Práctica 4 : Algoritmos de Vuelta Atrás(Backtracking) -Parte I-

Alejandro Campoy Nieves

David Criado Ramón

Nour Eddine El Alaoui

Luis Gallego Quero

Índice :

1. Introducción

1.1 Algoritmos de Vuelta Atrás (Backtracking)

Metodología que se puede utilizar para buscar varias secuencias de decisiones hasta encontrar la que sea "correcta". Para ello mira todas las posibles soluciones y se queda con la mejor. También hace uso de un espacio de búsqueda, el cual consiste en un árbol en el que vamos a representar todas las posibles soluciones y los caminos para llegar a ellas.

1.2 Presentación del problema

El problema en cuestión es el de cena de gala. Este consiste en "n" invitados que se sientan todos en una mesa, y hay que colocarlos en función de las características de cada invitado, como diferentes normas de protocolo que marcan el nivel de conveniencia con el invitado que se sienta a su izquierda y su derecha. En definitiva, se desea sentar a los invitados de forma que el nivel de conveniencia global sea lo mayor posible.

Para ello vamos a utilizar el algoritmo de backtracking con un espacio de búsqueda basado en un árbol permutacional.

2. Código

Inicialmente hemos creado una serie de alias con las que identificar diferentes conceptos importantes del problema. Destacar **Mesa** en la cual almacenamos todos los invitados y **MatrizConveniencia** en la que almacenar los valores de conveniencias generados de cada invitado respecto a los demás. **ConvenienciaGlobal** es lo que buscamos maximizar.

```
using ConvenienciaGlobal = long int;
using Conveniencia = int;
using Persona = int;
using Mesa = vector<Persona>;
using MatrizConveniencia = vector<vector<Conveniencia>>;
```

Para crear los valores de conveniencia anteriormente generados hemos utilizado la siguiente función. Estos valores se generan aleatoriamente en un rango de 0-100 y la conveniencia de uno mismo está iniciada a -1.

```
MatrizConveniencia generarConveniencia(int N) {
    MatrizConveniencia salida(N,Mesa(N, -1));

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0,100);

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            if (i != j)
            salida[i][j] = dis(gen);

    return salida;
}
```

La siguiente función también es global al ejercicio, tanto para fuerza bruta como utilizando backtracking. En ella se calcula la **conveniencia global** que genera la mesa. La conveniencia de una persona en la mesa viene definida por la conveniencia de una persona con la que está a su izquierda y esa persona con la que está en la derecha.

```
ConvenienciaGlobal calcularConvenienciaGlobal(const Mesa& mesa,
                                              const MatrizConveniencia& v) {
    // Casos especiales

    if (mesa.size() == 0)
        return 0; // No hay nadie en la mesa
    else if (mesa.size() == 1)
        return 200; // Sólo hay uno en la mesa
    else if (mesa.size() == 2)
        return 2*v[mesa[0]][mesa[1]] + 2*v[mesa[1]][mesa[0]]; // Hay 2 en la mesa


    ConvenienciaGlobal salida = 0;
    // Como en el vector el 0 y el N - 1 no estan contiguos entre sí los
    // calculamos primero
    salida += v[mesa[0]][mesa[mesa.size()-1]] + v[mesa[0]][mesa[1]];
    salida += v[mesa[mesa.size()-1]][mesa[mesa.size()-2]]
           +  v[mesa[mesa.size()-1]][mesa[0]];

    // Calculamos el resto de la conveniencia
    for (unsigned i = 1; i < mesa.size() - 1; ++i)
        salida += v[mesa[i]][mesa[i-1]] + v[mesa[i]][mesa[i+1]];

    return salida;
}
```

