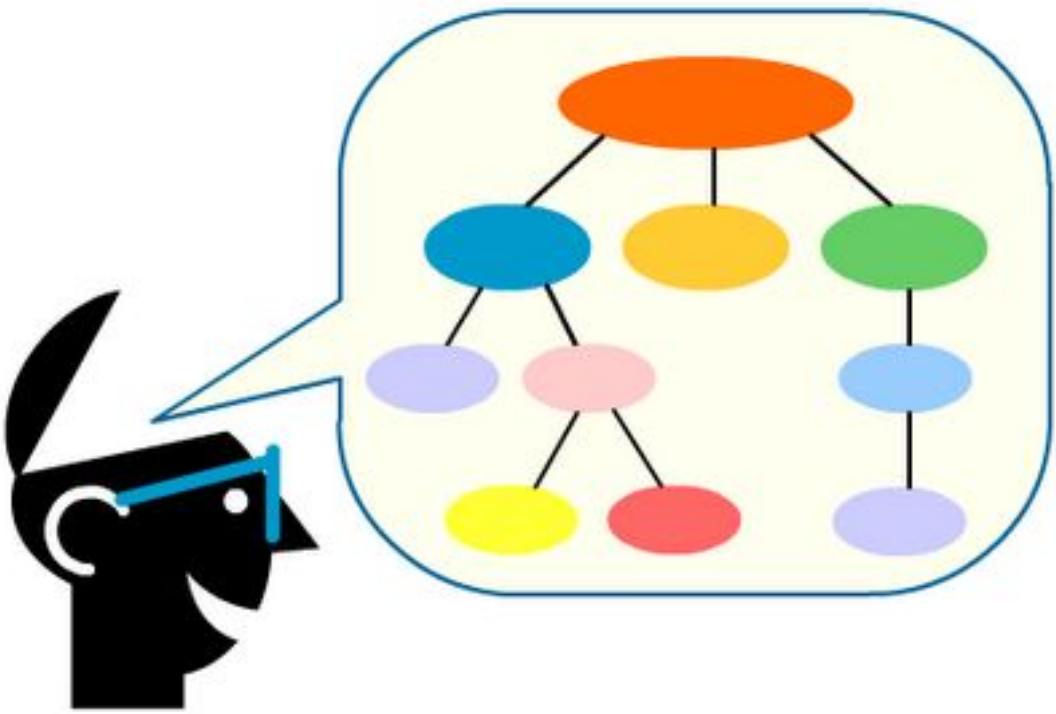
Práctica 4 Algorítmica
Algoritmos BackTracking

Realizado por:
Sergio Cervilla Ortega
Daniel Díaz Pareja
Marina Hurtado Rosales
Adrián Morente Gabaldón
Joaquín Rodríguez Claverías



Índice

1. Descripción del problema

El caso que nos ocupa en esta práctica trata del problema de “la cena de gala”, donde grandes eminencias de distintos países del mundo se sentarán a cenar en torno a una mesa rectangular, de forma que cada uno de los invitados tendrá junto a él a otros 2 comensales (uno a su izquierda y otro a su derecha). En función de las características de cada invitado, existen unas normas de protocolo que indican el nivel de conveniencia de que dos invitados se sienten en lugares contiguos (este valor vendrá dado por un número entre 0 y 100). El nivel de conveniencia total de una asignación en la mesa es la suma de su compatibilidad con el comensal de su izquierda con la del de su derecha. Se desea sentar a los invitados de forma que el nivel de conveniencia global sea máximo (no vamos a sentar a *Obama* con *Putin*, o a *Kim Jong-un* con *Park Geun-Hye*).

Más formalmente, expresaremos la búsqueda de la resolución del problema en forma de árbol; y seguiremos dos distintos modelos de trabajo, en los que inicialmente presentaremos una planificación por un algoritmo de “Fuerza bruta”, en el que se compararán todas las posibles soluciones; y a continuación, veremos la solución mediante planificación “Backtracking”, y demostraremos de forma empírica que éste último supera en eficiencia y decrece en tiempos de cómputo respecto al primero.

2. Algoritmo de “Fuerza bruta”

Este algoritmo genera todas las posibles permutaciones de la mesa de invitados, y va comprobando cuál de ellas es la mejor, con vistas a maximizar la conveniencia global. Para ello, primero creamos un vector ordenado de 0 a n, calculamos la valoración de la mesa actual (la cual podría ser la óptima) y vamos realizando permutaciones sobre la mesa con la función “next_permutation” que reordena los elementos en el rango. En cada permutación calculamos la conveniencia global, de forma que almacenamos (y actualizamos) siempre la mejor solución, que será la que se devuelva al final del programa.

