UNIVERSIDAD DE GRANADA

E.T.S. DE INGENIERÍAS INFORMATICA y DE TELECOMUNICACIÓN

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica 3: BACKTRACKING Estación de ITV

grupo b4

Alejandro Casado Quijada

Andrés Ortiz Corrales

Antonio Jiménez Martínez

Jesús Prieto López

Salvador Rueda Molina

Curso 2013-2014

Grado en Informática

1. Objetivo

El objetivo de esta práctica es que el estudiante aprecie la utilidad de los métodos Backtracking para resolver problemas de forma muy eficiente con un método exhaustivo, obteniendo soluciones óptimas.

2. Estación de ITV

Una estación de ITV consta de m líneas de inspección de vehículos iguales. Hay un total de n vehículos que necesitan inspección. En función de sus características, cada vehículo tardara en ser inspeccionado un tiempo ti; i = 1,…., n. Se desea encontrar la manera de atender a todos los n vehículos y acabar en el menor tiempo posible. Diseñamos e implementamos un algoritmo vuelta atrás que determine como asignar los vehículos a las líneas. Utilizando una función de factibilidad y una poda. Realizamos un estudio empírico de la eficiencia de los algoritmos.

2.1 Códigos del algoritmo

-solucion.h

```
using namespace std;

//posibles estados:0,1,2,3... lineas+1
//Comienza en linea 0
//0==NULO
//lineas+1==END
class solucion {
private:

   vector<int> sol; //solucion, cada posicion es un coche y su valor la cola
   vector<int> x;
   vector<int> vehiculos;//cada posicion un coche y su tiempo.
   int lineas,min_time;
public:
   solucion(int numero_vehiculos,int num_lineas) {
      srand(time(0));
      for(int i=0; i<numero_vehiculos; i++) {
         vehiculos.push_back((rand()%50)+1);//aleatorio entre 1-50
      }
      lineas=num_lineas;
      calcular_greedy(); //resultado greedy para tener referencia
   }

   int calc_time(const vector<int> &x2) const { //calcula el tiempo dado por x2
      vector<int> times(lineas,0); //vector con los tiempos de cada linea
      for(int i=0; i<x2.size(); i++) {
         times[x2[i]]+=vehiculos[i]; //en la cola dada por x[i] añado el tiempo del vehiculo i
```

```
      }
      int r=times[0];
      for(int i=1; i<times.size(); i++) {
         if(times[i]>r) r=times[i];
      }
      return r;
   }

   //Calcula una solución mediante algoritmo greedy como cota
   void calcular_greedy() {
      vector<int> times(lineas,0);
      for(int i=0; i<vehiculos.size(); i++) {
         int minlin=0; //linea con menor t
         for(int j=1; j<times.size() && times[minlin]>0; j++)
            if(times[minlin]>times[j]) minlin=j;
         times[minlin]+=vehiculos[i]; //añade el siguiente vehiculo a la linea con menos
tiempo
         sol.push_back(minlin);
         min_time=calc_time(sol); //calcula el tiempo solucion de greedy
      }
   }



   //añade un vehiculo a la solucion parcial x
   void add_vehicle() {
      x.push_back(0);
   }
   int num_vehiculos() {
      return vehiculos.size();
   }
   int size() {
      return x.size();
   }

   int num_lineas() {
      return lineas;
   }
   void last_vehicle_change_line() {
      x[x.size()-1]++;
   }
   void erase_vehicle() {
      x.pop_back();
   }
   void mostrar() {
      cout<<"Vehiculos:"<<vehiculos.size()<<"     lineas:"<<lineas<<endl;
```

```
      cout<<"Tiempo Solucion:"<<min_time<<endl;
      for(int i=0; i<vehiculos.size(); i++) {
         cout<<"Vehiculo "<<i<<"("<<vehiculos[i]<<") -- linea "<<sol[i]<<endl;
      }
   }

   void actualizar_solucion() {
      int tim=calc_time(x);
      if(tim<min_time) {
         sol=x;
         min_time=tim;
      }
   }
   bool factible() {
            return (calc_time(x)<min_time);
   }

};
```

- Backtracking

```
void back_recursivo(solucion &s) {
   if(s.size()<s.num_vehiculos()) {
      int lin=0;
      s.add_vehicle();
      while(lin<s.num_lineas()) {
         if(s.factible()) {
            back_recursivo(s);
            s.erase_vehicle(); //elimina el vehiculo temporal creado en back_recursivo
         }
         s.last_vehicle_change_line();
         lin++;
      }
   }
   else {
      s.actualizar_solucion();
      s.add_vehicle(); //Añade un vehiculo extra, ya que no se llama a back_recursivo y si se
llamara  a erase_vehicle
   }
}
```

3. Algoritmo.

Primero creamos tres vectores:

-Vector con las tuplas de la solución, es decir cada posición representa los vehículos y su valor la línea en la que se inspecciona.

