Parte 1

PRÁCTICA 4

Algoritmos de Vuelta Atrás y de Ramificación y Poda

Grupo 1

Alba Moyano, Mª Mercedes
Andrades Molina, Christian
Azpeitia Muñoz, Fco.Javier
Keane Cañizares, Miguel
Muriel Sánchez lafuente, Guillermo

Introducción

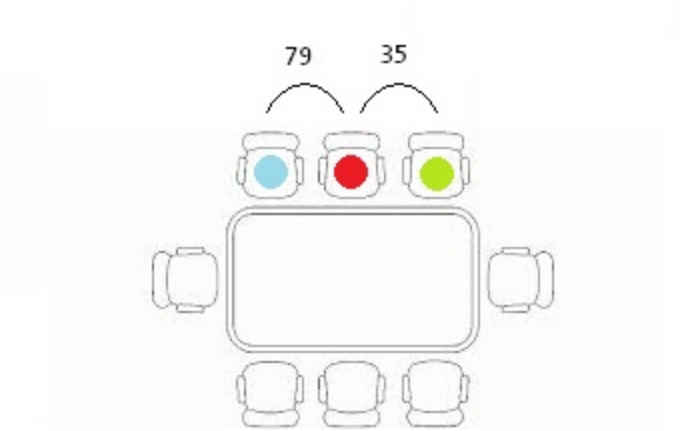
El problema consiste en encontrar la mayor “afinidad” entre comensales de una boda. El número de invitados será *n*, que se sentarán alrededor de una gran mesa rectangular teniendo sentados alrededor de una única gran mesa rectangular, de forma que cada invitado tiene a dos comensales sentados a su lado. El nivel total de afinidad es el propio de cada invitado con las dos personas que tiene a su lado.

Ver como con la técnica backtracking podemos conseguir la solución óptima y Branch and Bound sin tener que recorrer todos los caminos posibles (poda) no como en Greedy.

Análisis del Problema

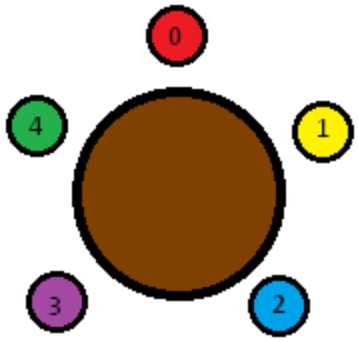
En nuestro problema tenemos *n* invitados. Cada invitado tiene una afinidad para cada uno de los demás invitados que se representa con un número entre 0 y 100. La convivencia total es la suma de las convivencias de cada invitado con los que tiene al lado. Buscamos la secuencia de invitados con la que conseguimos mayor nivel de convivencia.

Nuestra solución al problema podría ser cualquier situación de los invitados, siempre que la afinidad sea la óptima.



En esta imagen se muestra la afinidad que tiene el invitado rojo con el azul (79) y el verde (35). Con los demás invitados también tendrá *x* afinidad.

Un ejemplo de invitados sería el siguiente:



Afinidad

0-1 (20)	1-2 (60)	2-3 (90)	3-4(2).
0-2 (30)	1-3 (80)	2-4 (70)	.
0-3 (40)	1-4 (10)		.
0-4 (50)			.

Tenemos que tomar una serie de decisiones cada decisión que tomamos nos lleva a un nuevo conjunto de decisiones y alguna secuencia de decisiones es la solución del problema.

Para calcularla utilizaremos *backtracking*. La eficiencia sería de O(q(n)n!) ya que tenemos que explorar n! nodos. Con un algoritmo de poda mejoramos el tiempo de ejecución evitando tener que recorrer todos las soluciones posibles.

Desarrollo de la Solución

La solución que hemos propuesto se divide en dos partes principales:

Primera parte. Inicialización del problema.

Para representar el problema hemos propuesto representar las empatías en una matriz simétrica con la diagonal a cero.

N	**0**	**1**	**2**	**3**	**4**
0	*0*	20	30	40	50
1	20	*0*	60	80	10
2	30	60	*0*	90	70
3	40	80	90	*0*	2
4	50	10	70	2	*0*

También disponemos de un vector con las afinidades ordenadas de mayor a menor para calcular la máxima eficiencia que se puede conseguir en un determinado nodo del árbol que se recorre en el backtracking.

