Práctica 4 : Backtracking y Branch and Bound

04/06/202

—

Antonio Aranda Hernández
Manuel Megías Briones
Cristina Pérez Ordóñez

Hitos

I. TSP, Problema del viajante

II. Backtracking

III. Branch and bound

IV. Conclusión

Problema a resolver

El problema a resolver trata del problema del viajante, también conocido como TSP.

Este problema se formuló por primera vez en 1930.

En esta práctica, se ha modelado como un grafo de manera que los vértices son ciudades, los caminos entre ellas son las aristas y las distancias entre ellas son los pesos de las aristas. Además, tomamos como referencia un nodo, el cual es punto de partida y de llegada para los caminos a encontrar, mientras que el resto de nodos se visita una única vez.

El objetivo del problema es encontrar la distancia mínima al recorrer todas las ciudades.

Backtracking

Es una mejora del algoritmo voraz, ya que uno de los problemas que podía acarrear el uso de una técnica voraz es que el algoritmo se encuentre en un estado infinito y no encuentre ninguna solución.

Por ello apareció la técnica del backtracking, en donde se usan restricciones, para encontrar siempre una solución.

Suelen estar asociados a los árboles binarios, por ejemplo en el problema de la mochila lo cogemos o no lo cogemos.

La búsqueda en estos problemas son en profundidad, y se utilizan de dos maneras :

- Recursiva
- Iterativa

Normalmente la recursiva suele ser la más óptima en estos casos , en donde la búsqueda es en profundidad.

```
public void Backtracking(int nodo) {//método recursivo
    solucion_parcial.push(nodo);
    if(solucion_parcial.size() == this.VE.size()) {
        if(this.sonAdyacentes(solucion_parcial.firstElement(), nodo)) {
            //si son adyacentes lo añade si no no lo añade
            soluciones_backtracking.add(this.solucion_parcial);
            for(int i:this.solucion_parcial)System.out.print(VF.get(i)+" -- ");
            System.out.print(this.peso(solucion_parcial));
            System.out.println(" ");
        }
        this.solucion_parcial.pop();
        return;//cuando haga el break , salgo de la solucion
    }
    Queue<Integer> IniciarOpciones = this.adyacentes(nodo); //para saber cuando ha llegado al fin
    while(!IniciarOpciones.isEmpty()) {
        int nueva_opcion = IniciarOpciones.poll(); //sacariamos el primero de la cola
        if(!this.solucion_parcial.contains(nueva_opcion)) {
                Backtracking(nueva_opcion);
            }
        }//si ya lo contiene no hago nada
    this.solucion_parcial.pop();//vuelta atrás cuando no sea hoja
}
//end backtracking

/**
 * Cargamos el archivo, e inicializamos las variables
```

Por ello, nosotros hemos usado un técnica de Backtracking recursivo :

- ➔ Hemos usado una solución parcial en donde es una estructura estática, y por lo tanto de acceso en todas las llamadas recursivas, y que además no produce una llamada recursiva en pila de esa dirección de memoria.
- ➔ Recibe por parámetro el nodo a estudiar, que es un int (4 bytes)
- ➔ Una Queque Iniciar opciones , que ocupa 8 bytes en pila.
- ➔ Y otra nueva opción, que ocupa 4 bytes en pila.

