UNA EXPERIENCIA DE "PODA" EN UN ALGORITMO

En este artículo se muestra una experiencia obtenida en la cátedra de Diseño y Análisis de Algoritmos, de la Carrera de Ingeniería Civil en Computación e Informática de la Universidad Central de Chile. En ella se toca entre otros temas, ideas fundamentales para la obtención de Algoritmos Eficientes. En esta ocasión el tema es BackTracking (búsqueda en profundidad, vuelta atrás), Branch And Bound (poda de ramificaciones) y Programación Dinámica, mediante el estudio de un caso clásico de uso de BackTracking.

Keywords: BackTracking, Branch And Bound, Programación Dinámica.

Dentro del estudio del análisis de un algoritmo, comúnmente nos encontramos con una solución mediante Backtracking, se han intentado muchas formas de mejoría, pero la mayoría son sólo mejoras parciales, es en este contexto que se presentan las posibilidades de hacer Poda, también conocida como Branch and Bound. Para una explicación detallada usaremos un problema matricial a modo de ejemplo, el cual se ha usado en varias oportunidades para mostrar a los estudiantes como se pueden implementar diversas mejoras usando poda, como idea base, sobre un algoritmo de búsqueda en profundidad, con el adicional que en esta oportunidad un estudiante aplicó Programación Dinámica exitosamente a este problema.

PROBLEMA

Problema: Dada una matriz numérica cualquiera, se desea encontrar el camino de menor costo desde un vértice de esta al vértice opuesto, considerando un avance con vecindad 4. El costo de pasar de una casilla a otra esta dado por la diferencia del contenido de ambas.

Una primera solución usando búsqueda en profundidad está dada por el algoritmo mostrado en la figura 1.

Este algoritmo genera durante su búsqueda todas las posibilidades de camino que tiene un largo menor a N*N, si bien resuelve el problema, su costo es exponencial $O(n^{n2})$.

Resulta evidente que la razón de esta demora está en que se repiten muchas veces, caminos que no pueden ser mejores, por pasar reiteradas veces por casillas repetidas. Una primera mejora sería entonces evitar que se pase más de una vez por una casilla. Esto tiene diversas posibilidades:

a) Evitar que se retroceda a la casilla desde la que se viene en el paso anterior.

b) Evitar que se vuelva a alguna casilla, por la que se ha pasado en varios pasos anteriores.

Con esto se logra reducir considerablemente el largo de los caminos de búsqueda, al evitar la creación de ciclos, que sólo aumentan el costo. Para lograr esto se agrega a la búsqueda una matriz que mantenga recuerdo de las casillas por las que se ha pasado anteriormente a fin de evitar repeticiones. El algoritmo obtenido se muestra en la figura 2.

Con esta mejoría, se logra que el árbol generado en la búsqueda pase de 4 ramificaciones a un máximo de 3 y que además no pase dos veces por la misma casilla.

A esta altura el tiempo real de espera para el termino del algoritmo bajo a exponencial O(n^n).

Después de intentar varias ideas se encuentran dos que significan un real aporte a la poda.

Estas son cambiar la pregunta de (costoActual<mejorCosto) y la verificación de noMeDvuelvo (i,j) a la entrada del algoritmo. Con esto nuestro algoritmo queda de acuerdo a lo mostrado en la figura 3.

Hasta aquí eran las mejoras obtenidas por los estudiantes del curso de Diseño y Análisis de Algoritmos de la carrera de Ingeniería Civil en Computación e informática, donde se había planteado este problema a modo de tarea. Como la ciencia de los algoritmos es dinámica y las buenas ideas no dejan de brotar, el alumno Ithan Moreira, de la Escuela de Ingeniería Civil en Computación e Informática, propuso una mejora considerable, " seguir intentando sólo caminos parciales, que son mejores que otros caminos ya intentados ", esto implica la idea de un "camino optimo está formado, por una serie de caminos intermedios también óptimos". Nuestro algoritmo queda entonces tal como se observa en la figura 4.

En él se agregó una matriz llamada mejorParcial, la que usando el concepto de Programación Dinámica, guarda el mejor camino obtenido hasta esa casilla, cuando vuelve a pasar por esa casilla con un costo mayor, poda el avance y retorna a buscar otra alternativa, pero si llega a esta con un costo mejor, reemplaza ese valor por la mejor alternativa.

Adicionalmente la función todaviaEsBueno(i,j,costoActual) permite reemplazar la verificación de paso < N*N para evitar un camino muy largo, que incluya ciclos, reemplaza la verificación costoActual<mejorCosto,pues avanza probando mientras esta condición se cumple y la verificación de noMeDvuelvo(i,j), pues resulta imposible obtener un camino de menor costo repitiendo alguna casilla en el camino (figura 5).

FIGURA 1

```
public void recorre(int  i, int j  , int   costoActual,  int   paso)
{
        if ( estaDentro(i,j)&&(paso < N*N))
        {
          if ( (i == N-1) && ( j == N-1)&& (costoActual<mejorCosto )) {
              mejorCosto = costoActual ;
             System.out.println("       se mejora costo a:"+mejorCosto );
          }
          if  ((i+1)<N) recorre(i+1,j,Math.abs(M[i][j]-M[i+1][j])+costoActual,paso + 1);
          if  ((j+1)<N) recorre(i,j+1,Math.abs(M[i][j]-M[i][j+1])+costoActual,paso + 1);
          if  ((i-1)>=0)recorre(i-1,j,Math.abs(M[i][j]-M[i-1][j])+costoActual,paso + 1);
          if  ((j-1)>=0)recorre(i,j-1,Math.abs(M[i][j]-M[i][j-1])+costoActual,paso + 1);
        }
    }
```