90	80	70	60	50	40	30	20	10	2

Pseudocódigo:

Al programa le pasamos el número de invitados que se desee. Se llama a la función matrizTriangular en la que se declara memoria y se generan aleatoriamente los valores de empatías para cada invitado.

Función: `matrizTriangular`
Entorno: `N_INVITADOS` es un parámetro de la función
`MATRIZ_EMPATIA` es una matriz de números enteros.
Algoritmo:

```
                  reserva memoria para MATRIZ_EMPATIA del tamaño de
N_INVITADOS*N_INVITADOS
      PARA i=0 HASTA N_INVITADOS CON INCREMENTO +1
            //Rellenamos matriz
            PARA j=0 HASTA N_INVITADOS CON INCREMENTO +1
                  SI i==j ENTONCES
                        MATRIZ_EMPATIA[i][j]=0;
                  SI i>j ENTONCES
                        MATRIZ_EMPATIA[i][j]=MATRIZ_EMPATIA[j][i]
                  SI NO
                        MATRIZ_EMPATIA[i][j]= Número aleatorio 0-100
            FIN_PARA
      FIN_PARA

Finfunción
```

Después de esto, tenemos creada una matriz simétrica con números aleatorios entre 0 y 100 y la diagonal a 0 y con el tamaño del número de invitados.

Creamos un vector en el que se guardan los valores de empatía ordenados de mayor a menor. El pseudocódigo sería el siguiente:

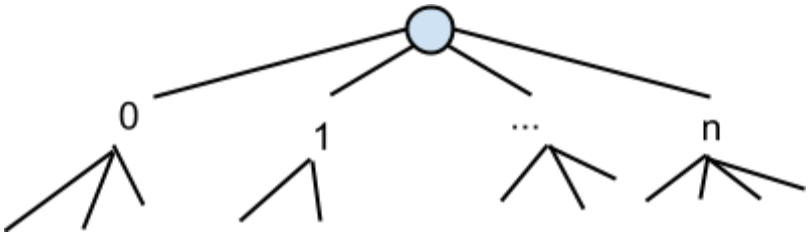
Función: `vectorR`
Entorno: `N_INVITADOS, MATRIZ_EMPATIA` y `TAM` es un parámetro de la función
`VECTOR_R` es un vector de enteros
`CONT, SIZE` son enteros
Algoritmo:

```
      reserva memoria SIZE para VECTOR_R
      //Rellenamos vector
      CONT=0
           PARA i=0 HASTA N_INVITADOS CON INCREMENTO +1
             PARA j=0 HASTA N_INVITADOS CON INCREMENTO +1
                  SI i<j ENTONCES
                        VECTOR_R[CONT]=MATRIZ_EMPATIA[i][j]
                        CONT++;
                        SI CONT==SIZE
                              se duplica la memoria del vector
                              SIZE=SIZE*2

                  FIN_PARA
      FIN_PARA
      TAM=CONT
      QUICKSORT(VECTOR_R,CONT)

      se invierte VECTOR_R
Finfunción
```

El vector resultante lo vamos a utilizar para la poda.

Segunda parte. Backtracking y poda.

Tenemos una serie de invitados. El árbol solución sería un árbol tal que así:



El recorrido desde arriba del árbol hasta cualquiera de las hojas es una solución de como colocar los invitados. Este árbol tiene tantos niveles como el número de invitados menos uno. Guardamos un valor de empatía en el que se guarda el máximo nivel de empatía que hemos conseguido hasta el momento.

Por ejemplo: Cuando recorremos la primera rama del arbol (0,1,2,3,4) se guardaría 222

0-1 (20) 1-2 (60) 2-3 (90) 3-4(2) 0-4 (50) = 222

Cuando encuentra una empatía mejor guarda el camino recorrido en un vector, que sería la solución definitiva cuando termina de recorrer el árbol entero.

El lado negativo del *backtracking* es que con valores de *n* muy grandes, nuestro programa tardaría demasiado en resolver el problema. Gracias a la recursividad, el árbol se genera sólo con la función VueltaAtras.

La poda implementada funciona de la siguiente forma. Imaginemos que estamos en un nodo con *x* de empatía temporal (la empatía que hemos conseguido hasta el momento desde la raíz del árbol hasta dicho nodo). Sabemos en el nivel del árbol en el que estamos, por los que nos quedan *n_invitados-1-nivel* niveles para llegar a una hoja. Antes de explorar en profundidad desde el nodo que estamos, calculamos la empatía máxima que podemos conseguir que sería la suma de la empatía temporal más los *n_invitados-nivel* enteros del vector ordenado de empatías. Si la empatía máxima que podemos conseguir es menor que la empatía máxima conseguida hasta el momento, se poda ese nodo del árbol.