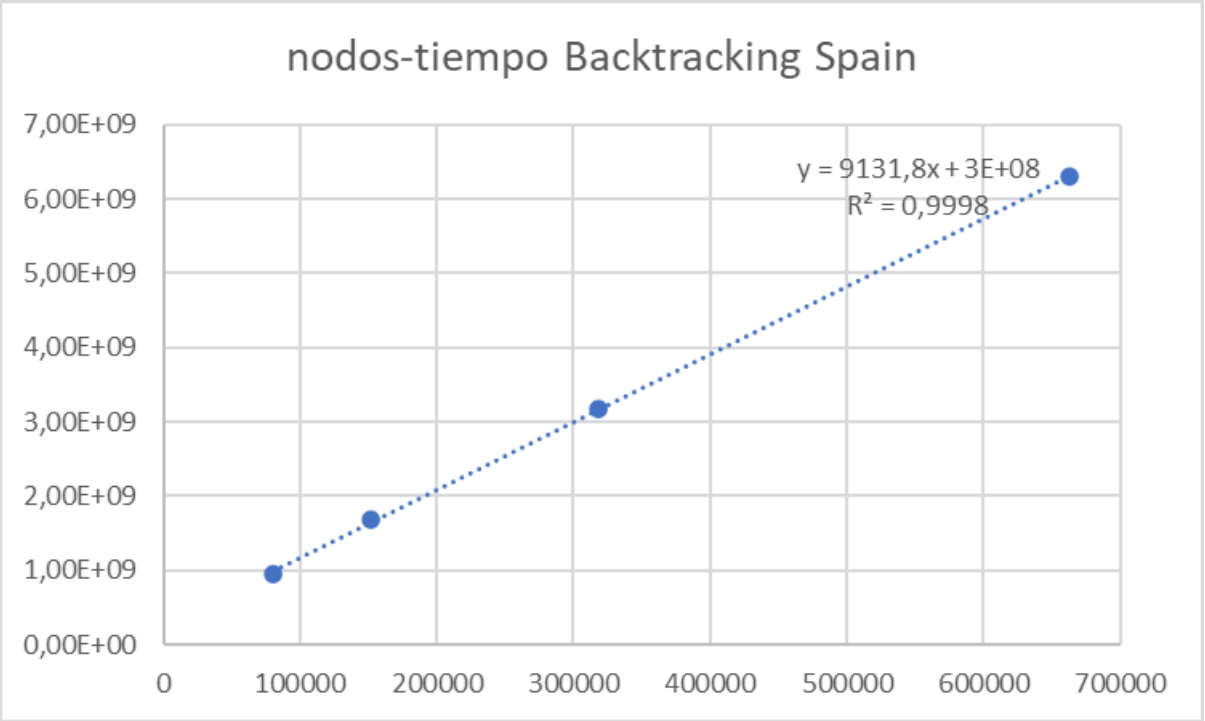
Por lo tanto necesitaríamos 16 bytes en pila, por cada llamada recursiva.

Una vez realizada la práctica para el gráfico de 21 nodos, obtenemos 34 soluciones de las cuales la de menor peso tiene 4487.

El tiempo del grafo reducido ha sido de 1,2168E8, mientras que no se ha podido obtener el caso de EdaLandLarge porque ha superado las 24h sin terminar la ejecución.

Gráfico Spain

Nº nodos visitados	Tiempo	Memoria disponible
663450	6,3139E9	155MB
79620	9,5317E8	229MB
151740	1,6869E9	236MB
317953	3,1853E9	163MB



