novática
UPGRADE
Revista de la Asociación de Técnicos de Informática
JULIO - AGOSTO 2002
158

SUMARIO

Monografía del próximo número: «Inteligencia Artificial»

***Novática*, revista fundada en 1975, es el órgano oficial de expresión y formación continua de ATI (Asociación de Técnicos de Informática). *Novática* publica también *Upgrade*, revista digital de CEPIS (*Council of European Professional Informatics Societies*), en lengua inglesa.**

<http://www.ati.es/novatica/>
<http://www.upgrade-cepis.org/>

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

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@ati.es*>

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

Portada: Antonio Crespo Foix / © ATI 2002

Programar es crear

Crucigramas: solución

Cristóbal Pareja Flores, José Alberto Verdejo López

<cpareja@sip.ucm.es>, <alberto@sip.ucm.es>

El enunciado de este problema apareció en el número 157 de *Novática* (mayo-junio 2002, p. 72). Es el programa C de los planteados en el 25º Concurso Internacional de Programación de la ACM (2001)

1. Un problema típico de vuelta atrás

El problema planteado en el número anterior consiste en resolver un crucigrama conociendo las palabras que se han de colocar y una palabra más que sobra. Vamos a resolver este problema utilizando la conocida técnica de *vuelta atrás*. Con ella se realiza una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una (o todas) que satisfaga los criterios exigidos, o hasta constatar que tal solución no existe. Esta búsqueda puede ser impracticable si el conjunto de posibles soluciones es muy grande, por lo que es imprescindible estructurar el espacio por explorar, tratando de descartar en bloque conjuntos de posibles situaciones que no conducen a ninguna solución. Para que esto sea posible, las soluciones deben poder ser construidas por etapas; en cada etapa se extiende la solución, posiblemente de varias formas distintas. De esta forma el conjunto de posibles soluciones se puede organizar en forma de *árbol de exploración*, donde en cada nivel se toma la decisión de la etapa correspondiente. Un elemento adicional imprescindible para la aplicabilidad de esta técnica lo constituyen los denominados *criterios de poda* que permiten determinar cuándo una solución parcialmente construida nunca va a conducir a una solución completa satisfactoria, por lo que es inútil seguir buscando a partir de ella.

2. Aplicación a la resolución de crucigramas

Veamos cómo aplicamos esta técnica a nuestro problema. Los datos iniciales del crucigrama consisten en la información de los N huecos (es decir, la posición de la casilla inicial, fila y columna; y la dirección en la que hay que colocar la palabra, horizontal o vertical) y las $N + 1$ palabras que hay que intentar colocar. Las soluciones (parciales) van a representarse mediante vectores de N posiciones con la información inicial de los N huecos, que es *estática*: no varía durante la ejecución. Lo que sí varía, y representa las decisiones que hay que tomar, es la colocación de las palabras. Por lo tanto una solución tendrá además, para cada posición, la palabra que se ha colocado en ella. Utilizamos el siguiente tipo para representar las soluciones,

```
typedef struct {
  int fila, columna;
  char direccion;
  int posicionPalabra;
} Dato;
```

donde para cada dato, los tres primeros campos no variarán durante la ejecución, y el cuarto, que representa el número de orden de la palabra colocada en el correspondiente hueco (es decir, la posición que ocupa la palabra en el vector con todas las palabras), sí variará de unas soluciones a otras.

Los crucigramas tienen un número fijo de filas y columnas, determinado por las siguientes constantes:

```
const int ANCHO = 10;
const int ALTO  = 10;
```

Las etapas de la construcción de una solución van a corresponder a colocar una palabra en el siguiente hueco, donde los huecos se recorren de forma lineal. Cada palabra que decidimos colocar en un hueco no debe haber sido ya utilizada anteriormente. La comprobación de si una palabra aparece ya colocada en la solución parcial que estamos construyendo se hará consultando el vector `usada` que, para cada palabra, dirá si aparece ya en la solución o no. La palabra también tiene que caber en el crucigrama, situada desde la posición inicial del hueco y en la dirección adecuada:

```
bool cabe(string palabras[],
          Dato datos[], int k) {
  int lon =
palabras[datos[k].posicionPalabra].size();
  switch (datos[k].direccion) {
    case 'H' : return
    (datos[k].columna + lon - 1 <= ANCHO);
    case 'V' : return
    (datos[k].fila + lon - 1 <= ALTO);
  }
}
```