Como mejor podemos comprobar el beneficio que se consigue con los algoritmos a estudiar, es comparando con otros algoritmos que no siguen ningún orden en concreto, por ello el siguiente. En este caso el de **fuerzaBruta** se centra en comprobar todas las posibles permutaciones viendo si la nueva es mejor que la ya guardada como mejor.

```
ConvenienciaGlobal fuerzaBruta(const MatrizConveniencia& v,
                               Mesa& mejorMesa) {
    // Creamos una mesa ordenada numéricamente
    Mesa mesa;
    for (unsigned i = 0; i < v.size(); ++i)
        mesa.push_back(i);

    // Calculamos la valoración de la mesa inicial
    auto valoracion = calcularConvenienciaGlobal(mesa,v);
    mejorMesa = mesa;

    // Realizamos todas las permutaciones y comprobamos si la nueva permutación
    // es mejor que el guardado como mejor
    while (next_permutation(mesa.begin(),mesa.end())) {
        auto valoracionLocal = calcularConvenienciaGlobal(mesa,v);
        if (valoracionLocal > valoracion) {
            mejorMesa = mesa;
            valoracion = valoracionLocal;
        }
    }

    return valoracion;
}
```

Para la resolución del problema con el algoritmo de backtracking hemos creado una clase con la que trabajar mejor la solución. Inicialmente hemos declarado cada una de las variables antes creadas como globales y también hemos recogido el número de comensales que se sentarán en la mesa.

Posteriormente podemos encontrar el constructor y una serie de métodos con los que tratar la clase.

Para las siguientes explicaciones N = size() = Nº Comensales Totales

- **Iniciar** añade al comensal nulo a la mesa.
- **siguienteValor(índice)** sienta al siguiente comensal válido a la mesa en la posición índice. Entendemos como comensal válido aquel que está dentro del rango [0, N] y está sentado únicamente una vez en la mesa.

- **todosGenerados(índice)** nos indica si ya se han generado todas las posibilidades, puesto que recorremos las posibilidades en orden ascendente se habrán generado todos si el comensal sentado en la posición índice es el comensal N ya que sólo existen los comensales [0, N-1]

```
class Solucion {
    // Apunta a la matriz de conveniencias creada
    const MatrizConveniencia* conveniencias;
    Mesa mesaActual;
    Mesa mejorMesa;
    ConvenienciaGlobal mejor;

    int nComensalesTotales;

public:
    Solucion(const MatrizConveniencia* conveniencias) {
        nComensalesTotales = conveniencias->size();
        this->conveniencias = conveniencias;
        mejor = -1;
    }

    int size() const {
        return nComensalesTotales;
    }
    void iniciar() {

        mesaActual.push_back(-1);
    }

    void siguienteValor(int indice) {
        ++mesaActual[indice];
        while(count(mesaActual.begin(),mesaActual.end(), mesaActual[indice]) > 1)
            ++mesaActual[indice];
    }

    bool todosGenerados(int indice) {
        return mesaActual[indice] == nComensalesTotales;
    }
```

- **esFactible** nos indica si la solución puede ser mejorada o no. Para ello puesto que cada comensal en el caso óptimo puede añadir 200 de conveniencia y el tamaño del vector nos indica el número de comensales sentados a la mesa, obtenemos las personas restantes y si a la situación actual le aplicamos el caso óptimo y nos quedamos por debajo o igual que en la mejor situación no será factible, en caso contrario sí.
- **actualizarMejor** comprueba si la nueva solución obtenido es mejor que la almacenada en mejor comparando sus conveniencias.

- **eliminarUltimo** quita al último comensal de la mesa. Corresponde con subir un nivel en el árbol permutacional.
- **mesa** devuelve la mejor solución obtenida.

```
    bool esFactible() {
        auto conveniencia Actual = calcularConvenienciaGlobal(mesaActual, *conveniencias);
        int personasRestantes = nComensalesTotales - mesaActual.size();

        ConvenienciaGlobal conveniencia Maxima = conveniencia Actual + 200 * personasRestantes;

        return conveniencia Maxima > mejor;
    }

    void actualizarMejor() {
        auto conveniencia Actual = calcularConvenienciaGlobal(mesaActual, *conveniencias);
        if (conveniencia Actual > mejor) {
            mejor = conveniencia Actual;
            mejorMesa = mesaActual;
        }
    }

    void eliminarUltimo() {
        mesaActual.erase(mesaActual.end()-1);
    }

    Mesa mesa() const {
        return mejorMesa;
    }
};
```