El código implementado para este fin sería el siguiente:

```
// Calcula la solución al problema mediante fuerza bruta
// En el parametro vector<int> mejorMesa se guardara la mejor mesa calculada,
// y la función devuelve el valor a maximizar.
int FuerzaBruta(const vector<vector<int>> & matrizConveniencias, vector<int> & mejorMesa) {

    // Creamos una mesa ordenada, desde 0 hasta n.
    vector<int> mesa;
    for (int i = 0; i < matrizConveniencias.size(); ++i)
        mesa.push_back(i);

    // Calculamos la valoración de la mesa inicial
    int conv = calcularConvenienciaDeMesa(matrizConveniencias, mesa);
    mejorMesa = mesa;

    // Hacemos sucesivas permutaciones de la mesa con la funcion next_permutation
    // y comprobamos si el valor de esta nueva mesa es mejor que el de la anterior
    while (next_permutation(mesa.begin(),mesa.end())) {
        int nuevaConv = calcularConvenienciaDeMesa(matrizConveniencias, mesa);
        if (nuevaConv > conv) {
            mejorMesa = mesa;
            conv = nuevaConv;
        }
    }

    return conv;
}
```

Cabe decir que este método de resolución del problema dista de la solución óptima, ya que el tiempo de cómputo se dispara conforme se incrementa levemente el número de comensales (por ejemplo: para 6 sujetos saldrían 720 combinaciones, para 10, 3628800). Vista la no-optimalidad de este método, debemos explorar otras opciones.

3. Algoritmo “Backtracking”.

Tenemos todas las posibles soluciones almacenadas en un conjunto en forma de árbol, de modo que el algoritmo BT va analizando cada nodo comprobando mediante una función de factibilidad si dicho nodo nos podría llevar a una solución mejor que la que tenemos; y si no es así, “podamos” la rama que desciende al expandir dicho nodo evaluado.

La función de factibilidad que estamos usando calcula la cota local multiplicando el número de comensales que nos quedan por sentar por 200.

Tomamos esta constante porque la conveniencia máxima que podría tener cada comensal con cualquier otro es 100, y como hay que tener en cuenta la conveniencia del comensal sentado a la derecha y el que está sentado a la izquierda, en un hipotético caso de obtener el máximo, sería 200, por lo que despreciamos cualquier solución menos productiva.

Cuando llegamos a un nodo hoja, teniendo en cuenta todo el rato que hemos ido podando según la función de factibilidad, la solución correspondiente pasa a ser una solución provisional, y así, cuando hayamos explorado el árbol entero tendremos la solución óptima.

3.1 TDA solución.

Este es un tipo de dato abstracto que hemos creado para poder gestionar siempre la mejor solución posible en cada caso. Los elementos internos de este tipo de dato son:

```
//********************************* CLASE Solucion ************************//
class Solucion {
    vector<int> mesaPosible; // Va almacenando posibles soluciones
    vector<int> mesaSol; // Almacena la solución final
    int mejorValorConveniencia; // Almacena el mejor valor para la conveniencia
    int numComensales; // Numero de comensales totales
    const vector< vector<int> > * matrizConv; // Apunta a la matriz de conveniencias creada
    static const int UMBRAL = 200;

public:
    // Constructor por defecto.
    Solucion(const vector<vector<int>> * matrizConveniencias) {
        numComensales = matrizConveniencias->size();
        this->matrizConv = matrizConveniencias;
        mejorValorConveniencia = -1;
    }
```

Donde:

- *mesaPosible* es la solución provisional, y sobre la que se trabajará.
- *mesaSol* es la solución final que se ha obtenido.
- *mejorValorConveniencia* es el valor que almacenará la mejor conveniencia global en cada momento.
- **matrizConv* es un puntero a la matriz de conveniencias que se generará.

Para generar el siguiente comensal que se podría sentar (*SigValComp*), aumentamos el valor de la mesa que estamos analizando y le añadimos el primer comensal que no esté sentado y que no hayamos analizado todavía.

```
//Genera el siguiente valor válido del dominio. Se usa la función count(inicio,fin,k), que
//devuelve el número de coincidencias del valor k buscadas entre el inicio y el fin del vector
//desde donde se llama.

void SigValComp(int k) {
    ++mesaPosible[k];
    while(count(mesaPosible.begin(),mesaPosible.end(), mesaPosible[k]) > 1)
            ++mesaPosible[k];

}

// Testea si quedan valores de la solución por generar.
bool TodosGenerados(int k) const{
    return (mesaPosible[k] == numComensales);
}
```

Una vez que tenemos este comensal sentado, para comprobar si podría dar lugar a una solución mejor que la que tenemos, usamos la siguiente función de factibilidad:

```
// Comprueba si la solución es factible (función de poda). Como restricciones
// ponemos que la conveniencia actual no puede superar el valor de un cierto umbral.
bool Factible(){

    int convenienciaActual = calcularConvenienciaDeMesa(*matrizConv, mesaPosible);
    int comensalesRestantes = numComensales - mesaPosible.size();

    int convMax = convenienciaActual + UMBRAL * comensalesRestantes;

    return (convMax > mejorValorConveniencia);
}

// Representa el proceso que se alcanza cuando se alcanza una solución.
// Como estamos en un problema de optimización, comparamos con la mejor solución
// alcanzada hasta el momento y actualizamos.
void ProcesaSolucion() {
    int convenienciaActual = calcularConvenienciaDeMesa(*matrizConv, mesaPosible);
    if (convenienciaActual > mejorValorConveniencia) {
        mejorValorConveniencia = convenienciaActual;
        mesaSol = mesaPosible;
    }
}
```

La cota global que tenemos en este caso, es la conveniencia de la solución que tenemos guardada en ese momento (mejorValorConveniencia).