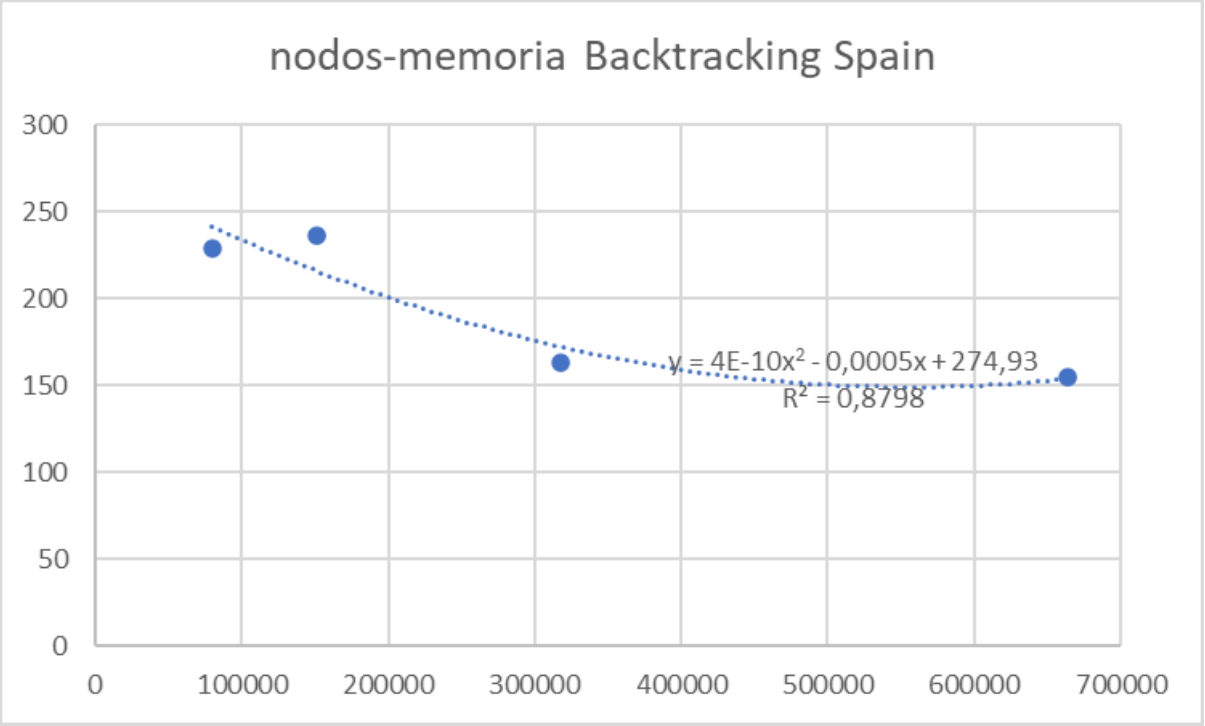
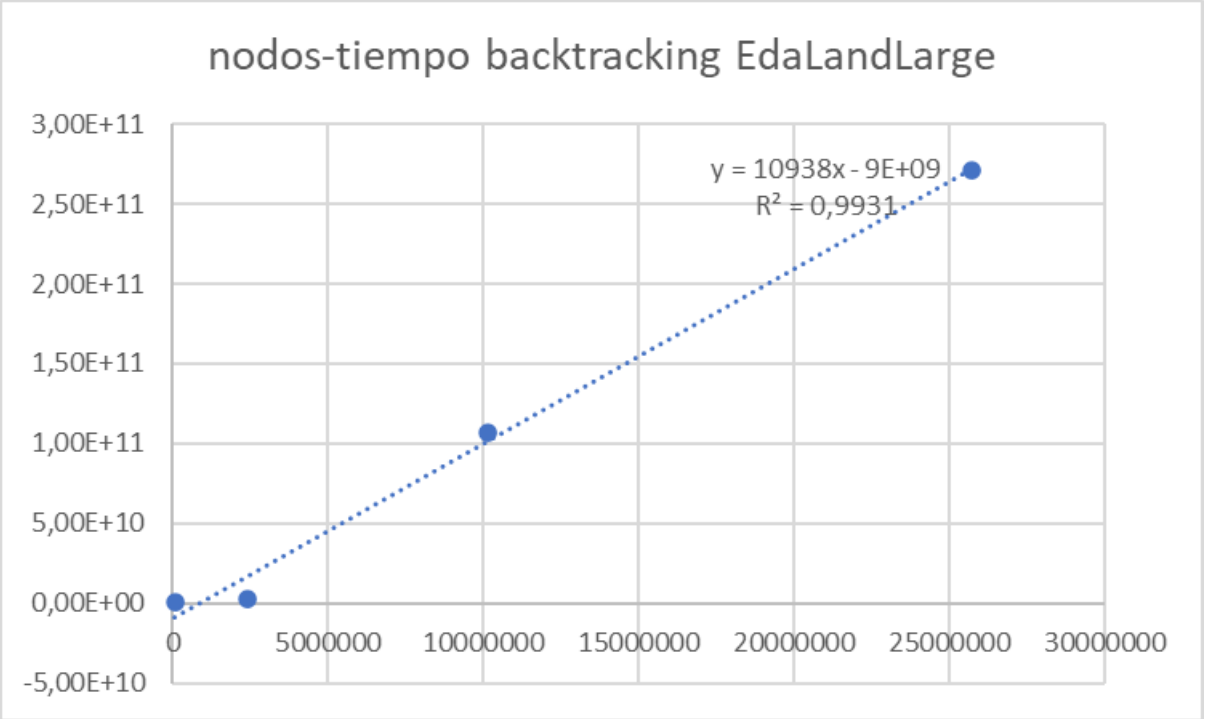
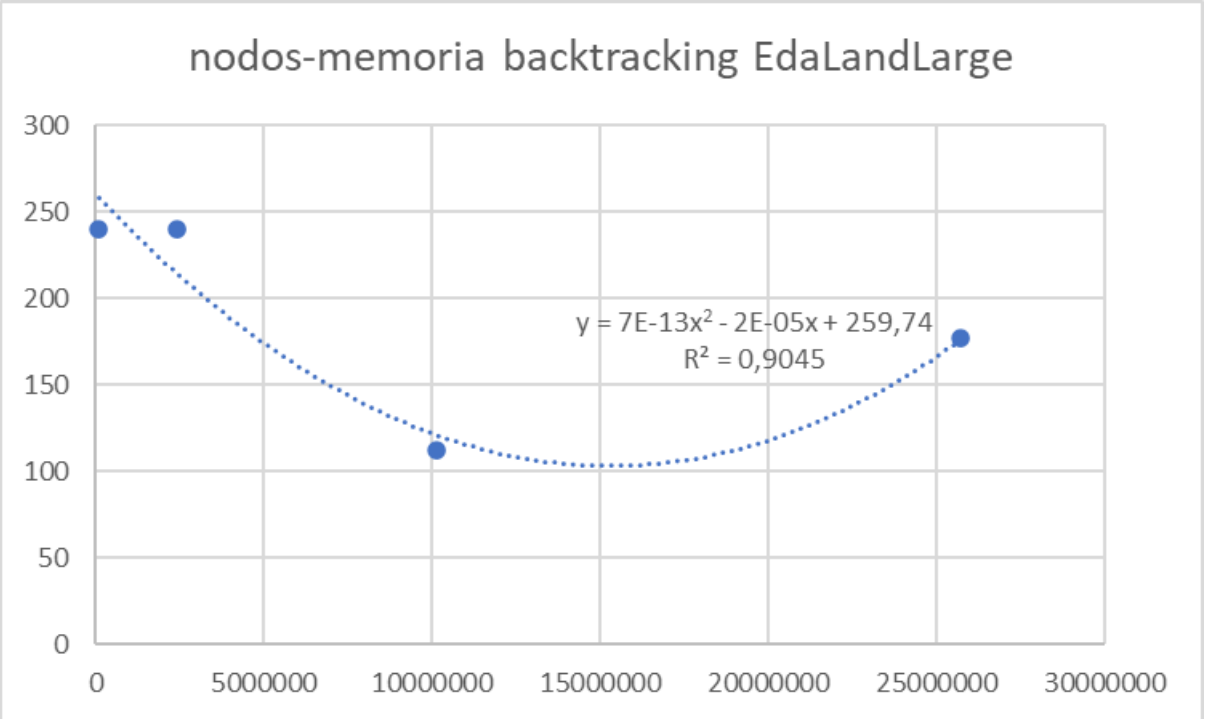


