Universidad de Granada

Algorítmica

Memoria de Prácticas

Práctica IV: Backtracking

Autor:
Antonio Gámiz Delgado

May 8, 2018

Universidad de Granada

1 Problema

Se va a celebrar una cene 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 característiscas de cada invitado (por ejemplo po categoría, puesto, lugar de procedencia...) existen unas normas de protocolo que indican el nivel de conveniencia de que dos invitados se siente 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 Representación de la conveniencia

Para implementar la conveniencia, he creado una clase independiente llamada *ConvenienceMatrix* para obtener mayor generalidad a la hora de representarla (en caso de que queramos modificar como se obtiene la conveniencia solo tenemos que modificar esa clase y no el algoritmo entero).

Además también he aprovechado para añadirle a esa clase un método que calcule el costo de una solución dada algunos métodos útiles para el algoritmo, como:

3 Diseño del algoritmo backtracking

Para resolver el problema, primero empecé a desorrar la implementación de un árbol para luego rellenarlo, podarlo, etc, pero luego me di cuenta que con una función recursiva bastante simple obteníamos el mismo resultado saltándonos la complejidad que acarrea un árbol general.

Por lo que, a continuación expongo el algoritmo en sí y algunas variables globales que he usado (no es buena práctica usar variables globales, pero debido a que simplifican el diseño y aportar claridad he visto conveniente usarlas en esta código).

```
1  void backtracking(ConvenienceMatrix & c, vector<int> v)
2  {
3    vector<int> available=supplementary(v, c.size()); //definida en 'auxliar.cpp'
4    for(int i=0; i<available.size(); i++)
5    {
6    v.push_back(available[i]); //aniadimos el siguiente numero que no este ya en la solucion
7    if( v.size() == c.size() ) ++count;//to_s(v); //descomentar para ver todas las
       posibilidades
8    aux_cost=c.costs(v);
9    if(aux_cost > cost) //guardamos la solucion con el mayor coste
10   {
11     cost=aux_cost;
```

```
12      solution=v;
13      backtracking(c, v);
14    }
15
16    if(v.size()<7) backtracking(c, v); //recursividad
17
18    v.pop_back(); //eliminamos el elemento aniadido antes para calcular
19            //la siguiente posibilidad
20    }
21  }
```

Listing 1: Algoritmo backtracking

Con esta función recursiva lo que hacemos es calcular todas las permutaciones sin repeteción de los $n-1$ números (los asistentes a la cena de gala los númeramos desde 0 hasta $n-1$). Para calcular todas esas posibilidades, he usado la función $vector < int > supplementary(vector < int > \&v)$:

```
1  //supplementary of v respect to [0...n-1]
2  vector<int> supplementary(vector<int> & v, int n)
3  {
4    vector<int> result;
5    for(int i=0; i<n; i++) if( !is_in(v, i) ) result.push_back(i);
6    return result;
7  }
```

Listing 2: función auxiliar

Que como su nombre indica, calcula el complementario un vector de enteros dado con respecto al vector que contiene los enteros desde 0 hasta $n-1$.

El resultado de *supplementary* son los enteros que faltan por contemplar en la solución, por lo que una vez calculado, llamamos otra vez a la función, con el mismo vector v, pero en cada iteración le añadimos un elemento de los que le falta, obteniendo así todas las posibilidades.

Además he declarado las siguientes variables globales para calcular el máximo:

```
1  static int count = 0; //numero de soluciones calculadas
2  vector<int> solution; //mejor solucuion
3  int cost; //coste de la solucion
4  int aux_cost; //coste auxiliar para calcular el maximo
```

Listing 3: Variables globales

Para ahorrarme calcular todas las posibilidades, calculo todas las posibilidades hasta que tengan longitud 7, después de eso, solo desarrollo las que tengan costo más grande, que son las que buscamos.

4 Eficiencia empírica

Para calcular la eficiencia empírica he hecho un script en python (llamado *script.py*) para ejecutar el programa con un tamaño n desde 1 hasta 11.

Además de la función descrita anteriormente, también he probado empíricamente otras dos variaciones de la misma:

En esta variación de la función desarrallomas todas las soluciones "*peores*" que la que tenemos actualmente. Parece un poco raro, pero al final llega a una solución óptima descartando algunos casos (menos de los deseados, se verá después en la representación).

```
void backtracking(ConvenienceMatrix & c, vector<int> v)
{
  vector<int> available=supplementary(v, c.size()); //definida en 'auxliar.cpp'
  for(int i=0; i<available.size(); i++)
  {
  v.push_back(available[i] ); //aniadimos el siguiente numero que no este ya en la
    solucion
  if( v.size() == c.size() ) ++count;//to_s(v); //descomentar para ver todas las
    posibilidades
  aux_cost=c.costs(v);
  if(aux_cost > cost  ) //guardamos la solucion con el mayor coste
  {
    cost=aux_cost;
    solution=v;
  }

  else backtracking(c, v); //recursividad

  v.pop_back(); //eliminamos el elemento aniadido antes para calcular
  //la siguiente posibilidad
  }
}
```

