gittins, Sian Hope, Ifor Williams

XP en proyectos complejos: algunas extensiones **27**
Martin Lippert, Stefan Roock, Henning Wolf, Heinz Züllighoven

XP e Ingeniería del Software: una opinión **32**
Luis Fernández Sanz

Programación Extremista **36**
Michael McCormick

Secciones Técnicas

Informática Gráfica
Modelado geométrico para visualización en tiempo real **39**
Inmaculada Remolar, José Ribelles, Óscar Belmonte, Miguel Chover, Cristina Rebollo

Interacción Persona-Computador
Modelos y herramientas para diseño y evaluación de la interfaz de usuario **44**
Fabio Paternò, Laila Paganelli, Carmen Santoro

Ingeniería Semiótica y comunicabilidad de las interfaces de usuario **49**
Clarisse Sieckenius de Souza

Software libre
Un entorno para enseñanza basado en software libre **55**
José Alfonso Accino

Tecnologías y Empresa
***Knowledge Management* en *su* organización** **61**
Pablo Hernández Medrano

Referencias autorizadas **65**

Sociedad de la Información

Personal y transferible
Mi opinión sobre las patentes de software en Europa **70**
Jesús M. Gonzalez-Barahona

Programar es crear
¡Queso! **71**
25º Concurso Internacional de Programación ACM (2001): problema B

«Configuración de un aeropuerto»: solución **72**
Manuel Carro, Ángel Herranz, Julio Mariño, Pablo Sánchez

Asuntos Interiores

Programación de Novática **76**
Normas de publicación para autores / Socios Institucionales **77**

Monografía del próximo número: «*Recuperación de la información y la Web*»

Programar es crear

Manuel Carro, Ángel Herranz, Julio Mariño, Pablo Sánchez

<mcarro@fi.upm.es>, <aherranz@fi.upm.es>, <jmarino@fi.upm.es>, <psanchez@skyrealms.org>

«Configuración de un aeropuerto»: solución

El enunciado de este problema apareció en el número 155 de Novática (enero-febrero 2002, p. 71). Es el programa A de los planteados en el 25º Concurso Internacional de Programación de la ACM (2001)

1. 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*[1] 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.

2. 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

$$a_1\ a_2 \ldots a_n$$
$$b_1\ b_2 \ldots b_n$$

según las cuales . 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,$(1 \blacktriangledown i \blacktriangledown n)$ en un vector indexado por a_i. En cada lectura de un b_i 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 $|i\text{-}j|+1$. 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.

3. 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

[Martin 1981] Martin Gardner: *Juegos Matemáticos*. Investigación y Ciencia, enero 1981, pp. 114-118.