Gráfico EdaLandLarge

Nº nodos visitados	Tiempo	Memoria disponible
25702529	2,7082E11	177MB
84111	1,1734E9	240MB
2414612	2,5896E9	240MB
10134020	1,0725E11	112MB

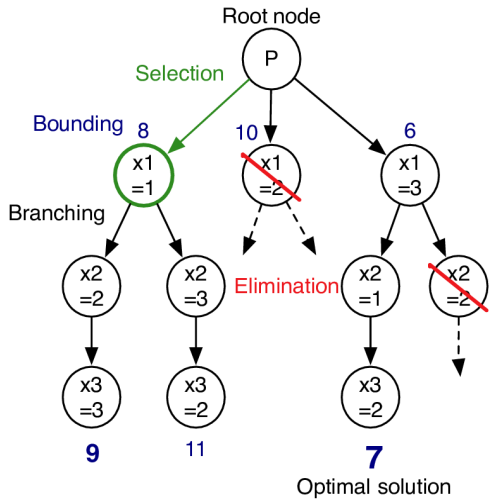




Branch and bound

Branch and bound, en español conocido como Ramificación y poda es una variante del algoritmo de Backtracking pero mejorado. Este algoritmo funciona como un “árbol” con ramas que nos llevan a una posible solución, el nombre lo recibe de la técnica que usa

para optimizar, la cual detecta las ramificaciones que dejan de ser óptimas y las “poda”, en decir las ignora para no malgastar tiempo y recursos.

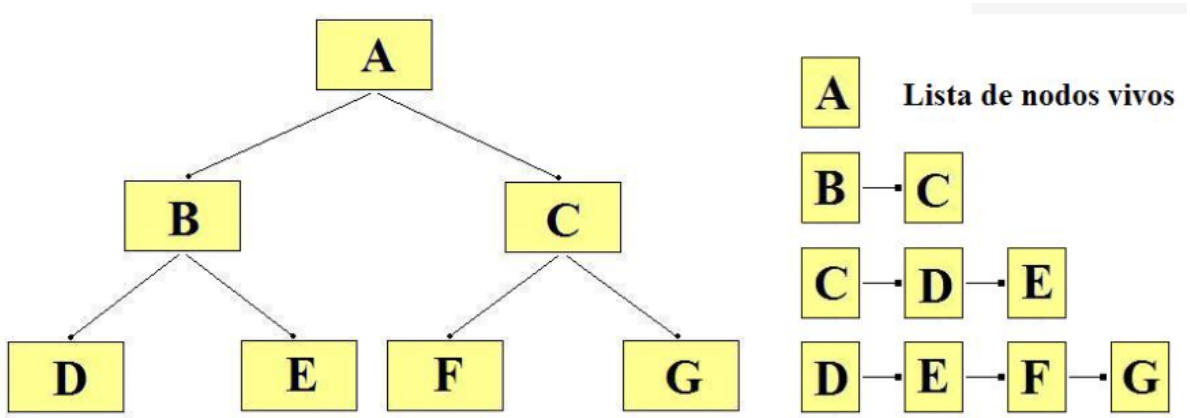


Ejemplo de Branch and Bound.

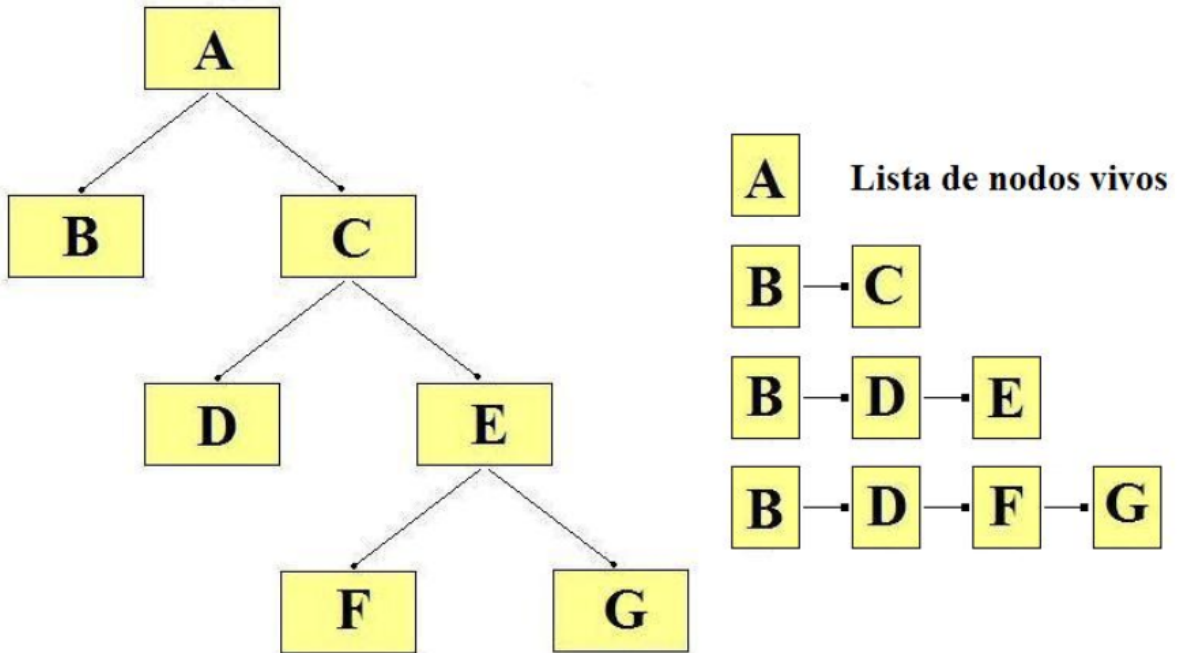
Pero, ¿Cómo funciona esto? Si el nodo hoja de la rama menor para algún árbol X es mayor que la hoja de la rama padre para otro nodo Y, entonces se descarta X de la búsqueda. Por lo general, esta poda se implementa manteniendo una variable global *m* que graba el mínimo nodo padre visto entre todas las subramas examinadas hasta entonces. Cualquier nodo cuyo nodo hijo es mayor que m es descartado.