Para calcular la cota local, tenemos una constante llamada “UMBRAL”, cuyo valor es 200, número que hemos determinado por las razones expuestas anteriormente. La cota local de la rama que queremos evaluar y/o podar se calcula multiplicando esta constante por el número de comensales que nos quedan por sentar (comensalesRestantes).

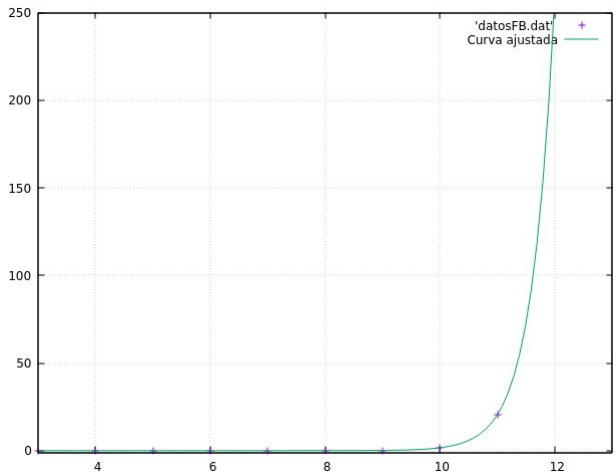
4. Análisis de eficiencia

Para la representación del análisis de eficiencia de estos algoritmos hemos usado la herramienta *gnuplot*. A partir de una secuencia de ejecuciones para distintos números de comensales hemos obtenido los siguientes datos:

APUNTE: como ya sabemos, el algoritmo por "fuerza bruta" es notablemente más lento que el de "vuelta atrás", ya que su tiempo de cómputo crece con mayor rapidez; por lo tanto hemos realizado 4 ejecuciones menos para el primero, ya que el tiempo de cómputo en nuestras máquinas llegaba a ser insostenible con cada aumento.

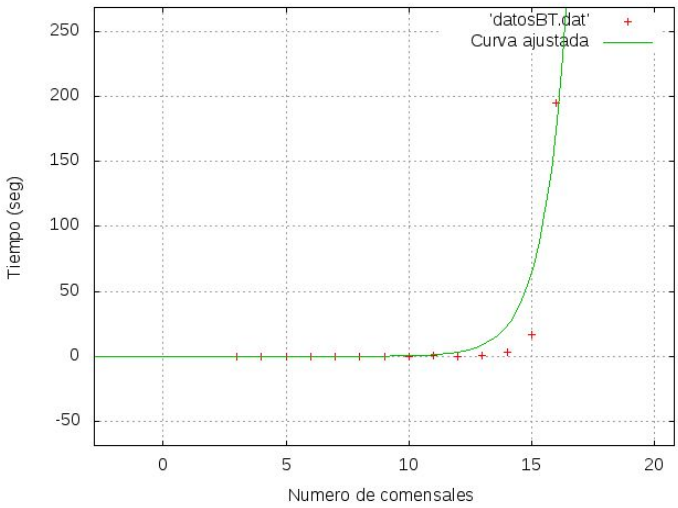
N.º comensales	Fuerza Bruta	Backtracking
3	8,46E-06	6,25E-06
4	1,30E-05	3,58E-05
5	5,16E-05	2,62E-04
6	0,000236364	0,000446034
7	0,00160993	0,001852
8	0,0165921	0,00406353
9	0,173868	0,0176419
10	1,75203	0,136734
11	20,6683	0,285196
12	261,372	0,162492
13		0,800234
14		3,16675
15		16,8327
16		195,526

Representando de forma gráfica el ajuste lineal con los datos originales del algoritmo de "fuerza bruta" hemos obtenido lo siguiente:



Como vemos, el ajuste con la función exponencial (2^n) es prácticamente perfecto; por lo que se confirma la teoría de que el crecimiento de cómputo de dicho algoritmo se comporta de forma exponencial. La diferencia más relevante se nota a partir de los 10 comensales.

Podemos deducir que el algoritmo “fuerza bruta” no es una buena opción para trabajar con valores grandes, así que debemos explorar la otra opción: “backtracking”:



El crecimiento del tiempo de cómputo de este algoritmo también crece de forma exponencial (2^n), al igual que el de “fuerza bruta”, salvo que en este caso el crecimiento más notable se realiza a partir de los 12 o 13 comensales.

Podemos constatar que el algoritmo “vuelta atrás” supone ventajas a la hora de encontrar la mejor solución a este problema, a pesar de tener también un crecimiento exponencial, ya que dicho crecimiento sucede más tarde (es decir, con mayor número de elementos). Cabe decir que conforme crece el número de comensales, el número de combinaciones posibles se dispara, viniendo dado este número por la función factorial de N (comensales).

Para terminar, realizamos una comparativa conjunta de las dos gráficas anteriores, en las que se confirma el análisis que acabamos de realizar en el párrafo anterior: