PRÁCTICA 4

Algoritmos de vuelta atrás y de ramificación y poda

Manuel Contreras Orge: 20%
Raúl Durán Racero: 20%
Alberto Palomo Campos: 20%
Abel Rios González: 20%
Salvador Romero Cortés: 20%

ÍNDICE

Dividiendo en dos grupos equilibrados

Se desea dividir un conjunto de n personas para formar dos equipos que competirán entre sí. Cada persona tiene un cierto nivel de competición, que viene representado por una puntuación (un valor numérico entero). Con el objeto de que los dos equipos tengan una capacidad de competición similar, se pretende construir los equipos de forma que la suma de las puntuaciones de sus miembros sea la misma.

Algoritmo de vuelta atrás

Para resolver el problema usando el Algoritmo de vuelta atrás, hemos creado una clase Solución la cual va a contener estos atributos y métodos:

Solucion.h

```
class Solucion {
private:
   std::vector<int> personas; // Personas con su puntuación
   std::vector<int> solucion;
   int numerosoluciones; // Numeros de soluciones (buscamos una)
   int total; // Suma total de todos las puntuaciones
   int obj; // Objetivo (Buscamos que sea total/2)
   int mejor_diferencia; // Mejor diferencia (buscamos la menor)
   std::vector<int> solucion_buena; //Solución ideal
   std::vector<int> solucion_dif_minima; //Solución en caso de
que la puntuación de ambos equipos no pueda ser igual
   void resolver(int s, int k, int r); //Función que resuelve
public:

   Solucion(const std::vector<int> &pesos); // Método solucion
   void equilibrarEquipos();
   std::vector<int> getSolucion() const; // Consultor solucion
   std::vector<int> getVectorEquipo() const; // Consultor vector
   int getTotal() const; // Consultor total
};
```

Solucion.cpp

```
#include "solucion.h"
#include <iomanip>
#include <cmath>

Solucion::Solucion(const std::vector<int> &pesos)  :
personas(pesos), solucion(pesos.size()),
```

```
solucion_dif_minima(pesos.size()){
    numerosoluciones = 0;
    total = 0;
    for (auto e: pesos){
        total += e;
    }
    obj = total/2;
    mejor_diferencia = total*2;
}

int sumaVector(const std::vector<int> & v){
    int suma = 0;
    for (auto e: v){
        suma += e;
    }
    return suma;
}

void print(const std::vector<int> &A) {
    for(int i = 0; i < A.size(); i++)
    {
        std::cout << std::setw(6) << A[i] << " ";
    }
    std::cout << "\n";
}

void Solucion::resolver_BackTrack(int s, int k, int r) {
    if (numerosoluciones != 1) { //saber si ha encontrado la
solución óptima
        solucion[k] = 1;
        if (s + personas[k] == obj) {
            numerosoluciones++;
            solucion_buena = solucion;
            print(solucion);
        } else if (s + personas[k] < obj) {    //si no te pasas =>
puede seguir explorando esa rama
resolver_BackTrack(s + personas[k], k + 1, r - personas[k]);
        }
        solucion[k] = 0;
        if ((s + r - personas[k] >= obj) && (s+personas[k+1] <=
obj)
 {
```

```
            solucion[k] = 0;
            resolver_BackTrack(s, k + 1, r - personas[k]);
        }
        if (abs(obj - s) < abs(obj - mejor_diferencia)){
            mejor_diferencia = s;
            solucion_dif_minima = solucion;
        }
    } else
        return;
}
```

Este problema lo resolvemos enfocándolo como un problema de suma de subconjuntos en el que sólo nos interesa 1 solución y la suma deseada es la mitad de la suma total de puntuaciones de los jugadores.
La función de Backtracking almacena la suma actual de ese nodo, el nivel en el que se encuentra y su total local. El esquema sigue un enfoque de fuerza bruta al que le aplicamos una serie de restricciones para limitar la búsqueda. Estas restricciones son las explícitas e implícitas definidas a continuación. De esta manera si por ejemplo vemos que con una solución superamos la suma deseada dejaremos de explorar esa rama y volveremos hacia atrás a por el siguiente hijo.

Para solucionar el problema de que sea imposible obtener una solución exacta, en cada iteración vamos obteniendo una cota cada vez más precisa que nos servirá para que en caso de que nunca se llegue al valor exacto, podamos tener una aproximación en “solucion_dif_minima”.

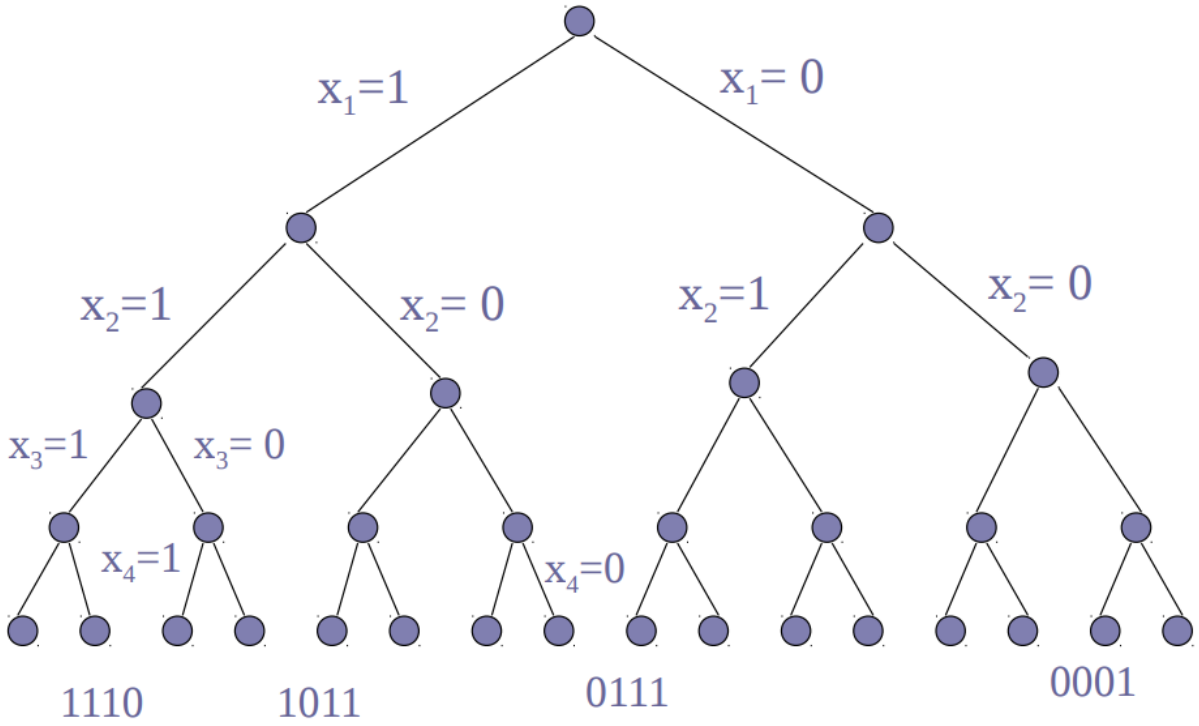
Representación del problema:

Disponemos de un vector de las puntuaciones de cada jugador del equipo, el cual está ordenado de manera ascendente.

Representación de la solución:

La solución se representa como una cadena de 0 y 1 donde un 0 indica que el jugador con tal posición pertenece a un equipo mientras que los que tengan 1 pertenecen al otro equipo. De esta manera una solución 0110 indicará que el segundo y tercer jugador pertenecen a un equipo y el primero y el cuarto al otro.

Árbol de exploración:



Árbol

Restricciones explícitas e implícitas

- Explícitas: El vector inicial debe estar ordenado de menor a mayor para llegar a una solución factible.
- Implícitas: Suma de los pesos de cada equipo debe ser igual al total/2 o minimizar la diferencia entre los dos equipos. El total de la suma debe ser par.

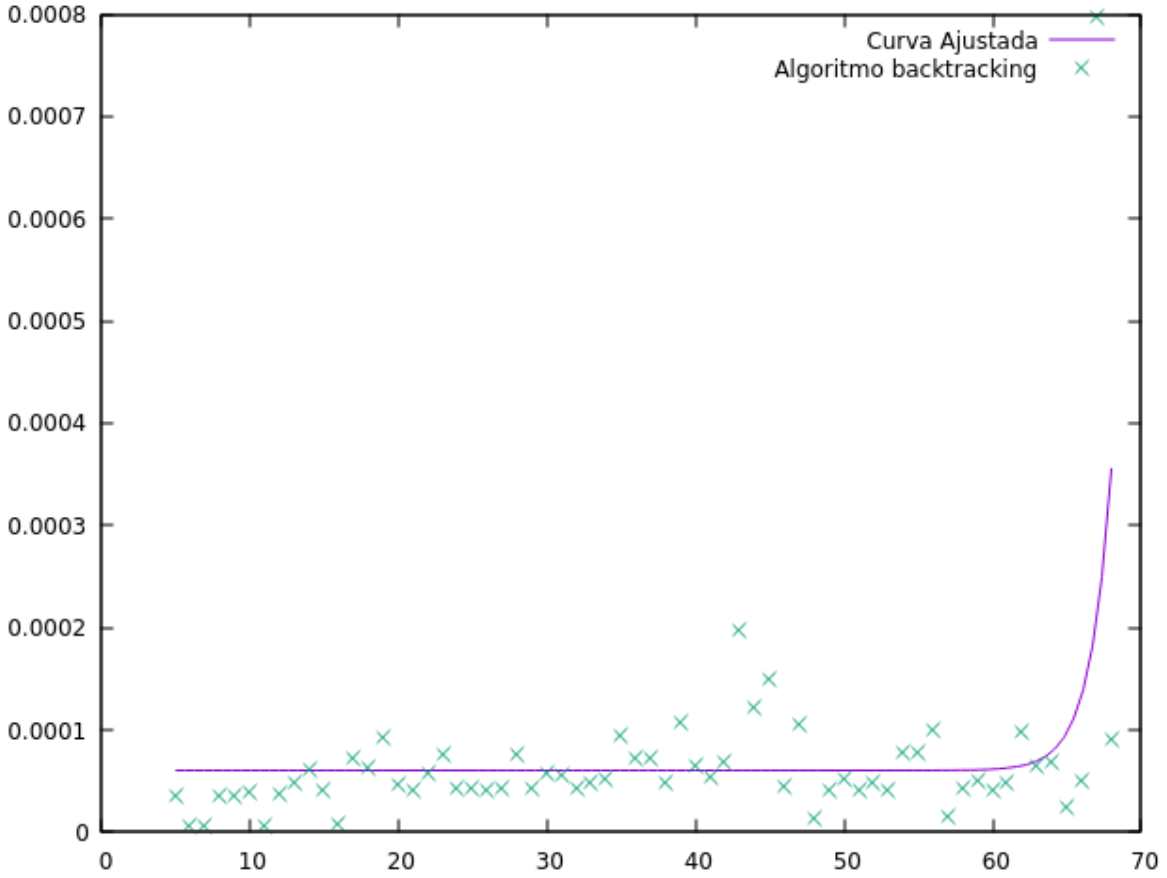
Cotas

Cota 1: Si la puntuación de un equipo pasa de un valor objetivo, igual a total/2, deja de explorar esa rama
Cota 2: Si el peso del equipo actual es mayor o igual que el objetivo, y el peso de la siguiente persona es menor o igual que el objetivo, pasa a meter en el otro equipo

Eficiencia teórica

La eficiencia teórica de este algoritmo sería de 2^n en el peor de los casos aunque sabemos que este no es su promedio.

Eficiencia experimental. Gráfica




```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 9.98334e-25     +/- 2.686e-25    (26.9%)
b                = 6.0542e-05      +/- 1.144e-05    (18.9%)

correlation matrix of the fit parameters:
                 a      b
a                1.000
b               -0.217  1.000
```