FIGURA 2

```
public void recorre(int  i, int    j  , int    costoActual,  int  paso)
   {
     if ( estaDentro(i,j)&&(paso < N*N))
      {
          yaPasePorAqui[i][j] = 1 ;
     if ( (i == N-1) && ( j == N-1)&& (costoActual<mejorCosto )) {
           mejorCosto = costoActual ;
          guardaMejorCamino();
         System.out.println("                    se mejora costo a:"+mejorCosto );
         }
        // la function noMeDevuelvo Evita que se visiteunacasillaporsegundavez
        if  (((i+1)<N) &&noMeDvuelvo(i+1,j)) recorre(i+1,j,Math.abs(M[i][j]-M[i+1][j])+costoActual,paso +1);
        if  (((j+1)<N) &&noMeDvuelvo(i,j+1)) recorre(i,j+1,Math.abs(M[i][j]-M[i][j+1])+costoActual,paso +1);
        if  (((i-1)>=0)&&noMeDvuelvo(i-1,j)) recorre(i-1,j,Math.abs(M[i][j]-M[i-1][j])+costoActual,paso +1);
        if  (((j-1)>=0) && noMeDvuelvo(i,j-1))recorre(i,j-1,Math.abs(M[i][j]-M[i][j-1])+costoActual,paso +1);
       yaPasePorAqui[i][j] = 0 ;
      }
   }
public   boolean   noMeDvuelvo(int  i,  int   j)   {
         return   yaPasePorAqui[i][j]==0;
   }
```

FIGURA 3

```
public   void   recorre(int  i, int j  , int   costoActual,   int    paso)   {
if ( estaDentro(i,j)&&(paso < N*N)&& (costoActual<mejorCosto )&&  noMeDvuelvo(i,j))
{
      yaPasePorAqui[i][j] = 1 ;
      if ( (i == N-1) && ( j == N-1)) {
          mejorCosto = costoActual ;
         guardaMejorCamino();
         System.out.println("                    se mejora costo a:"+mejorCosto );
       }
        if  ((i+1)<N)  recorre(i+1,j,Math.abs(M[i][j]-M[i+1][j])+costoActual,paso +1);
        if  ((j+1)<N)  recorre(i,j+1,Math.abs(M[i][j]-M[i][j+1])+costoActual,paso +1);
        if  ((i-1)>=0) recorre(i-1,j,Math.abs(M[i][j]-M[i-1][j])+costoActual,paso +1);
        if  ((j-1)>=0) recorre(i,j-1,Math.abs(M[i][j]-M[i][j-1])+costoActual,paso +1);
       yaPasePorAqui[i][j] = 0 ;
     }
   }
```

FIGURA 4

```
public void recorre(int  i, int j  , int   costoActual ,   int    paso)
{
if ( estaDentro(i,j)&&(paso < N*N)&& (todaviaEsBueno(i,j,costoActual)))
     {
         yaPasePorAqui[i][j] = 1 ;
        mejorParcial[i][j] = costoActual ; // guardo el mejorcosto, porahora
       if ( (i == N-1) && ( j == N-1)) {
            mejorCosto = costoActual ;
            guardaMejorCamino();
           System.out.println("     se mejora costo a:"+mejorCosto );
       }
       if  ((i+1)<N)  recorre(i+1,j,Math.abs(M[i][j]-M[i+1][j])+costoActual,paso +1);
       if  ((j+1)<N)  recorre(i,j+1,Math.abs(M[i][j]-M[i][j+1])+costoActual,paso +1);
       if  ((i-1)>=0) recorre(i-1,j,Math.abs(M[i][j]-M[i-1][j])+costoActual,paso +1);
       if  ((j-1)>=0) recorre(i,j-1,Math.abs(M[i][j]-M[i][j-1])+costoActual,paso +1);
        yaPasePorAqui[i][j] = 0 ;
     }
   }
```

FIGURA 5

```
public    boolean   todaviaEsBueno(int  i,  int  j  ,  int   costoActual) {
              return    costoActual<mejorParcial[i][j] ;
}
```

CONCLUSIONES

Las mejoras obtenidas permitieron usar matrices más grandes, las primeras versiones no permitían en un tiempo razonable, encontrar resultados sobre matrices de 10x10, las segundas versiones permitieron llegar en 20 minutos a matrices 20x20, la versión final permite en 2 minutos obtener soluciones de matrices de 500x500, lo que se puede considerar como una mejora significativa. Pues el algoritmo que originalmente era de $O(^{4n})$ luego del uso de programación dinámica llega a ser de $O(n^2)$.

APLICABILIDAD

En un trabajo futuro es posible aplicar este concepto a la obtención de una mejor ruta entre dos ciudades considerando como costo las alturas usadas en el terreno, bajo el principio de "el mejor camino es el más plano", es decir el que tiene la menor cantidad de cuestas posibles, que en algún momento llevan a la construcción de túneles.

BIBLIOGRAFÍA

Gonzalez, I., P. Dreyse, D. Cortes-Arriagada, M. Sundararajan, C. Morgado, I. Brito, C. Roldan-Carmona, H. J. Bolink and B. Loeb (2015). "A comparative study of Ir(iii) complexes with pyrazino[2,3-f][1,10] phenanthroline and pyrazino[2,3-f][4,7] phenanthroline ligands in light-emitting electrochemical cells (LECs)." Dalton Transactions 44(33): 14771-14781.

AUTOR:

Julio Andres Fuentealba Vivallo.

Escuela de Computación de Informática
Facultad de Ingeniería
Universidad Central

E-mail: julinspi@gmail.com