Listing 4: Variación 1

Esta variación hace que el algoritmo no sea *backtracking*, es decir, esta versión contempla las $n!$ posibilidades y escoge la mejor. La he añadido para ver la mejora respecto a usar backtracking o no.

```
void backtracking(ConvenienceMatrix & c, vector<int> v)
{
  vector<int> available=supplementary(v, c.size()); //definida en 'auxliar.cpp'
  for(int i=0; i<available.size(); i++)
  {
  v.push_back(available[i] ); //aniadimos el siguiente numero que no este ya en la
    solucion
  if( v.size() == c.size() ) ++count;//to_s(v); //descomentar para ver todas las
    posibilidades
  aux_cost=c.costs(v);
  if(aux_cost > cost  ) //guardamos la solucion con el mayor coste
  {
    cost=aux_cost;
    solution=v;
  }

  backtracking(c, v); //recursividad
```

```
16
17   v.pop_back(); //eliminamos el elemento aniadido antes para calcular
18   //la siguiente posibilidad
19   }
20 }
```

Listing 5: Variación 2

A continuación veremos las representaciones gráficas por separado de cada una de las variaciones del algoritmo, y luego las veremos conjuntamente para ver la diferencia.

En las gráficas podemos ver que los puntos, hacia el tamaño $n = 11$, dan un salta muy brusco, esto se debe a que la eficiencia del algoritmo a lo bruto es de orden $O(n!)$.

Además, vemos que la diferencia entre el algoritmo a lo bruto (3), y la variación 2 (2), es muy pequeña (aunque para n lo suficientemente grande la diferencia es considerable). Esto se debe a que la condición que tiene cumplir el algoritmo (2) para desarrollar más soluciones, se cumple para muchos casos.

En cambio, vemos que el tiempo del algoritmo (1) es mucho menor. Esto se debe a que solo desarrollamos los primeros 7 niveles iniciales de las soluciones, después de eso, aplicamos backtracking y solo desarrollamos las soluciones con mayor costo. Aunque ganamos mucho en eficiencia, la solución no es la mejor (como pasa siempre en (2) y bastante frecuentemente con (3).

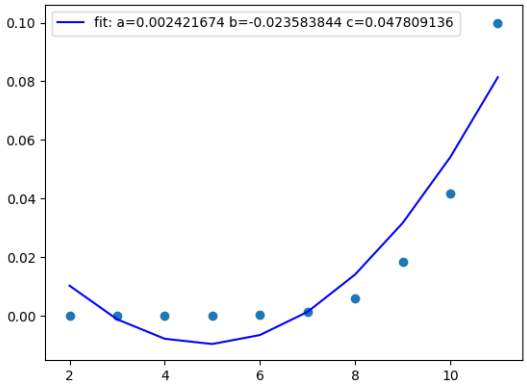

Figure 1: Eficiencia Empírica Método 1

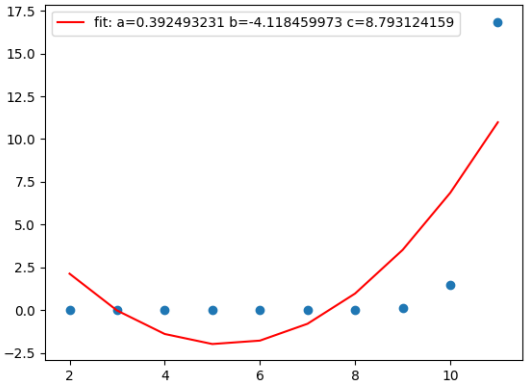

Figure 2: Eficiencia Empírica Método 2

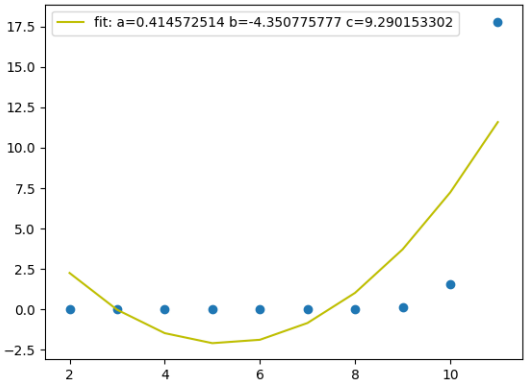

Figure 3: Eficiencia Empírica Método 3

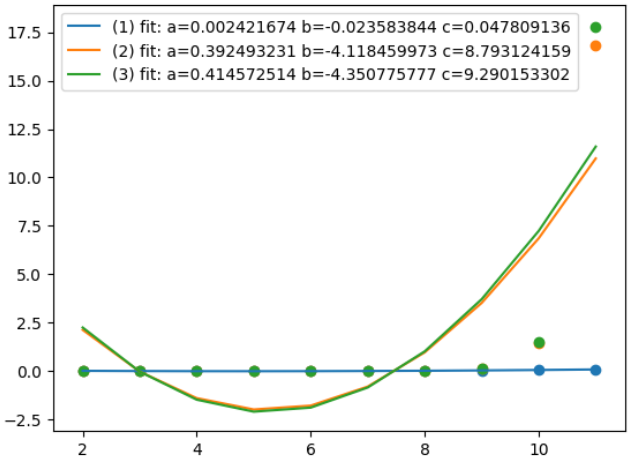

Figure 4: Eficiencia Empírica General