Universidad de Granada

Algorítmica

Práctica 4: Backtracking

Autora:
Elena Merelo Molina

Mayo de 2018

UNIVERSIDAD
DE GRANADA

1 Problema a resolver

Se va a celebrar una cena de gala a la que asistirán n invitados. Todos se van a sentar alrededor de una única gran mesa rectangular, de forma que cada invitado tendrá sentados junto a él a otros dos comensales (uno a su izquierda y otro a su derecha). En función de las características de cada invitado (por ejemplo categoría o puesto, lugar de procedencia,...) existen unas normas de protocolo que indican el nivel de conveniencia de que dos invitados se sienten en lugares contiguos (supondremos que dicho nivel es un número entero entre 0 y 100). El nivel de conveniencia total de una asignación de invitados a su puesto en la mesa es la suma de todos los niveles de conveniencia de cada invitado con cada uno de los dos invitados sentados a su lado. Se desea sentar a los invitados de forma que el nivel de conveniencia global sea lo mayor posible. Diseñar e implementar un algoritmo vuelta atrás para resolver este problema. Realizar un estudio empírico de su eficiencia.

2 Explicación de mi solución

Para resolver el problema he creado la clase `matriz_de_conveniencia`, en la que agrupamos todos los métodos y atributos necesarios. Veámoslos uno a uno.

Atributos privados

• `vector<vector<int> > m` Es la matriz que contiene los niveles de conveniencia entre los invitados a la cena (de esta manera, por ejemplo $m[0][1]$ contiene el nivel de conveniencia del invitado 0 al 1, y así).

• `vector<int> solucion` vector con el orden en el que han de sentarse los invitados para que el nivel de conveniencia total sea máximo.

Métodos públicos

• Constructor de la matriz de conveniencia a partir de un tamaño n pasado como parámetro, rellena las componentes de la misma con un número aleatorio entre 0 y 100.

```
1  matriz_de_conveniencia(int n){
2      int i, j;
3      //Inicializamos la semilla para generar los nÃºmeros aleatorios
4      srand(time(NULL));
5
6      m.resize(n);
7
8      for(i= 0; i< n; i++)
9        m[i].resize(n);
10
11     for(i= 0; i< n ; i++)
12       for(j= 0; j< n; j++)
```

```
        m[i][j]= i != j? rand() % 100: 0; //el nÃºmero generado es un entero entre 0 y 100

  }
```

- Método que devuelve el vector solución:

```
vector<int> get_solucion(){
  return solucion;
}
```

- Método que calcula el nivel de conveniencia total para un vector dado:

```
  int conveniencia_total(vector<int> v){
    int sum= 0, n= v.size();

    for(int i= 1; i<n-1; i++)
      sum+=m[ v[i] ][ v[i+1] ] + m[ v[i] ][ v[i-1] ];

    sum+=m[ v[n-1] ][ v[n-2] ] + m[ v[n-1] ][ v[0] ];
    sum+=m[ v[0] ][ v[1] ] + m[ v[0] ][ v[n-1] ];

    return sum;
  }
```

El algoritmo es simple; para cada elemento del vector v pasado como parámetro, que consideramos como invitados, obtiene la conveniencia con el invitado de la izquierda y el de la derecha, y las suma al total. No obstante, hay que tener en especial consideración el primer y último elemento del vector, por eso los tenemos fuera del bucle: a la izquierda de $v[0]$ está $v[n-1]$ y viceversa, esto es lo que se expresa en las dos últimas líneas. Con un ejemplo queda más claro: Supongamos que tenemos cuatro invitados, n= 4. Entonces empezamos desde el invitado 1 y vemos la conveniencia con 2 y 0, son su izquierda y derecha, y una vez visto esto nos movemos a la persona 2. Sumamos la conveniencia de 2 con 1 y con 3 y nos movemos a 3, mirando la conv con 2 y con 4. De esta manera, iteramos desde 1 hasta n-1. Una vez hecho esto habremos de añadir la conveniencia de 0 con 1 y 0 con el último comensal, así como la de 4 con 3 y 4 con 0.

- Método que comprueba si un invitado ha sido ya considerado:

```
bool sentado(vector<int> v, int x){
    return count(v.begin(), v.end(), x) == 1;
  }
```

- Método que calcula el vector complementario a uno pasado como parámetro dentro de la matriz de conveniencia, esto es, si nos pasan un vector con tres invitados y en total a la cena de gala van seis, este método devolvería un vector con los invitados de la cena que no están en el vector inicial:

```
  vector<int> complementario(vector<int> v, int n){
    vector<int> result;
```

```
    for(int i= 0; i< n; i++)
      if(!sentado(v, i))
        result.push_back(i);

    return result;
```

- Algoritmo de backtracking propiamente dicho que calcula el máximo nivel de conveniencia:

```
  int max_nivel_conv(vector<int> v){
    int max_conv= 0, conv= 0;
    vector<int> por_sentar= complementario(v, m.size());

    for(int i= 0; i< por_sentar.size(); i++){
      v.push_back(por_sentar[i]);

      conv= conveniencia_total(v);

      if(conv > max_conv){
        max_conv= conv;
        solucion= v;
        max_nivel_conv(v);
      }

      v.pop_back();
    }
    return max_conv;
  }
```

Su funcionamiento no es muy complejo: calcula las permutaciones sin repetición de los n invitados, obteniendo la conveniencia de esa combinación y va comparando con la máxima de ellas, de manera que al final ésta es la que devuelve.