Pseudocódigo:

Donde se consigue el vector solución. Se llama desde el `main` a `VueltaAtras`, pasándole como parámetros un vector `SOLUCION`, del cual solo lo tenemos declarado en memoria, sin inicializar, que es el vector donde, cuando halla acabado la `VueltaAtras`, guardará la solución de cómo tienen que ir sentados los invitados.

Función: `VueltaAtras`
Entorno: `SOL` es un parámetro de la función. Vector de enteros donde se almacena un primer orden de invitados.

`INVITADO` es un entero (parámetro de la función)

`E_TEMP` es un entero (parámetro de la función).

`E_MEJOR` es un entero (parámetro de la función).Guarda la mejor empatía conseguida hasta el momento.

`N_INVITADOS` es un entero (parámetro de la función).

`MATRIZ_EMPATIA` es una matriz de enteros (parámetro de la función) Guarda la matriz empatía de los invitados.

`SOLUCION` es un vector de enteros (parámetro de la función). Guarda la solución final

`VECTOR_R` es un vector de enteros (parámetro de la función). Vector de empatías ordenado.

`NUEVA_E` es un entero.

Algoritmo:

```
SI INVITADO==N_INVITADOS-1
      E_MEJOR = max(E_MEJOR, E_TEMP +
MATRIZ_EMPATIA(SOL[INVITADO]][SOL[0]]))
      SI E_TEMP+MATRIZ_EMPATIA[SOL[INVITADO]][SOL[0]]==E_MEJOR
      ENTONCES //hemos llegado a la solucion final
            PARA i=0 HASTA N_INVITADOS CON INCREMENTO +1
                  SOLUCION[i]=SOL[i]
            FIN_PARA
SI NO
      PARA i=INVITADO HASTA N_INVITADOS CON INCREMENTO +1
            Swap(SOL[INVITADO+1],SOL[i] )  //Intercambia los
                                           valores que se le pasan
            SI    PODA(E_TEMP,    E_MEJOR,    VECTOR_R,    INVITADO,
N_INVITADOS)) ENTONCES
                  hacer_nada
            SI NO
                  NUEVA_E = E_TEMP +
MATRIZ_EMPATIA[SOL[i]][SOL[i-1]]
                  E_MEJOR=max(E_MEJOR,
VueltaAtras(SOL,INVITADO+1,NUEVA_E,E_MEJOR,VECTOR_R,
N_INVITADO,MATRIZ_EMPATIA,SOLUCION));
                  Swap(SOL[INVITADO+1],SOL[i]);
      FIN_PARA
DEVUELVE E_MEJOR
Finfunción
```

Función: `Poda`
Entorno: `INVITADO` es un entero (parámetro de la función)

`E_TEMP` es un entero (parámetro de la función).

`E_MEJOR` es un entero (parámetro de la función).Guarda la mejor empatía conseguida

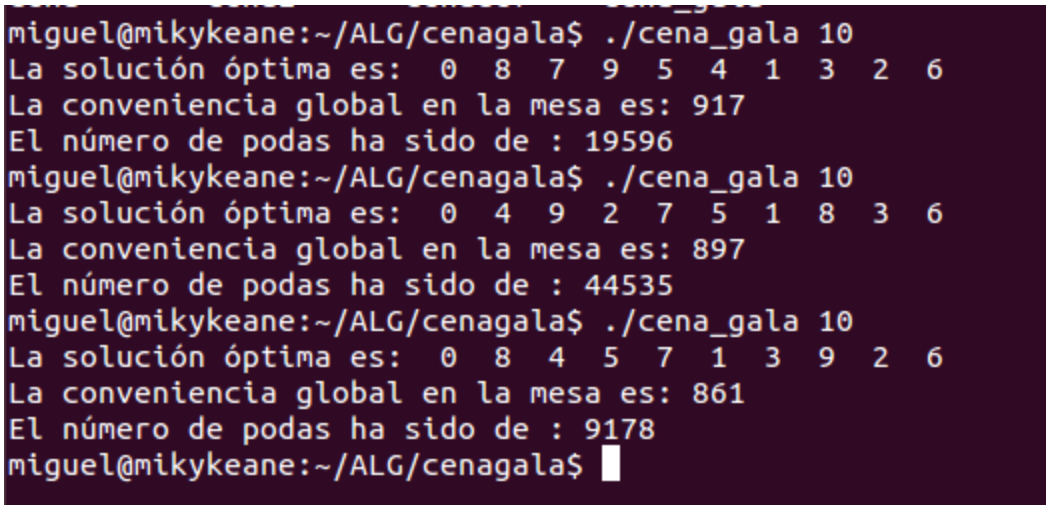
hasta el momento.
N_INVITADOS es un entero (parámetro de la función).
VECTOR_R es un vector de enteros (parámetro de la función). Vector de empatías ordenado
SUMA es un entero..

Algoritmo:

```
PARA i=0 HASTA N_INVITADOS-INVITADO INCREMENTO +1
      SUMA += VECTOR_R[i]
FIN_PARA
SI E_TEMP+SUMA<E_MEJOR
      PODAN++            //Variable global
      DEVUELVE TRUE
SI NO
      DEVUELVE FALSE
Finfunción
```

Resultados

Captura de pantalla con una ejecución del programa.



Como se puede apreciar, para un mismo número de personas (en este caso 10), con cada ejecución obtenemos tanto una conveniencia global diferente, como un número de podas diferente. Ya que al fin y al cabo el encontrar la solución óptima antes o después, será fruto del azar, ocasionando mayor o menos número de podas.

Tabla de tiempos (con poda y sin poda).

N	Sin poda	Con poda
4	1e-05	3e-06
5	4e-05	8e-06
6	0.000226	2.1e-05
7	0.001638	2.4e-05
8	0.009009	5.9e-05
9	0.048593	0.000216
10	0.43964	0.001469
11	5.22177	0.000273
12	88.5629	0.000909
13	1261.49	0.002588
14		0.01045
15		0.007614
16		0.013795
17		0.002229
18		0.037677
19		0.014088
20		0.085449

PD. Debido al enorme tiempo que requerían los tiempos del programa sin el uso de poda, sólo hemos tomado los 13 primeros para poder compararlos, ya que después los tiempos eran demasiado altos. (véase los 1261.49 segundos que tardó en ejecutar para 13 invitados). Esto sirve para matizar aún más la importancia de la poda en el algoritmo, haciéndolo indispensable para números elevados.

Conclusiones

El *backtracking* es un algoritmo el cual nos asegura que siempre se va a encontrar una solución correcta, siempre que sea un problema del tipo de que tenemos información incompleta, que un opción nos lleva a otro nuevo conjunto de opciones, cuando nos piden máximo o mínimo de algo y/o alguna secuencia de decisiones es la solución del problema.

Como hemos podido comprobar en el apartado anterior, con el algoritmo de poda correcto mejoramos el tiempo notablemente en el que el *backtracking* encuentra la solución óptima, resolviendo el problema de eficiencia que tiene este método.

El código del algoritmo diseñado es el siguiente:

```
void Swap(int &in, int &fin){
    int aux= in;
    in=fin;
    fin=aux;
}


bool poda(int &convT, int convF, const int *vo, int nodo, int &tam ){
    int suma=0;
    for(int i=0; i<tam-nodo;i++){
        suma+=vo[i];
    }
    if (convT+suma<convF){
                        podan+=1;
        return true;}
    else
        return false;
}




int VueltaAtras( int *sol, int nodo, int convT, int &convF, const int *vo, int &tam, int **m,int *ts ){
```

```
    int tem=0;


    if (nodo==tam-1){
        convF= max(convF, convT + m[sol[nodo]][sol[0]]);
                        if(convT + m[sol[nodo]][sol[0]]==convF){
                                for(int i=0;i<tam;i++){
                                        ts[i]=sol[i];
                                }
                        }
    }
    else{
        for(int i=nodo; i<tam; i++){
                                Swap(sol[nodo+1],sol[i]);
                                for(int j=0;j<tam;j++){


                                }
            if (poda(convT,convF,vo,nodo,tam)){}
            else{
                tem=convT+m[sol[i]][sol[i-1]];
                convF=max(convF, VueltaAtras(sol,nodo+1,tem,convF,vo,tam,m,ts));
            }
            Swap(sol[nodo+1],sol[i]);
        }
    }
            return convF;
}
```