Finalmente tenemos la función que sigue el esquema general del algoritmo backtracking (**vueltaAtrasRecursiva**), en ella hacemos las declaraciones iniciales y mientras no han sido generados todos los comensales vamos comprobando si los nuevos añadidos pueden ser factibles o no. En el caso de serlo avanzamos un nivel en el árbol. Si es una solución posible, comprobamos si es mejor que la que ya tenemos guardada. Continuamos el proceso hasta tener generados a todos los comensales.

También tenemos la función **vueltaAtras** que tan solo se encarga de generar la solución llamando a la función anterior y de generar la conveniencia global final.

```
void vueltaAtrasRecursiva(Solucion& sol, int i) {
    sol.iniciar();
    sol.siguienteValor(i);
    while (!sol.todosGenerados(i)) {
        if (sol.esFactible())
            vueltaAtrasRecursiva(sol, i+1);
        if (i == sol.size()-1)
            sol.actualizarMejor();
        sol.siguienteValor(i);
    }



    sol.eliminarUltimo();
}

ConvenienciaGlobal vueltaAtras(const MatrizConveniencia& v,
                               Mesa& mejorMesa) {
    Solucion miSolucion(&v);
    vueltaAtrasRecursiva(miSolucion,0);

    mejorMesa = miSolucion.mesa();
    auto VA = calcularConvenienciaGlobal(mejorMesa, v);
    return VA;
}
```

Para concluir la parte del código, el main con el que hacer funcionar los dos algoritmos. No es más que pedir el número de comensales, generar las conveniencias aleatorias y la mesa tanto para fuerza bruta como backtracking. El resto es código para calcular los tiempos con los que posteriormente generar los gráficos comparativos.

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "Introducir como argumento numero de comensales" << endl;
        return -1;
    }

    int N = atoi(argv[1]);
    auto conveniencias = generarConveniencia(N);

    Mesa mesaFB;
    Mesa mesaVA;
    high_resolution_clock::time_point tAntes, tDespues;

#if (MOSTRARMATRIZCONVENIENCIAS && !ONLYPRINTTIME)
    cout << "Matriz de conveniencias" << endl;
    for (auto fila : conveniencias) {
        for (auto d: fila)
            cout << d << '\t';
        cout << endl;
    }
#endif
```

```
#if !ONLYPRINTTIME
    tAntes = high_resolution_clock::now();
    auto FB = fuerzaBruta(conveniencias, mesaFB);
    tDespues = high_resolution_clock::now();
    duration<double> tiempoFB = duration_cast<duration<double>>(tDespues - tAntes);


    cout << endl << "FUERZA BRUTA" << endl << endl;
    for (auto persona : mesaFB)
        cout << persona << endl;

    cout << endl << "La conveniencia global de la mesa es de: " << FB << endl;
    cout << "El algoritmo de fuerza bruta ha tardado: " << tiempoFB.count() << endl;

    tAntes = high_resolution_clock::now();
    auto VA = vueltaAtras(conveniencias, mesaVA);
    tDespues = high_resolution_clock::now();
    duration<double> tiempoVA = duration_cast<duration<double>>(tDespues - tAntes);
    cout << endl << "VUELTA ATRAS" << endl << endl;

    for (auto persona : mesaVA)
        cout << persona << endl;

    cout << endl << "La conveniencia global de la mesa es de: " << VA << endl;
     cout << "El algoritmo de vuelta atras ha tardado: " << tiempoVA.count() << endl;
#else