-Vector con las tuplas de la solución parcial.

-Vector con los tiempos de cada vehículo.

Calculamos una cota superior, a partir de un algoritmo greedy de forma eficiente.

Los diferentes estado de las tuplas son 0 (corresponde al nulo), 1,2, …, lineas+1 (corresponde al END). Y los números corresponden a las diferentes tuplas.

De forma recursiva vamos trabajando cada estado y vamos comparándolo en la función de factibilidad, si su valor es mayor o igual a la cota superior, podamos. En caso contrario continuamos hasta un nodo hoja, en este caso si es factible modificamos la cota superior y el vector de tuplas solución.

4. Cálculo del tiempo Empírica

Realizaremos el cálculo de la eficiencia empírica estudiando el comportamiento del algoritmo.

Con el uso del archivo de entradas proporcionado mediremos los tiempos de ejecución para cada tamaño de entrada. Será mediante entradas de vectores aleatorios para los tiempos de cada vehículo.

Para el cálculo empírico nos hemos ayudado de la biblioteca STL::chrono para una mayor precisión en el tiempo.

Hemos creado un script para ejecutar los distintos algoritmos con diferentes tamaños de entradas.

Sabemos que el algoritmo con el que nosotros trabajamos es de orden exponencial. Para determinarlo nos fijamos en las gráficas y vemos la forma que tiene.

Cuando decimos que aplicamos el algoritmo a <<una situación concreta>> nos referimos a por ejemplo:

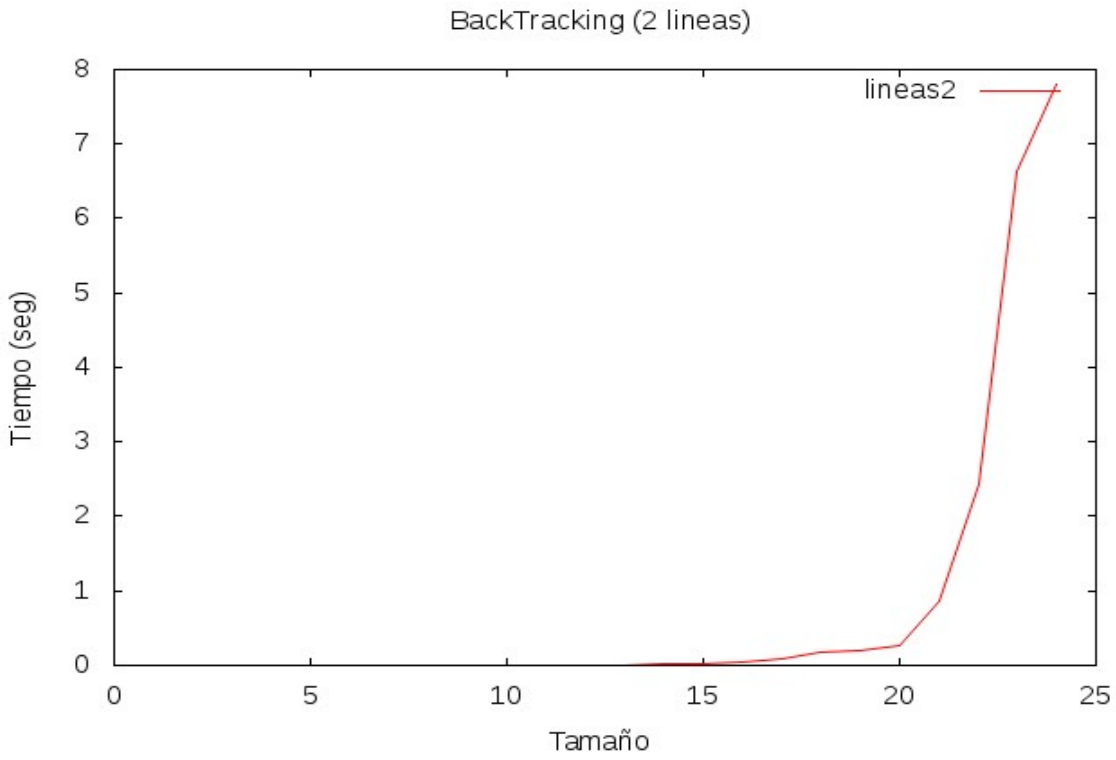
– Compilador utilizado: g++

– Ordenador sobre el que se ejecuta, en nuestro caso con las siguientes características:

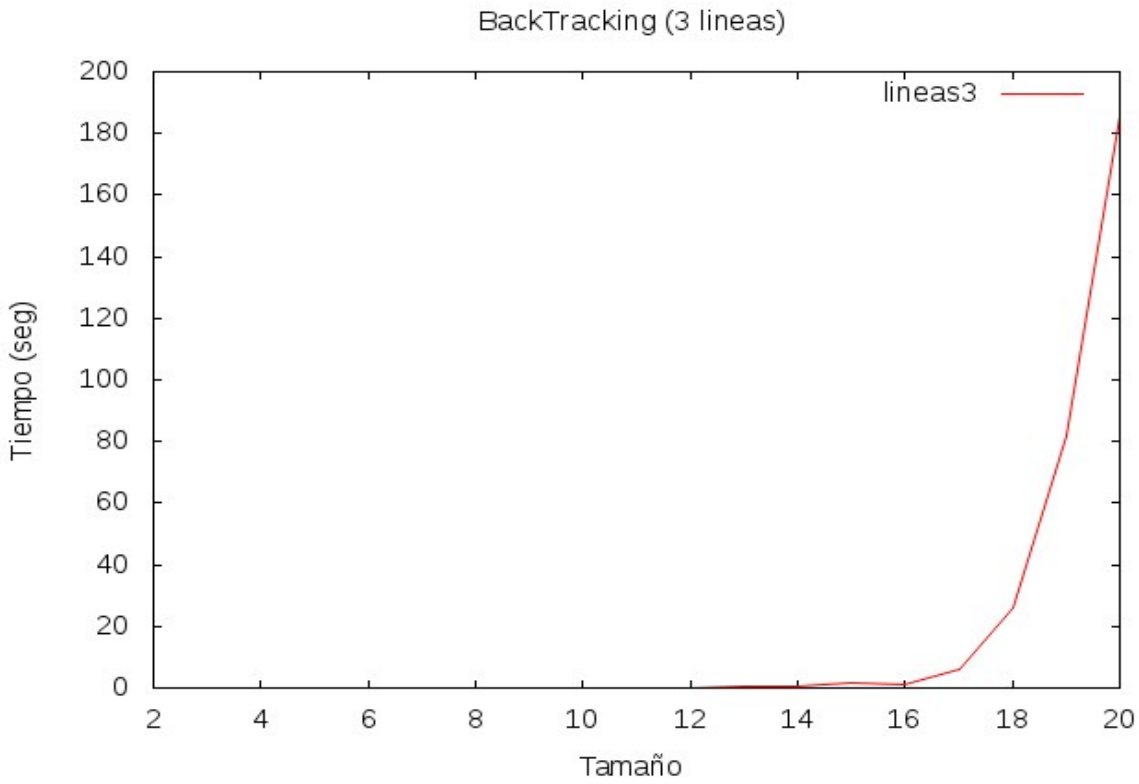
- – Intel core i3 m330 2.13 Ghz
- – 4 GB RAM
- – S.O. Ubuntu 13.04 64 bits

A continuación le mostraremos las siguientes gráficas y tablas de los diferentes algoritmos. La representación ha sido resultado de la unión de los puntos de las tablas.
También realizaremos una comparación gráficas del algoritmo de fuerza de orden lineal y el de vuelta atrás de orden logarítmico