Algoritmo Fuerza Bruta

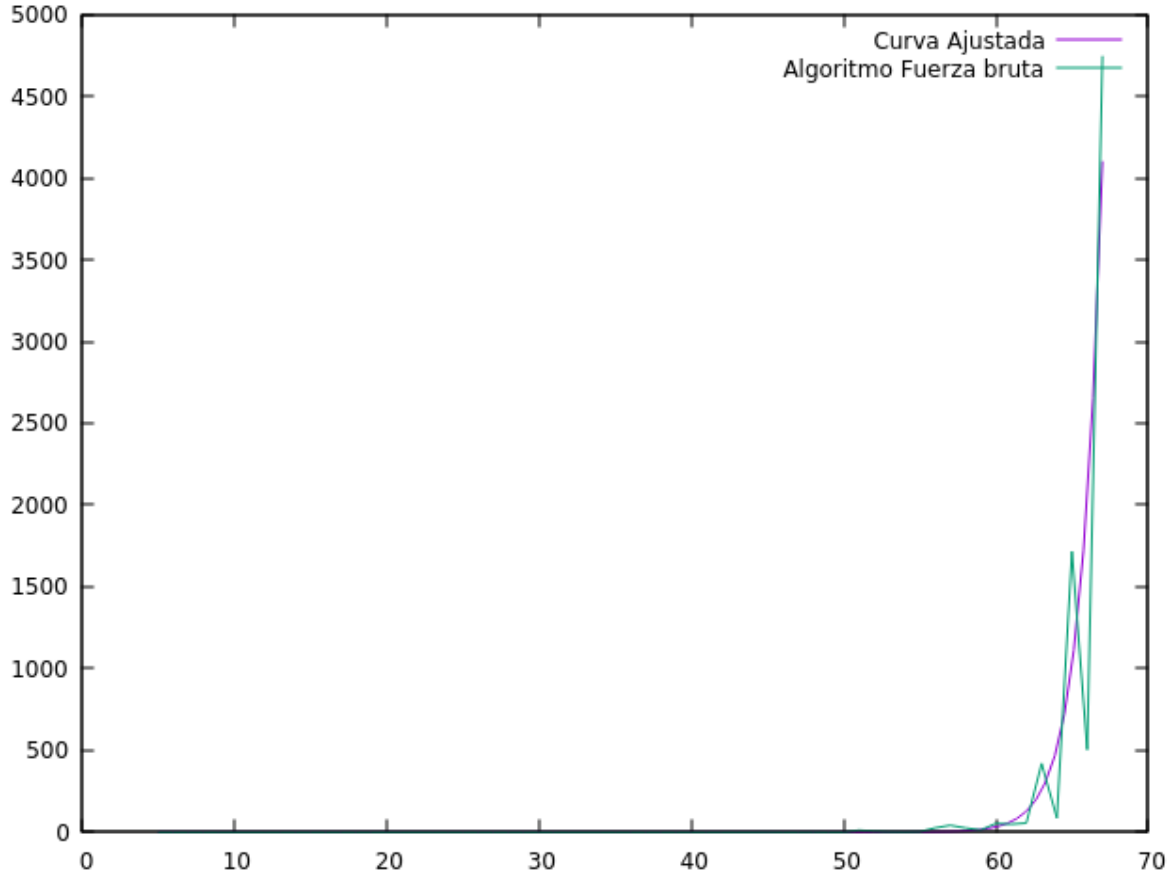
```
void Solucion::resolver_FB(int s, int k) {
    if (numerosoluciones == 0) {
        if (k == personas.size()) {
            if (s == obj) {
                numerosoluciones++;
                print(solucion);
                solucion_buena = solucion;
            }
            if (abs(obj - s) < abs(obj - mejor_diferencia)){
                mejor_diferencia = s;
                solucion_dif_minima = solucion;
            }
            return;
        }
        solucion[k] = 0;
        resolver_FB(s, k + 1);
        solucion[k] = 1;
        resolver_FB(s + personas[k], k + 1);
    } else return;
}
```

Código diferente con respecto al programa de vuelta atrás que le hace ser de fuerza bruta. La diferencia es que en fuerza bruta no va comparando si el resultado es menor o mayor al peso objetivo, si no que comprueba simplemente si es igual o no. Si es igual, termina, si no, continúa. Es decir, genera todas las posibles soluciones (cadenas de 0 y 1) y las compara con la suma que deseamos.

Eficiencia teórica

La eficiencia teórica de este algoritmo sería de O(2^n) ya que la ecuación de recurrencia es T(n) = 2 T(n-1) +1 (exploración completa del grafo como el peor de los casos).

Eficiencia experimental. Gráfica




```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.77421e-17      +/- 1.439e-18    (5.186%)
b               = 1                +/- 30.89        (3089%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.218  1.000
```

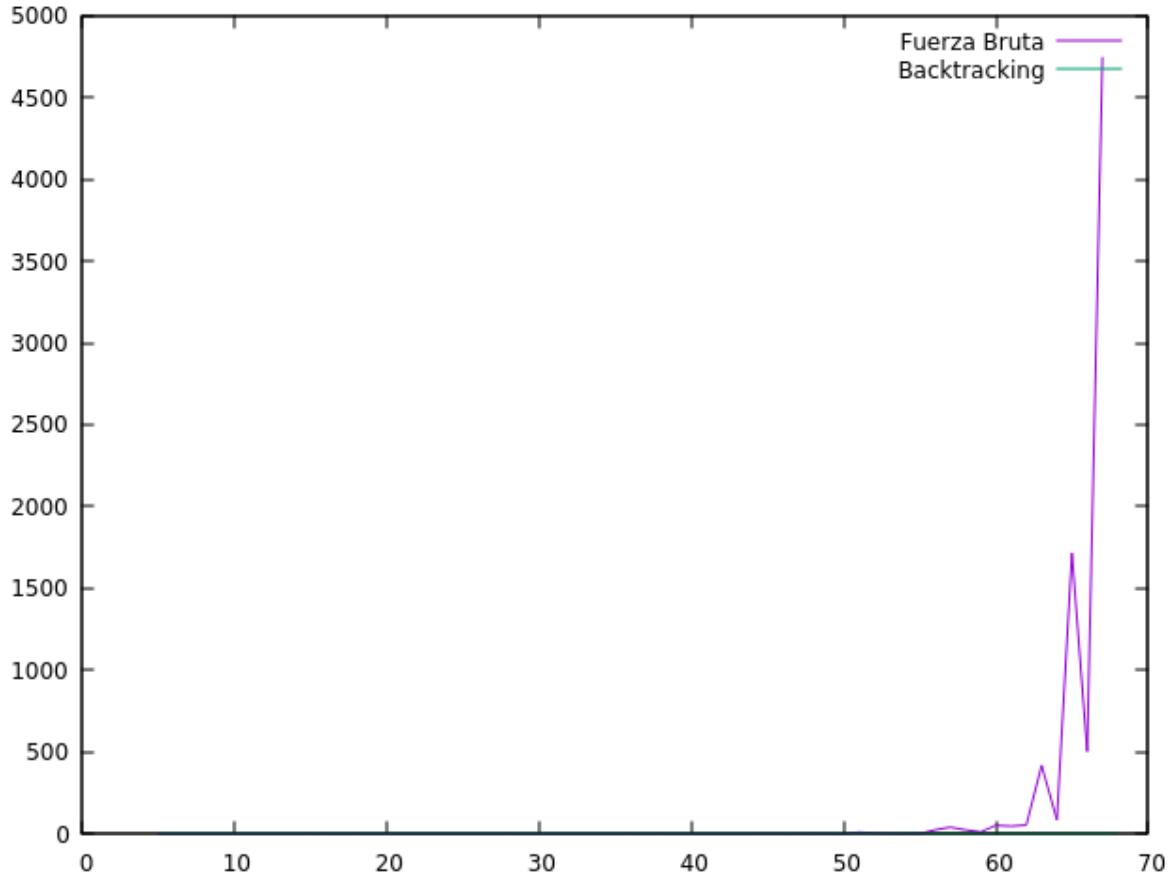
Como podemos observar, tiene un crecimiento exponencial y lo hemos podido notar en el incremento de tiempos de ejecución dependiendo de cómo subía el tamaño.
Podemos destacar picos de subida y bajada pero esto es normal ya que con el azar el algoritmo de fuerza bruta puede encontrar la solución al principio o al final, haciendo que varíen mucho más estos tiempos.