Y si al colocarla se cruza con alguna otra palabra ya colocada, la letra en la intersección debe ser la misma. Del enunciado se deduce que dos palabras pueden solaparse sólo si se cruzan, es decir, si cada una está en una dirección distinta: en efecto, si dos palabras se solaparan estando en la misma dirección (un prefijo de una coincide con un sufijo de la otra o una es una subcadena de la otra) entonces habría una secuencia máxima distinta de alguna de las palabras dadas. La función `dosOK` comprueba si no hay colisiones entre las palabras colocadas en las posiciones `i` y `k`:

```
bool dosOK(string palabras[],
     Dato datos[], int i, int k) {
  int f1 = datos[i].fila;
  int c1 = datos[i].columna;
  string p1 =
   palabras[datos[i].posicionPalabra];
```

```
  int l1 = p1.size();
  int f2 = datos[ k] .fila;
  int c2 = datos[ k] .columna;
  string p2 =
   palabras[ datos[ k] .posicionPalabra] ;
  int l2 = p2.size();
  bool secruzan;

  if (datos[ i] .direccion !=
      datos[ k] .direccion) {
    //pueden cruzarse
    switch (datos[ i] .direccion) {
      case 'H' :
        secruzan = ((f1 >= f2) &&
               (f1 <= f2+l2-1)
            && (c2 >= c1)
            && (c2 <= c1+l1-1));
        return !(secruzan &&
         (p1[ c2-c1] != p2[ f1-f2] ));
      case 'V' :
        secruzan = ((c1 >= c2)
            && (c1 <= c2+l2-1)
            && (f2 >= f1)
            && (f2 <= f1+l1-1));
        return !(secruzan &&
         (p1[ f2-f1] != p2[ c1-c2] ));
    }
  }
  else // las palabras no se solapan
    return true;
}
```

Y la función `nuevaOK` comprueba si no hay colisiones entre la palabra colocada en la posición `k` y las anteriores:

```
bool nuevaOK(string palabras[ ],
             Dato datos[ ], int k) {
  bool r = true;
  int i = 0;
  while( r && (i < k)) {
    r = dosOK(palabras, datos, i, k);
    i++;
  }
  return r;
}
```

Si hay varias palabras que cumplen las condiciones para un determinado hueco, habrá que probar con todas ellas, por turno. Colocaremos una palabra que cumpla las restricciones y haremos una llamada recursiva para extender la solución, rellenando el resto de huecos. Cuando dicha llamada recursiva termine habrá explorado todas las posibles extensiones, por lo que podremos colocar en el hueco actual la siguiente palabra válida, y seguir probando. Las soluciones se alcanzan cuando se han conseguido colocar *N* palabras, una en cada hueco. En ese caso, habrá una palabra no colocada. Como el resultado final del problema debe ser la lista de palabras que pueden ser omitidas en una solución válida, apuntaremos en el vector `sobra` que esta palabra no se ha colocado en una solución válida.

El siguiente procedimiento realiza la búsqueda recorriendo, recursivamente, el árbol de exploración, siguiendo el algoritmo de vuelta atrás descrito. Utiliza como parámetros el vector `palabras` con las palabras dadas, el vector `datos` con la información sobre los huecos y la asignación parcial de palabras a huecos, el número `tam` de huecos (uno menos que el número de palabras), el siguiente hueco a rellenar `k`, el vector `usada` para conocer qué palabras ya se han colocado, la variable `solucion` donde se indicará si se ha encontrado al menos una solución válida, y el vector `sobra` para indicar qué palabras han sobrado en alguna solución válida. El bucle `for` más externo recorre todas las palabras para ver cuáles se pueden colocar en el hueco `k`. El primer criterio de poda es que la palabra `p` esté ya usada. En caso contrario la palabra se coloca (temporalmente) en el hueco `k` y se aplica el resto de criterios de poda, a saber, si la palabra no cabe o entra en conflicto con las anteriores. Si estas condiciones se cumplen satisfactoriamente, puede ocurrir que hayamos encontrado una solución completa (`k == tam-1`) en cuyo caso se busca qué palabra no ha sido usada, o que tengamos que seguir colocando palabras en el resto de los huecos (a partir de `k+1`), lo que se realiza con una llamada recursiva. Cuando la llamada recursiva termine y devuelva el control, se intentará colocar la siguiente palabra en el hueco `k`, con la siguiente vuelta del bucle `for`.

```
void resolverCrucigrama(string palabras[ ],
     Dato datos[ ], int tam, int k,
     bool usada[ ], bool& solucion,
     bool sobra[ ]) {
  for(int p = 0; p <= tam; p++) {
    if (! usada[ p] ) {
      datos[ k] .posicionPalabra = p;
      usada[ p] = true;
      if (cabe(palabras, datos, k)
         && nuevaOK(palabras, datos, k)) {
        if (k == tam-1) { // solución
          solucion = true;
          for (int i = 0; i <= tam; i++) {
            sobra[ i] = sobra[ i]
                      || (! usada[ i] );
          }
        } else {
          resolverCrucigrama(palabras,
            datos, tam, k+1, usada,
            solucion, sobra);
        }
      }
      usada[ p] = false;
    }
  }
}
```

El programa principal, después de leer los datos de un crucigrama del fichero de entrada, y poner como falso todas las posiciones de los vectores `sobra` y `usada`, llama al algoritmo anterior para resolver el crucigrama de todas las formas posibles. Por último, si se han encontrado soluciones, se escriben en la pantalla todas las palabras que han sobrado en alguna solución:

```
leerCasoEstudio(palabras, datos, numslots);
for (int i = 0; i <= numslots; i++) {
  usada[ i] = false;
  sobra[ i] = false;
}

haySolucion = false;
resolverCrucigrama(palabras, datos,
```

```
  numslots, 0, usada, haySolucion, sobra);

cout << "Prueba " << numprueba << ": ";
if (haySolucion) {
  for (int i = 0; i <= numslots; i++) {
    if (sobra[ i] )
      cout << palabras[ i] << " ";
  }
  cout << endl;
} else {
  cout << "Imposible" << endl;
}
```

3. Solución en Haskell

Como los aspectos algorítmicos ya están explicados para la solución anterior, y coinciden en ambos paradigmas, nos centramos en la implementación según el paradigma funcional.