3 Eficiencia empírica

Para hallarla empleamos la librería `time.h`. Tenemos el siguiente `main`:

```
/*
#include "matriz_de_conveniencia.cpp"
using namespace std;

int main(int argc, char **argv){
  struct timespec cgt1, cgt2;
  double ncgt;  //para tiempo de ejecucion

  if(argc != 2){
    cout << "\nNumero de argumentos incorrecto";
```

```
    exit(-1);
  }

  int n= atoi(argv[1]), max_conv;
  matriz_de_conveniencia m(n);
  vector<int> v, f;

  clock_gettime(CLOCK_REALTIME, &cgt1);
  max_conv=  m.max_nivel_conv(v);
  clock_gettime(CLOCK_REALTIME, &cgt2);

  //cout << "\nEl maximo nivel de conveniencia es: " << max_conv;
  f= m.get_solucion();
  //cout << "\nPara ello los invitados han de estar sentados en el orden: ";
  for(int i= 0; i< f.size(); i++)
    // cout << f[i] << " ";

  ncgt= (double)(cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec - cgt1.
    tv_nsec) / (1.e+9));
  cout << "\n" << ncgt; //tiempo de ejecucion
}
```

Están comentadas las líneas que no son relativas al tiempo de ejecución para que a la hora de recoger todos los datos y dibujarlos sea más sencillo. Luego ponemos en marcha un script que ejecuta el main desde n=1 hasta 15, incrementando de 1 en 1 (ya para el tamaño 16 tardaba demasiado) y guarda los tiempos en un fichero:

```
#!/bin/csh
@ inicio = 0
@ fin = 15
@ incremento = 1
@ i = $inicio
echo > tiempos_backtracking.dat
while ( $i <= $fin )
  echo Ejecucion tam = $i
  echo `../bin/main $i` >> tiempos_backtracking.dat
  @ i += $incremento
end
```

Comparando con diferentes funciones, vemos cómo es la obtenida, que no se puede aproximar bien por ninguna de ellas y da un salto muy grande del valor 14 al 15, al ser de orden O(n!):

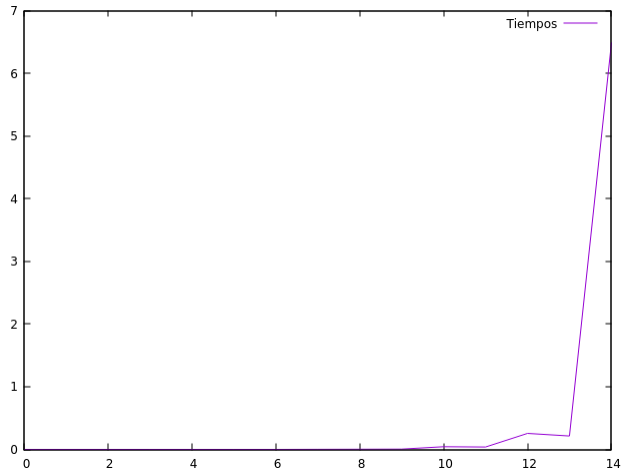

Figure 1: Eficiencia empírica del algoritmo

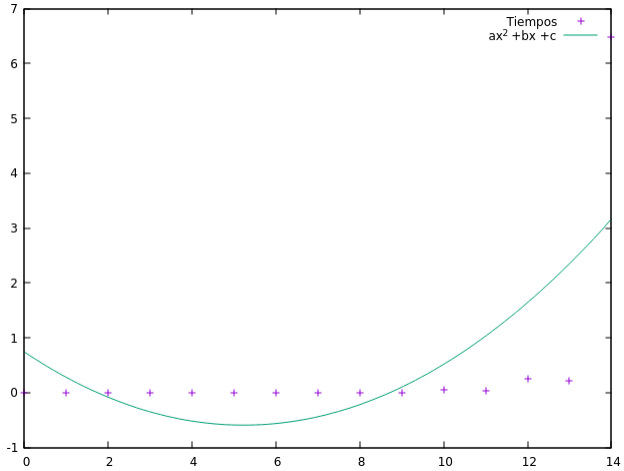

Figure 2: Gráfica del algoritmo comparada ajustada a una función cuadrática

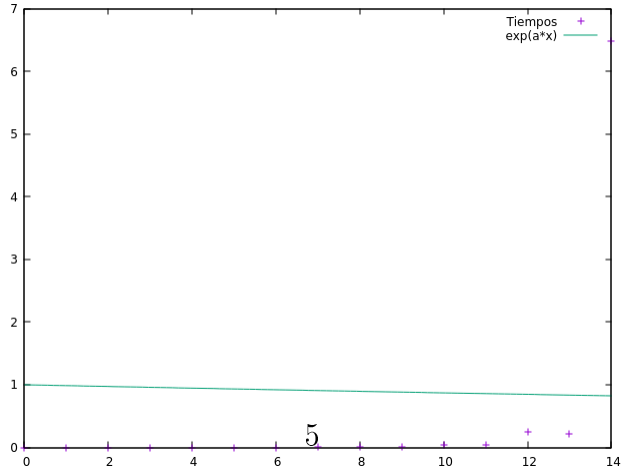

Figure 3: Algoritmo comparado con una exponencial