Main probando ambos algoritmos

```
#include "solucion.h"
#include <algorithm>
#include <fstream>
using namespace std;

int puntuaciones_random(int max) {  // Entre 0 y 2147483647
    return rand() % max + 1;
}

vector<int> generaParticipantes(int n, int m) {
    vector <int> p(n,0);
    int max = 2147483647; //las puntuaciones serán números reales entre
0 y 2147483647
    long long suma = 0;
    srand (time(NULL));
    if (!(m <= 0 || m > max))
        max = m;

    for (int i = 0; i < n; i++) {
        int u = puntuaciones_random(max);
        p[i] = u; //uniforme entre 1 y max
        suma += p[i];
    }

    if(suma % 2 != 0){ // Si la suma de todas las puntuaciones es impar,
sumamos 1 al último elemento del array.
        p[n-1] += 1;
        suma++;
    }

    cout << "Suma: " << suma << endl;
    return p;

}
int main(int argc, char ** argv)
{
    int arg1, arg2;

    if(argc != 2 && argc != 3){
        cerr << "Error al introducir los argumentos. Use: ./binario
<numero_personas> <maximo_puntuacion>" << endl;
        exit(-1);
    }
    if(argc == 3){
```

```
        arg2 = atoi(argv[2]);
    }
    arg1 = atoi(argv[1]);

    ofstream fichero_backtrack("salida_backtrack.txt",ofstream::app);
    ofstream fichero_fb("salida_fb.txt",ofstream::app);

    vector<int> p = generaParticipantes(arg1,arg2);
//    sort(p.begin(),p.end());
    cout << "BACKTRACKING" << endl;
    print(p);
    Solucion bt(p);
    auto inicio_bt = clock();
    bt.equilibrarEquipos();
    auto tiempo_bt = clock() - inicio_bt;
    cout << "Suma equipo 1: " << sumaVector(bt.getVectorEquipo()) <<
endl;
    cout << "Suma equipo 2: " << bt.getTotal() -
sumaVector(bt.getVectorEquipo()) << endl;
    cout << "TIEMPO: " << (double) tiempo_bt / CLOCKS_PER_SEC << endl;
    fichero_backtrack << arg1 << " " << (double) tiempo_bt /
CLOCKS_PER_SEC << endl;

    cout << "\nFUERZA BRUTA" << endl;
    Solucion fb(p);
    auto inicio_fb = clock();
    fb.equilibrarEquipos_FB();
    auto tiempo_fb = clock() - inicio_fb;
    cout << "Suma equipo 1: " << sumaVector(fb.getVectorEquipo()) <<
endl;
    cout << "Suma equipo 2: "<< fb.getTotal() -
sumaVector(fb.getVectorEquipo()) << endl;
    cout << "TIEMPO: " << (double) tiempo_fb / CLOCKS_PER_SEC << endl;
    fichero_fb << arg1 << " " << (double) tiempo_fb / CLOCKS_PER_SEC <<
endl;
    fichero_backtrack.close(); fichero_fb.close();
}
```

Comparación



Como podemos ver, conforme avanzamos en el tamaño de los vectores de pesos, vamos viendo una clara diferencia en los tiempos entre el Back tracking (apenas se distingue) y el Fuerza Bruta
Esto lo hemos notado bastante en las ejecuciones ya que como se puede ver, las ejecuciones con fuerza bruta han llegado a tardar más de 1 hora para tamaños no tan excesivamente grandes.

Problema de los recipientes

Tenemos n objetos de pesos w1, w2, . . . , wn, y un número ilimitado de recipientes iguales con capacidad m´axima R (siendo wi ≤ R, ∀i). Los objetos se deben meter en los recipientes sin partirlos, y sin superar su capacidad máxima. Se busca el m´ınimo número de recipientes necesarios para colocar todos los objetos, y conocer cuál es la asignación de objetos a los recipientes para esa configuración óptima.

Algoritmo de ramificación y poda

```
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

double uniforme() //Genera un número uniformemente distribuido en el
//intervalo [0,1) a partir de uno de los generadores
//disponibles en C.
{
    int t = rand();
    double f = ((double)RAND_MAX+1.0);
    return (double)t/f;
}

void generaPesos(vector<double> & w, int n) {

    srand(time(0));
    for (int i = 0; i < n; i++) {
        double u=uniforme();
        w.push_back(u);
    }
}

struct Nodo{
    vector<int> indice_recipientes;
    vector<double> pesos;
    int num_objetos_asignados;
    int num_recipientes;
};

struct ComparaNodos{
    bool operator()(const Nodo & a, const Nodo & n)const{
        if(a.num_recipientes > n.num_recipientes)
```

```
            return true;
        else
            return false;
    }
};


void BranchAndBound(const vector<double> & objetos, double n, int r,
vector<double> & pesos_salida, int & n_recipientes, vector<int> &
solucion_salida, int & podas_salida, unsigned long long &
nodos_explorados_salida, int & tamanio_salida){
    priority_queue<Nodo, vector<Nodo>, ComparaNodos> Abiertos;
    Nodo current;
    current.num_objetos_asignados = 0;
    current.num_recipientes = 1;
    current.indice_recipientes.resize(n);
    current.pesos.push_back(0);
    current.pesos.resize(n);
    int cota_global = n;
    bool primera_iteracion = true;
    bool salir;
    bool bandera = true;
    int n_podas = 0;
    int nodos_explorados = 0;
    int max_abiertos = 1;

    Abiertos.push(current);

    while(!Abiertos.empty() && current.num_objetos_asignados < n){
        if(max_abiertos < Abiertos.size())
             max_abiertos = Abiertos.size();
        Abiertos.pop();
        ++nodos_explorados;

        for(int i=0; i<current.num_recipientes; ++i){
            for(int j=0; j<n; ++j){
                Nodo hijoRecipienteUsado = current;
                hijoRecipienteUsado.num_objetos_asignados++;
                if(hijoRecipienteUsado.indice_recipientes[j] == 0){
                    hijoRecipienteUsado.indice_recipientes[j] = i+1;
                    hijoRecipienteUsado.pesos[i] += objetos[j];

                    if(hijoRecipienteUsado.pesos[i] <= r &&
hijoRecipienteUsado.num_recipientes < cota_global){
                        if(hijoRecipienteUsado.num_objetos_asignados ==
n && hijoRecipienteUsado.num_recipientes < cota_global){    //COTA
```

```
                            cota_global =
hijoRecipienteUsado.num_recipientes;
                            Abiertos.push(hijoRecipienteUsado);
                        }
                        else
if(hijoRecipienteUsado.num_objetos_asignados < n)
                            Abiertos.push(hijoRecipienteUsado);
                        else
                            ++n_podas;
                    }
                    else
                        ++n_podas;
                }
            }
        }

        if(!primera_iteracion){
            Nodo hijoNuevoRecipiente = current;

hijoNuevoRecipiente.pesos[hijoNuevoRecipiente.num_recipientes] = 0;
            salir = false;
            for(int i=0; i<n && !salir; ++i){
                if(hijoNuevoRecipiente.indice_recipientes[i] == 0){
                    salir = true;
                    hijoNuevoRecipiente.indice_recipientes[i] =
hijoNuevoRecipiente.num_recipientes+1;

hijoNuevoRecipiente.pesos[hijoNuevoRecipiente.num_recipientes] +=
objetos[i];
                }
            }
            ++hijoNuevoRecipiente.num_objetos_asignados;
            ++hijoNuevoRecipiente.num_recipientes;

if(hijoNuevoRecipiente.pesos[hijoNuevoRecipiente.num_recipientes-1] <=
r){
                if(hijoNuevoRecipiente.num_objetos_asignados == n &&
(hijoNuevoRecipiente.num_recipientes < cota_global || bandera)){
                    bandera = false;
                    cota_global = hijoNuevoRecipiente.num_recipientes;
                    Abiertos.push(hijoNuevoRecipiente);
                }
                else if(hijoNuevoRecipiente.num_objetos_asignados < n)
                    Abiertos.push(hijoNuevoRecipiente);
                else
                    ++n_podas;
```

```
            }
            else
                ++n_podas;
        }
        primera_iteracion=false;
        if(!Abiertos.empty())
            current = Abiertos.top();

    }

    pesos_salida = current.pesos;
    n_recipientes = current.num_recipientes;
    solucion_salida = current.indice_recipientes;
}


int main(int argc, char **argv){
    time_t ini, fin;
    vector<double> objetos;
    generaPesos(objetos, atoi(argv[1]));
    vector<double> pesos;
    int n_recipientes;
    const double R = 1;
    vector<int> solucion;
    solucion.resize(objetos.size());
    int podas, tamanio;
    unsigned long long nodos;

    for(int i=0; i<objetos.size(); ++i)
        cout << objetos[i] << " (" << i+1 << "),\t";

    ini = clock();
    BranchAndBound(objetos, objetos.size(), R, pesos, n_recipientes,
solucion, podas, nodos, tamanio);
    fin = clock();

    cout << endl;
    for(int i=0; i<solucion.size(); ++i){
        cout << solucion[i] << "\t";
    }
    cout << endl;
    for(int i=0; i<objetos.size(); i++){
        cout << "Recipiente " << i+1 << ": ";
        for(int j=0; j<objetos.size(); j++){
            if(solucion[j] == (i+1))
                cout << objetos[j] << " (" << j+1 << "),\t";
```

```
        }
        cout << endl;
    }

    cout << endl << endl << endl
         << "TIEMPO: " << ((double)(fin-ini))/CLOCKS_PER_SEC << endl
         << "TAMAÑO MAX COLA: " << tamanio << endl
         << "NODOS EXPLORADOS: " << nodos << endl
         << "NÚMERO DE PODAS: " << podas << endl;

}
```

El algoritmo recibe como entrada el vector de los pesos de los objetos, su tamaño y el tamaño de los recipientes. El resto de parámetros son de salida y no son necesarios para calcular la solución.

Primero se crea una cola de prioridad de Nodos, donde cada elemento representa una posible combinación de de asignaciones, incluyendo estados incompletos en los que no todos los objetos están asignados aún. Esta cola tiene los nodos de mayor prioridad como aquellos que llevan un menor número de recipientes usados. Se tiene que almacenar el número de objetos que se han asignado a un recipiente, el número de recipientes utilizados hasta ese momento, cómo se han asignado (vector que podría llegar a ser la solución) y los pesos acumulados de los recipientes abiertos hasta el momento.

Luego se crea un nodo raíz que se introduce a la cola en el que no se ha asignado ningún recipiente, solo hay un recipiente y los pesos y asignaciones están inicializados a 0.

Ahora, mientras la cola no esté vacía y el nodo actual no haya asignado todos sus objetos, se crean los nodos hijos del nodo actual (el nodo actual al principio es el nodo raíz, y luego se va escogiendo de la cola el elemento más prioritario).

Dichos hijos son los que surgen de introducir a partir del nodo actual uno de los objetos que falta por introducir en uno de los recipientes abiertos y un hijo que abre un nuevo recipiente, en el que se introduce algún objeto.

Los hijos que suman más del tamaño límite de recipiente en algún recipiente son podados, así como aquellos nodos que superan o igualan (en caso de que la cota superior actual no sea la cota superior inicial) la cota superior hasta el momento. La cota comienza siendo el número de objetos (máximo número de recipientes que se pueden necesitar), y cada vez que un nodo asigna todos los objetos, dicha cota se actualiza al número de recipientes usado por el nodo en caso de ser menor a la cota superior actual.

Al final, cuando se escoge el primer nodo de la cola que ha asignado todos sus elementos, este nodo contiene la solución óptima, por la forma en la que se ordena la cola, y la solución es su vector de asignaciones a recipientes.

Representación del problema

Vector con números en coma flotante (decimales) que indican los pesos de los objetos que hay que repartir en recipientes y número que determina el tamaño de los recipientes.

Representación de la solución

Vector de tamaño igual al número de elementos a repartir en los recipientes. En cada posición del vector aparece el recipiente en el que se debe introducir. Por ejemplo, si tenemos el siguiente problema:
pesos = { 0.2, 0.3, 0.4, 0.8, 0.1, 0.9}
Una posible solución sería:
s = { 1, 1 , 1, 2, 1, 3 }
Indicando que los objetos se almacenan en el recipiente con el número indicado en el vector solución.

Restricciones explícitas e implícitas

- Explícitas:
- Implícitas: La suma de los pesos de los objetos de cada recipiente no puede exceder la capacidad del recipiente.

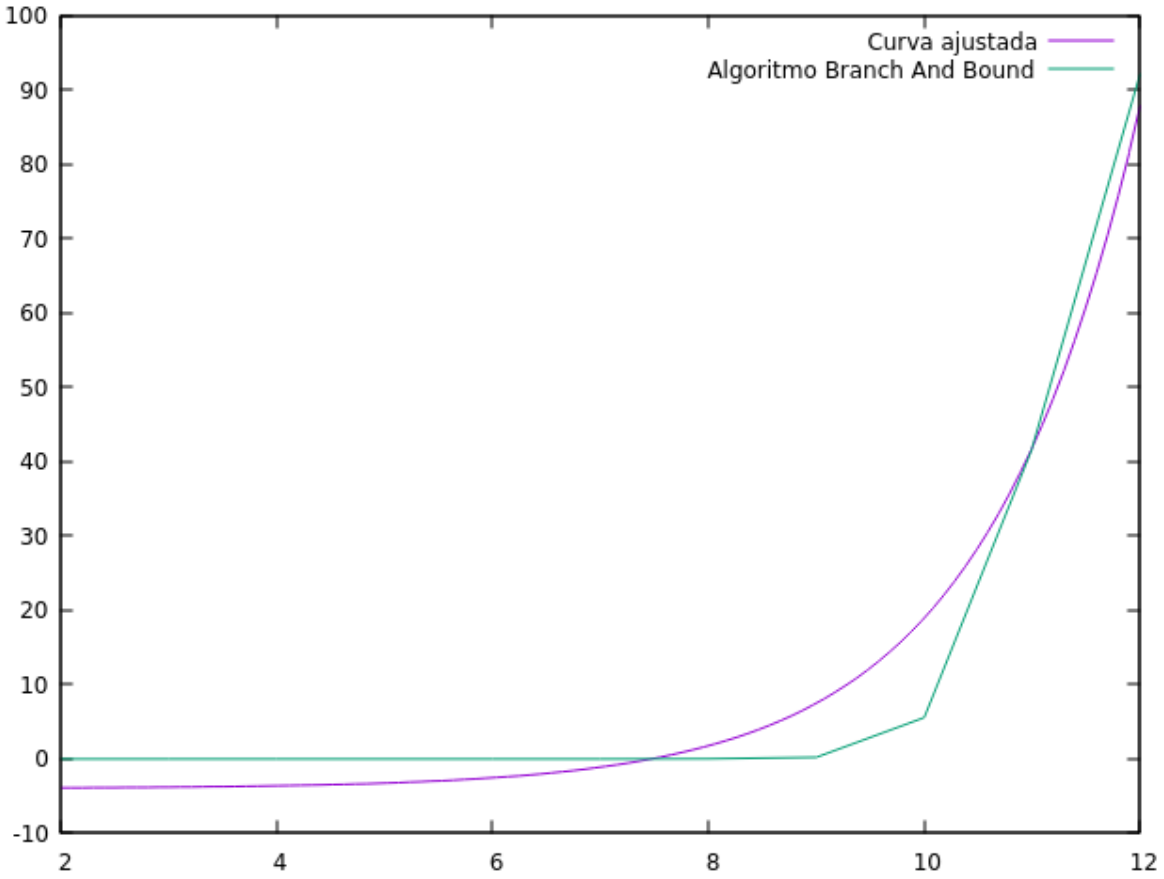
Cotas

- Cota superior: número de recipientes.
- Cota global: peor resultado en el momento de ejecutarse (se va actualizando en cada iteración)
- Cota local: número de recipientes usados por la solución actual.

Eficiencia teórica

La eficiencia teórica de este algoritmo sería de 2^n en el peor de los casos aunque sabemos que este no es su promedio.

Eficiencia empírica. Gráfica.




```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 0.0223861        +/- 0.001465     (6.542%)
b               = -3.97285         +/- 2.089        (52.57%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.522  1.000
```

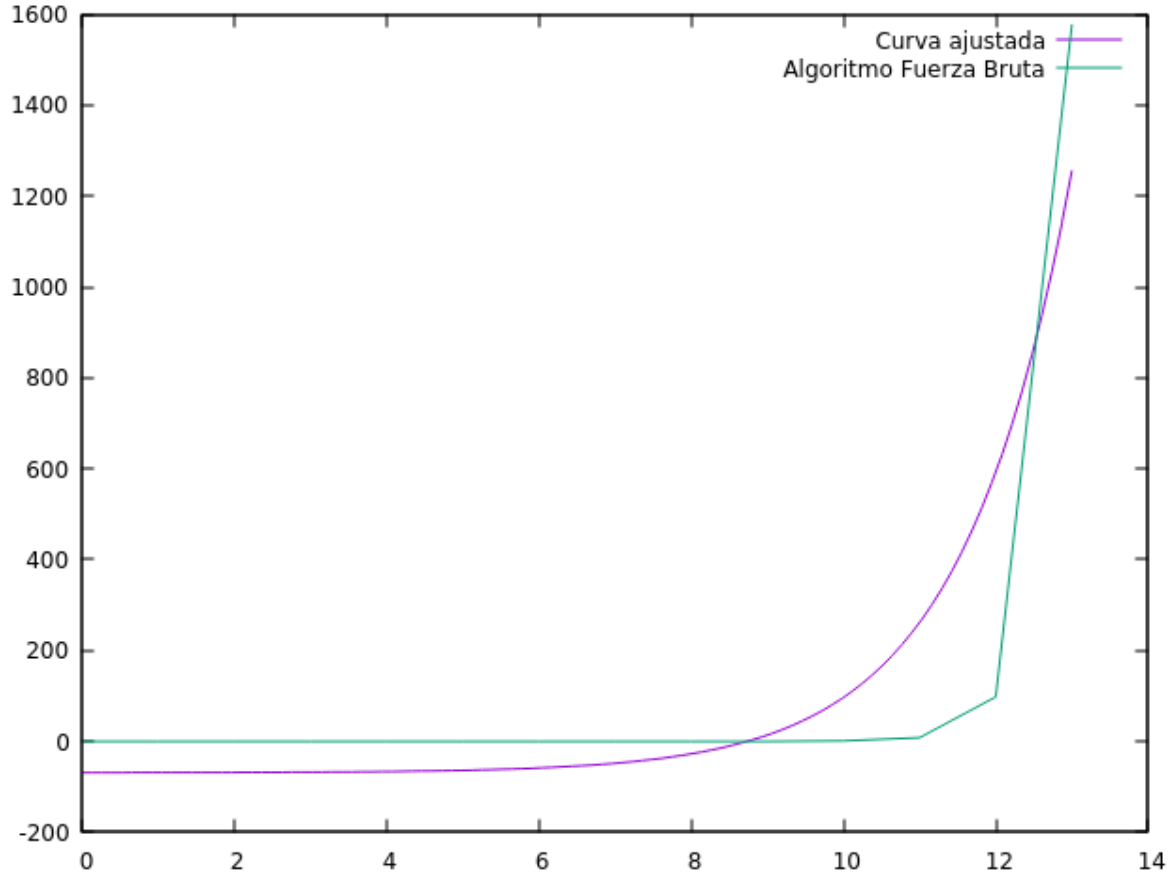
Algoritmo de fuerza bruta.

```
void recipientes_FB(int k) {

int usados;
if (k==n) {
    if (esvalido()) { //comprueba la validez de la solucion, si los
objetos no sobrepasan la capacidad de los recipientes
       usados = cuantosuso();
       if (usados < mejor) {
           mejor = usados;
           for (int i = 0; i<n; i++) solucion[i]=x[i];
       }
    }
}
else {
      for (int j=1; j<=k+1; j++) {
          x[k]=j;
          recipientes(k+1);
       }
}

}
```