    ofstream archivoVA("cenaGalaVA.dat",ofstream::app);
 #if !ONLYVUELTAATRAS
    ofstream archivoFB("cenaGalaFB.dat",ofstream::app);

    tAntes = high_resolution_clock::now();
    fuerzaBruta(conveniencias, mesaFB);
    tDespues = high_resolution_clock::now();
    duration<double> tiempoFB = duration_cast<duration<double>>(tDespues - tAntes);

    cout << "Fuerza bruta: " << N << " " << tiempoFB.count() << endl;
    archivoFB << N << " " << tiempoFB.count() << endl;
 #endif

    tAntes = high_resolution_clock::now();
    vueltaAtras(conveniencias, mesaVA);
    tDespues = high_resolution_clock::now();
    duration<double> tiempoVA = duration_cast<duration<double>>(tDespues - tAntes);

    cout << "Vuelta atras: " << N << " " << tiempoVA.count() << endl;
    archivoVA << N << " " << tiempoVA.count() << endl;
#endif
```

4. Ejemplo de Ejecución

Sin más, os dejamos con una captura del resultado generado por los dos algoritmos. Encontramos una gran mejora en el algoritmo de backtracking llegando a ser hasta 3 veces más rápido que el de fuerza bruta.

```
Matriz de conveniencias
-1      56      73      35      92      0       60      88      52      96
98      -1      17      66      32      53      70      79      25      7
58      33      -1      34      58      24      3       17      93      97
1       66      78      -1      48      33      6       34      1       41
84      91      28      65      -1      92      57      85      0       56
28      89      73      0       70      -1      26      45      34      22
94      88      48      72      98      89      -1      77      33      70
67      16      61      50      6       65      77      -1      67      83
85      14      53      0       30      49      33      9       -1      41
69      56      63      70      76      58      46      32      78      -1

FUERZA BRUTA

0
7
6
4
5
1
3
9
2
8

La conveniencia global de la mesa es de: 1454
El algoritmo de fuerza bruta ha tardado: 2.72985