3.1. Implementación

Definimos los conceptos necesarios usualmente al resolver crucigramas: las casillas del crucigrama son pares de enteros, acotadas por el tamaño del crucigrama; las direcciones posibles son horizontal y vertical; las flechas se indican mediante su casilla de partida y su dirección; las palabras son cadenas de caracteres; un enunciado consiste en una lista de *N* flechas y otra de *N*+1 palabras:

```
type Casilla   = (Int,Int)
(ancho,alto)   = (10,10)
data Direccion = Horizontal | Vertical
     deriving (Eq,Show)
type Flecha    = (Casilla,Direccion)
type Palabra   = String
type Enunciado = ([ Flecha] ,[ Palabra] )
```

Cuando se decide situar una palabra a partir de una flecha, en una casilla,

```
type PalColocada = (Flecha,Palabra)
```

las letras de la palabra ocupan algunas casillas del crucigrama:

```
casillas :: PalColocada ->
             [ (Casilla,Char)]
casillas (((y,x), Horizontal),palabra) =
  zip [ (y ,x' ) | x' <- [ x,x+1..]] palabra
casillas (((y,x), Vertical  ),palabra) =
  zip [ (y' ,x ) | y' <- [ y,y+1..]] palabra
```

Pero no siempre será posible poner una palabra a partir de una flecha, porque puede salirse del crucigrama (lo que controla la función `cabe`)

```
cabe :: PalColocada -> Bool
cabe (((x,_),Horizontal),pal) =
  x + length pal - 1 <= ancho
cabe (((_,y),Vertical  ),pal) =
  y + length pal - 1 <= alto
```

o por tropezar con otra ya colocada (lo que controla la función `dosOK`): dos palabras serán compatibles si no hay colisión, esto es, si en las casillas comunes no hay una letra distinta:

```
dosOK :: PalColocada ->
         PalColocada -> Bool
dosOK palColocada1 palColocada2 =
  and [ xy1 /= xy2 || c1 == c2 |
   (xy1,c1) <- casillas palColocada1,
   (xy2,c2) <- casillas palColocada2]
```

En un momento dado, intentaremos situar una palabra nueva, comprobando que es compatible con una lista de palabras ya situadas y que se suponen compatibles entre sí:

```
nuevaOK :: [ PalColocada] ->
            PalColocada -> Bool
nuevaOK colocadas nueva =
  all (dosOK nueva) colocadas
```

Pero el problema no es situar una palabra, sino todas las palabras pendientes, suponiendo ya colocadas unas cuantas:

```
todasOK' :: [ PalColocada] -> Enunciado ->
            [ ([ PalColocada] ,Palabra)]
todasOK' ac ([ ]  ,[ palabra] ) =
 [ (ac,palabra)]
todasOK' ac (f:fs,palabras)  =
  let posibles = [  ((f,p), (fs,ps)) |
    p <- palabras,
    cabe nueva && nuevaOK ac (f,p),
    let ps = quitar p palabras]
  in concat [ todasOK' (fp:ac) fsps |
              (fp,fsps) <- posibles]
  where quitar x xs =
          takeWhile (x /=) xs ++
          tail (dropWhile (x/=) xs)
```

Y entonces, la resolución de una sopa de letras es un caso particular de la función anterior, partiendo de que no se tiene ninguna colocada y de que todas las palabras del enunciado están pendientes:

```
todasOK :: Enunciado ->
           [ ([ PalColocada] , Palabra)]
todasOK enunciado = todasOK' [] enunciado
```

3.2. Pero, ¿dónde está la vuelta atrás?

En esta solución, las listas definidas por comprensión nos han permitido expresar clara y limpiamente la vuelta atrás (sin completar el árbol). La función `todasOK'` busca todas las palabras que encajan en un hueco (casilla y dirección) de forma compatible con las ya colocadas. Esta compatibilidad la dilucida el predicado `nuevaOK` que, insertado como un filtro, criba el resultado de la función `todasOK'` , evitando prolongar el árbol de la búsqueda con ese proyecto de solución, y revertiendo hacia atrás un resultado vacío. Y también ocurre así cuando en un nodo todos los proyectos de solución se frustran. El caso positivo ocurre únicamente con las palabras que encajen, profundizando en la búsqueda. En resumidas cuentas, la lista de `posibles` ya se genera limpiamente, incluyendo sólo los proyectos de solución viables.