Eficiencia teórica

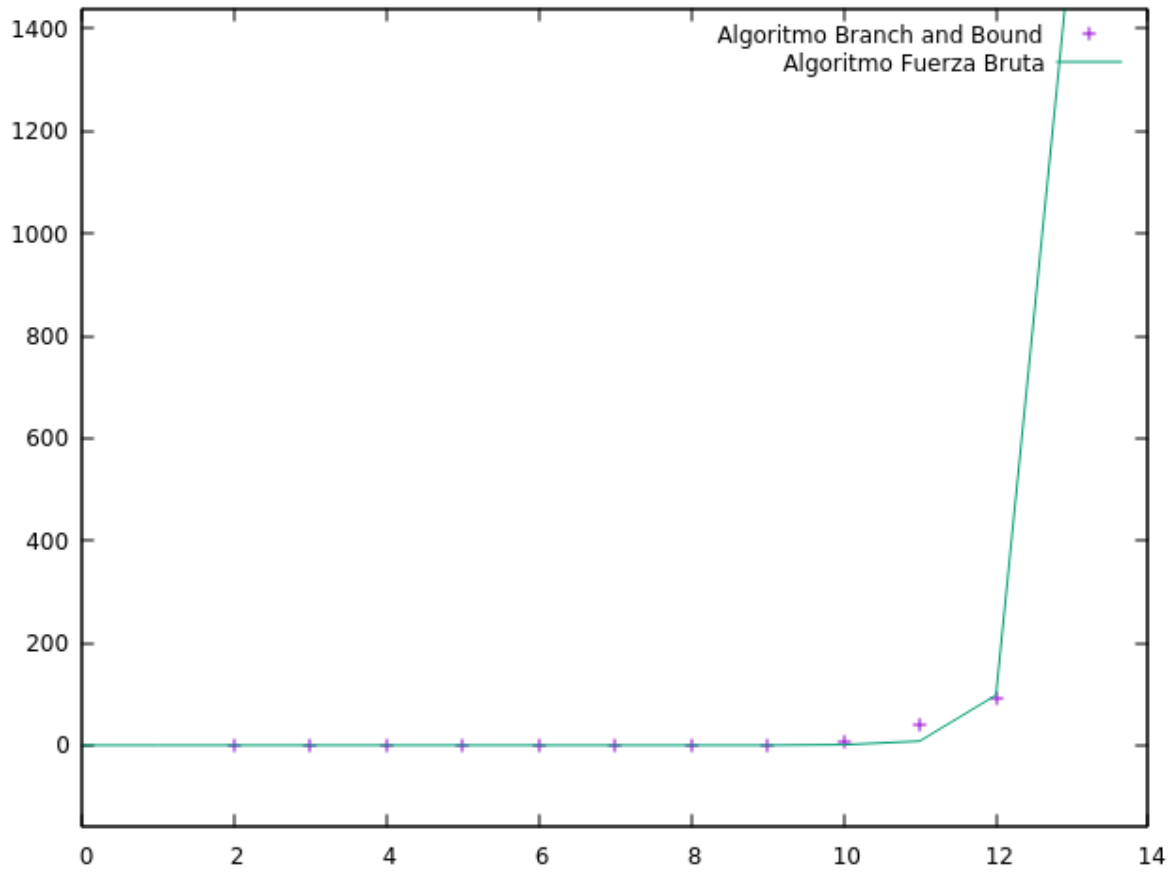
La eficiencia teórica de este algoritmo sería de O(2^n) ya que la ecuación de recurrencia es T(n) = 2 T(n-1) +1 (exploración completa del grafo como el peor de los casos).

Eficiencia experimental. Gráfica



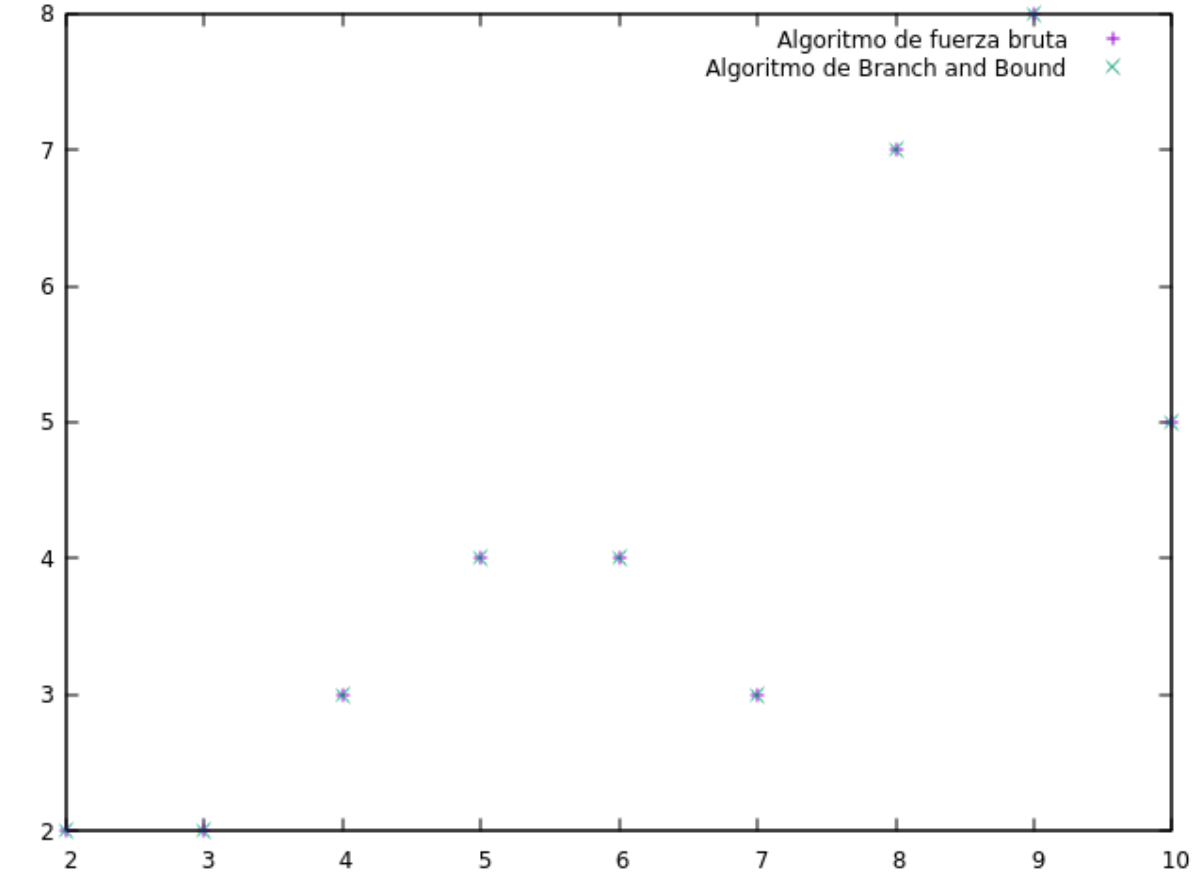

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.77421e-17      +/- 1.439e-18    (5.186%)
b               = 1                +/- 30.89        (3089%)