VUELTA ATRAS

0
7
6
4
5
1
3
9
2
8

La conveniencia global de la mesa es de: 1454
El algoritmo de vuelta atras ha tardado: 0.96112
```

4. Gráficas

En ellas podemos ver representado en la parte de abajo el número de comensales y en el lateral el tiempo empleado.

Analizando primero la de fuerza bruta vemos que llegamos a alcanzar un pico de 70000 segundos para el caso de 14 comensales, siendo en el caso de backtracking para este caso poco mayor de 1000 segundos.

Encontramos para este algoritmo un pico de algo más de 30000 segundos en el caso de 16 comensales.

Finalmente tenemos una gráfica comparativa en la que vemos claramente el beneficio que obtenemos utilizando este tipo de algoritmos.

Las gráficas generadas son las siguientes:

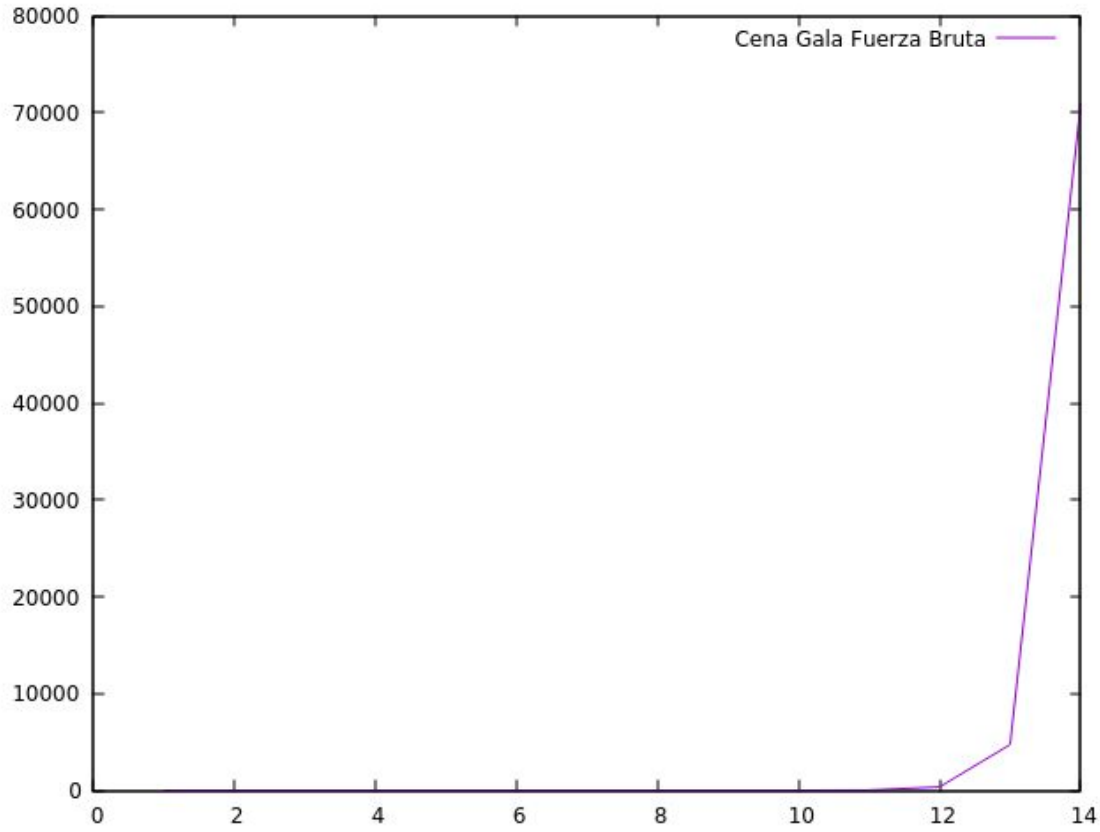

Figura 1 : Gráfica del Algoritmo de fuerza Bruta

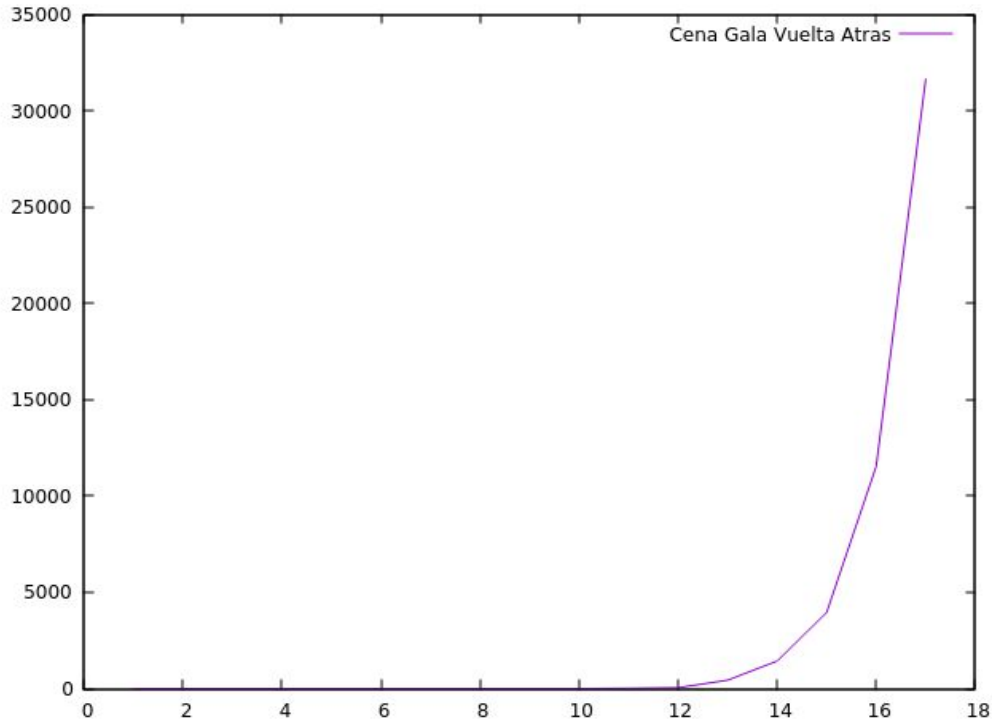

Figura 2 : Gráfica del Algoritmo Backtracking

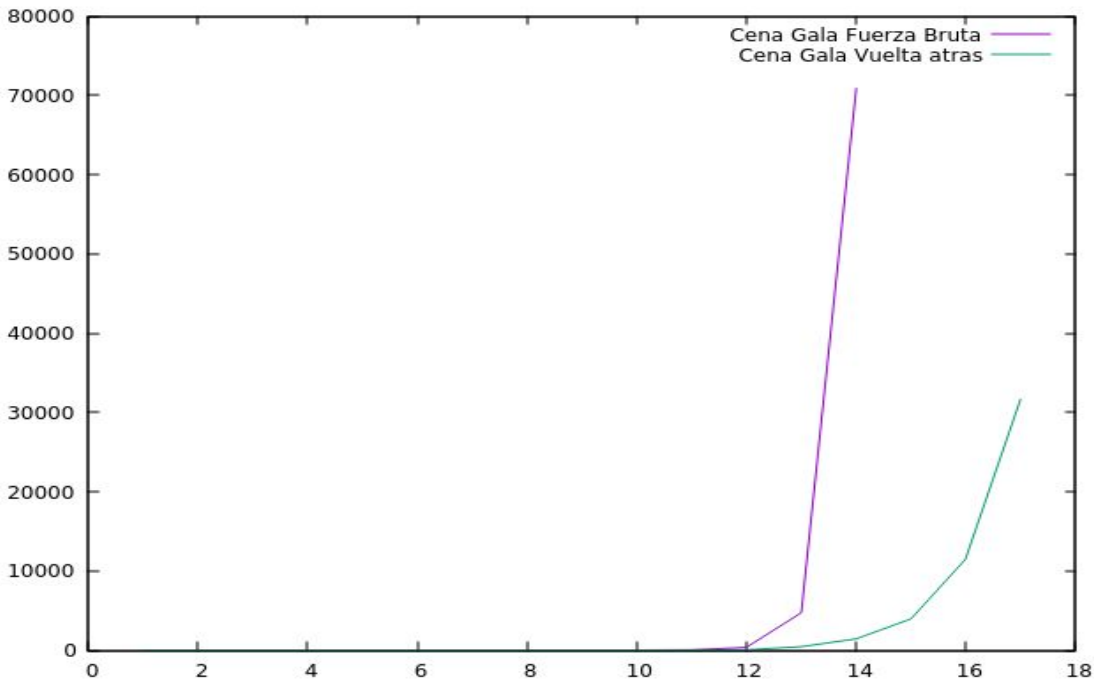

Figura 3 : Gráfica Comparativa de los 2 algoritmos

5. Anexo – Datos generados

5.1 – Cena de gala fuerza bruta

Nº COMENSALES	TIEMPO
1	1.0344e-05
2	9.787e-06
3	1.2733e-05
4	2.6982e-05
5	0.000108002
6	0.000609496
7	0.00412485
8	0.0304424
9	0.222308
10	2.34565
11	27.1133
12	344.752
13	4711.45
14	70855.5

5.2 – Cena de gala backtracking

Nº COMENSALES	TIEMPO
1	2.2457e-05
2	2.4448e-05
3	4.3537e-05
4	0.000141079
5	0.000577136
6	0.00188217
7	0.00987799
8	0.0539059
9	0.282056
10	0.834245
11	17.0584
12	44.5635
13	437.105
14	1429.8
15	3951.71
16	11511.2
17	31627