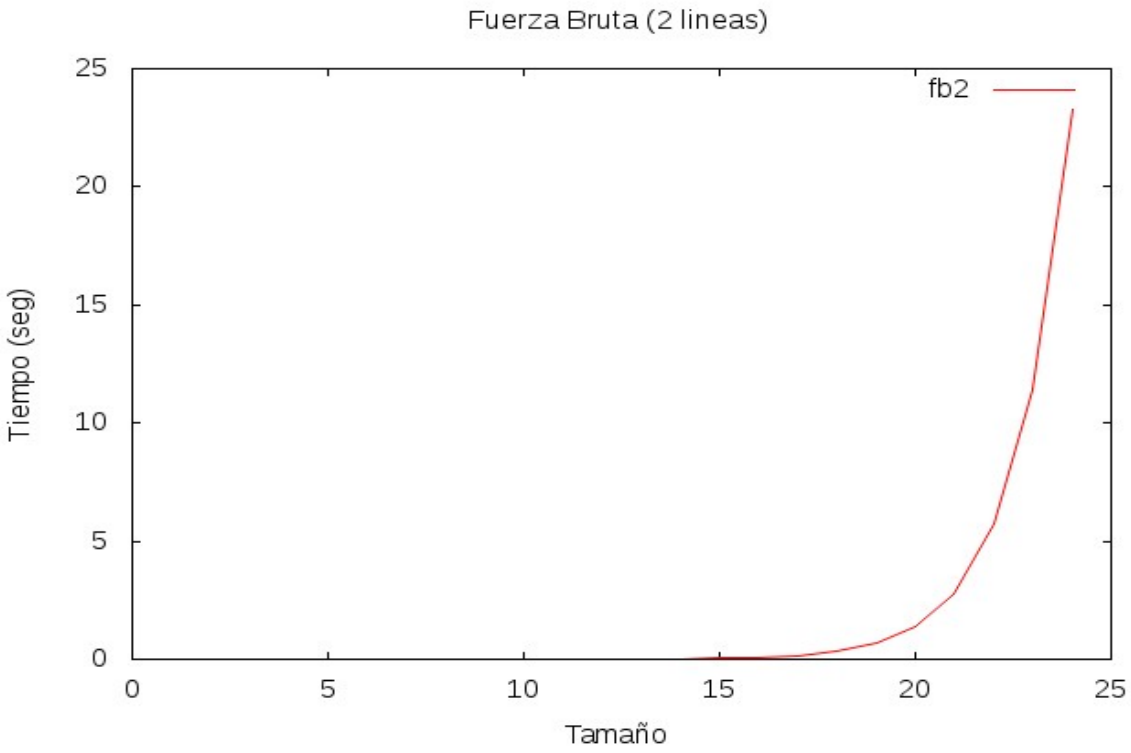
- Backtracking 2 lineas



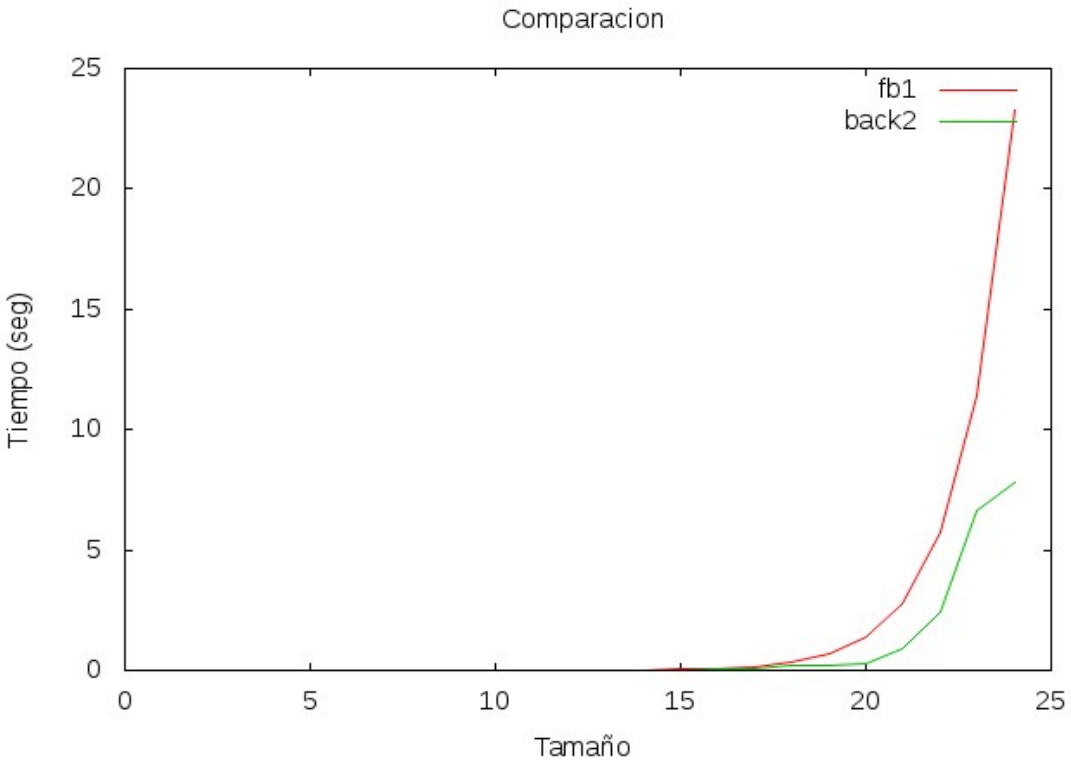
- Backtracking 3 lineas

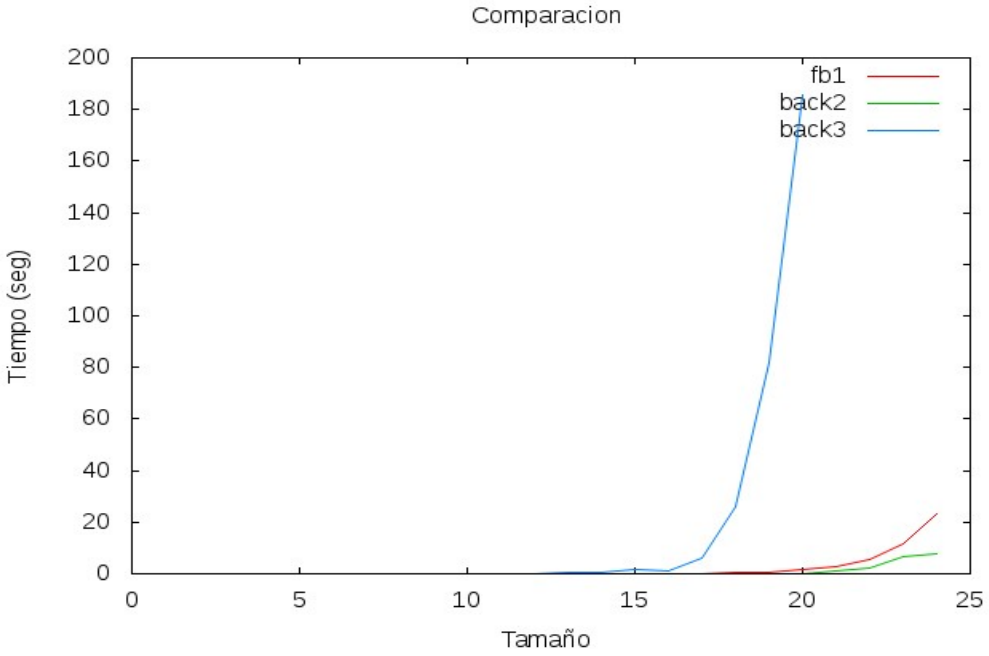


- Fuerza bruta 2 lineas



- Comparación




Comparacion
Tiempo (seg)
Tamaño
fb1
back2
back3
0
20
40
60
80
100
120
140
160
180
200
0
5
10
15
20
25

Fuerza bruta 2 lineas	
3	1.3484e-05
4	1.7118e-05
5	4.3947e-05
6	8.1844e-05
7	0.000165506
8	0.000240513
9	0.000541649
10	0.00151373
11	0.00299358
12	0.00493846
13	0.0115167
14	0.0214285
15	0.0375824
16	0.0780097
17	0.160728
18	0.317012
19	0.668638
20	1.39436
21	2.76579
22	5.73106
23	11.3899
24	23.2908

Backtracking 2 lineas	
3	9.286e-06
4	5.4348e-05
5	4.2745e-05
6	7.1437e-05
7	0.000103798
8	0.000269987
9	0.000249557
10	0.000391509
11	0.00195438
12	0.00573127
13	0.00666314
14	0.0111839
15	0.0269183
16	0.0435355
17	0.0943487
18	0.173304
19	0.194276
20	0.27261
21	0.870747
22	2.44056
23	6.62013
24	7.79135

Backtracking 3 lineas	
3	2.718e-06
4	2.652e-06
5	3.557e-06
6	4.987e-06
7	3.678e-06
8	0.000247133
9	0.0011576
10	0.0122852
11	0.027892
12	0.0284674
13	0.284654
14	0.675456
15	1.7002
16	0.879275
17	5.96139
18	25.9126
19	81.9558
20	185.873