correlation matrix of the fit parameters:
                a      b
a               1.000
b              -0.218  1.000
```

Comparación

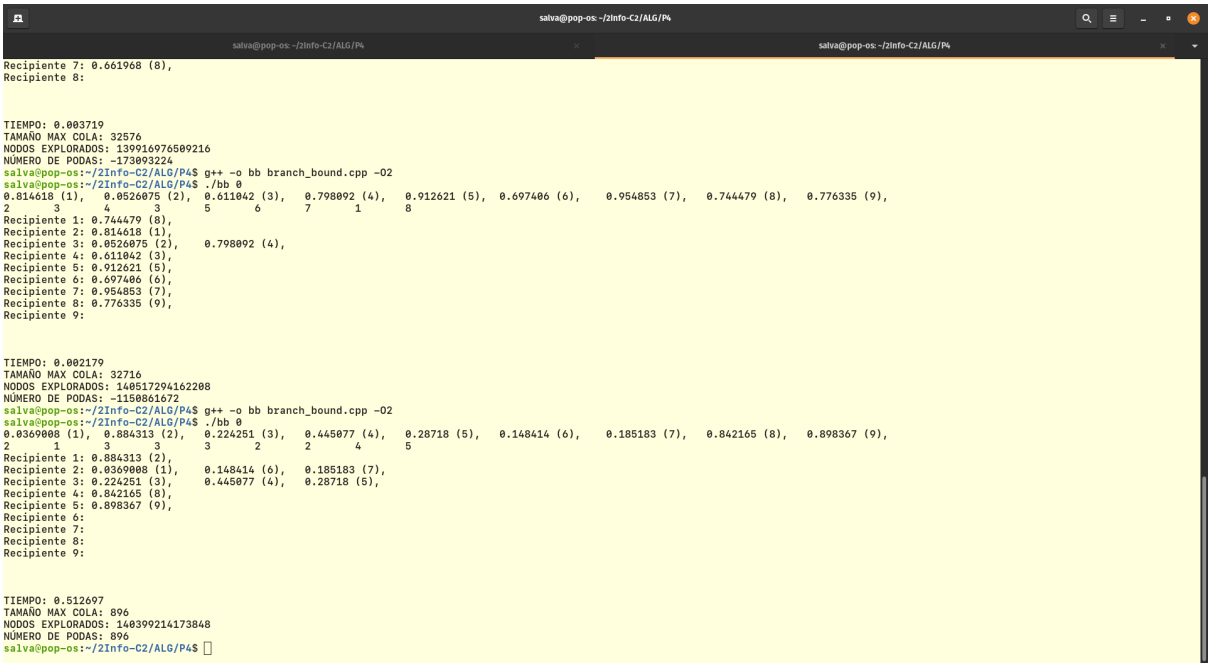


Comprobación optimalidad

Ahora comprobamos que nuestro branch and bound tiene la misma salida que el fuerza bruta.




```
 0.401882  en recipiente 1
 0.403151  en recipiente 2
 0.518676  en recipiente 2
 0.678233  en recipiente 3
 0.31736  en recipiente 3
 0.0121229  en recipiente 1

salva@pop-os:~/2Info-C2/ALG/P4$ ./recipientes-fuerzabruta 8
Los pesos son:
 0.103676  0.731691  0.512177  0.664881  0.550839  0.763324  0.741232  0.661968
tiempo: 0.001044
Se usan 7 recipientes
La distribucion es:
 0.103676  en recipiente 1
 0.731691  en recipiente 1
 0.512177  en recipiente 2
 0.664881  en recipiente 3
 0.550839  en recipiente 4
 0.763324  en recipiente 5
 0.741232  en recipiente 6
 0.661968  en recipiente 7

salva@pop-os:~/2Info-C2/ALG/P4$ ./recipientes-fuerzabruta 9
Los pesos son:
 0.814618  0.0526075  0.611042  0.798092  0.912621  0.697406  0.954853  0.744479  0.776335
tiempo: 0.009375
Se usan 8 recipientes
La distribucion es:
 0.814618  en recipiente 1
 0.0526075  en recipiente 1
 0.611042  en recipiente 2
 0.798092  en recipiente 3
 0.912621  en recipiente 4
 0.697406  en recipiente 5
 0.954853  en recipiente 6
 0.744479  en recipiente 7
 0.776335  en recipiente 8

salva@pop-os:~/2Info-C2/ALG/P4$ ./recipientes-fuerzabruta 10
Los pesos son:
 0.0369008  0.884313  0.224251  0.445077  0.28718  0.148414  0.185183  0.842165  0.898367  0.582267
tiempo: 0.141143
Se usan 5 recipientes
La distribucion es:
 0.0369008  en recipiente 1
 0.884313  en recipiente 1
 0.224251  en recipiente 2
 0.445077  en recipiente 2
 0.28718  en recipiente 2
 0.148414  en recipiente 3
 0.185183  en recipiente 3
 0.842165  en recipiente 4
 0.898367  en recipiente 5
 0.582267  en recipiente 3

salva@pop-os:~/2Info-C2/ALG/P4$
```


salva@pop-os: ~/2Info-C2/ALG/P4
salva@pop-os: ~/2Info-C2/ALG/P4
salva@pop-os: ~/2Info-C2/ALG/P4
Recipiente 7: 0.661968 (8),
Recipiente 8:
TIEMPO: 0.003719
TAMAÑO MAX COLA: 32576
NODOS EXPLORADOS: 139916976509216
NÚMERO DE PODAS: -173093224
salva@pop-os:~/2Info-C2/ALG/P4$ g++ -o bb branch_bound.cpp -O2
salva@pop-os:~/2Info-C2/ALG/P4$ ./bb 0
0.814618 (1), 0.0526075 (2), 0.611042 (3), 0.798092 (4), 0.912621 (5), 0.697406 (6), 0.954853 (7), 0.744479 (8), 0.776335 (9),
2 3 4 3 5 6 7 1 8
Recipiente 1: 0.744479 (8),
Recipiente 2: 0.814618 (1),
Recipiente 3: 0.0526075 (2), 0.798092 (4),
Recipiente 4: 0.611042 (3),
Recipiente 5: 0.912621 (5),
Recipiente 6: 0.697406 (6),
Recipiente 7: 0.954853 (7),
Recipiente 8: 0.776335 (9),
Recipiente 9:
TIEMPO: 0.002179
TAMAÑO MAX COLA: 32716
NODOS EXPLORADOS: 140517294162208
NÚMERO DE PODAS: -1150861672
salva@pop-os:~/2Info-C2/ALG/P4$ g++ -o bb branch_bound.cpp -O2
salva@pop-os:~/2Info-C2/ALG/P4$ ./bb 0
0.0369008 (1), 0.884313 (2), 0.224251 (3), 0.445077 (4), 0.28718 (5), 0.148414 (6), 0.185183 (7), 0.842165 (8), 0.898367 (9),
2 1 3 3 3 2 2 4 5
Recipiente 1: 0.884313 (2),
Recipiente 2: 0.0369008 (1), 0.148414 (6), 0.185183 (7),
Recipiente 3: 0.224251 (3), 0.445077 (4), 0.28718 (5),
Recipiente 4: 0.842165 (8),
Recipiente 5: 0.898367 (9),
Recipiente 6:
Recipiente 7:
Recipiente 8:
Recipiente 9:
TIEMPO: 0.512697
TAMAÑO MAX COLA: 896
NODOS EXPLORADOS: 140399214173848
NÚMERO DE PODAS: 896
salva@pop-os:~/2Info-C2/ALG/P4$