Para las estrategias que puede seguir este algoritmo es necesario conocer las posibles ramas, los nodos que se generan y no son explorados se almacenan en la LNV (Lista de Nodos Vivos), dependiendo de cómo estén los nodos en esta lista las estrategias que se pueden seguir son:

- FIFO: En la que LNV es una cola y el árbol se recorre en anchura.



- LIFO: En la que LNV es un pila y el recorrido del árbol es en profundidad.



- LC o Menor Coste: esta estrategia se aplica junto a otra de las anteriores y lo que hace es buscar el nodo con mayor beneficio y seguirlo, si se usa con FIFO en caso de empate escogerá el primero y en el LIFO escogerá el último.

En nuestro caso hemos creado LNV como un ArrayList y por lo tanto nuestra estrategia sería LC-FIFO.

Nos ha sido imposible obtener la gráfica porque no hemos podido arreglar un pequeño fallo en el código que no nos permite terminar la ejecución del mismo.

Hemos cambiado Vertex node por String ya que la clase Vertex no permitía usar comparable Vertex fuera de esa clase.

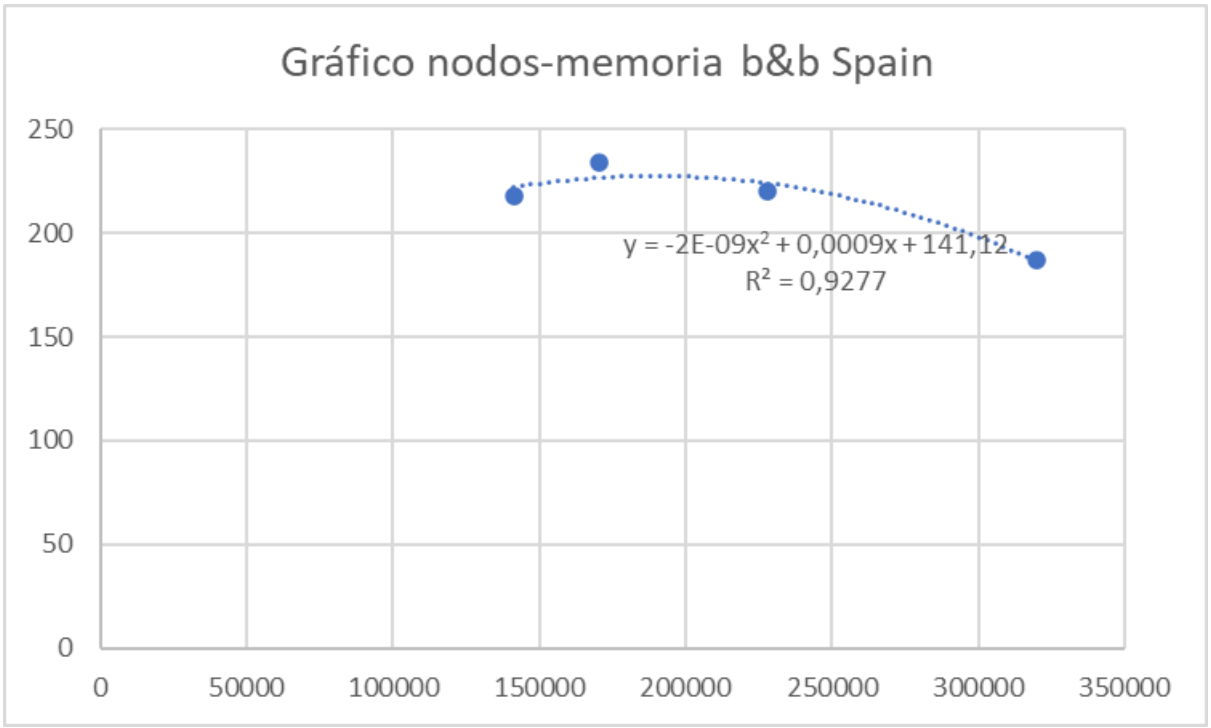
Por lo tanto, hemos dividido en otras clases para mayor eficiencia y legibilidad.

Además, se ha tenido que modificar eclipse.init para poder aumentar el tamaño de heap para la realización de pruebas para la obtención de las gráficas.

```
-Xmx512M
-XX:PermSize=64M
-XX:MaxPermSize=128M
```

Gráfico Spain

Nº nodos visitados	Tiempo	Memoria disponible
320199	1,8766E9	187MB
170649	1,0639E9	234MB
141231	8,4656E8	218MB
227910	1,4661E9	220MB



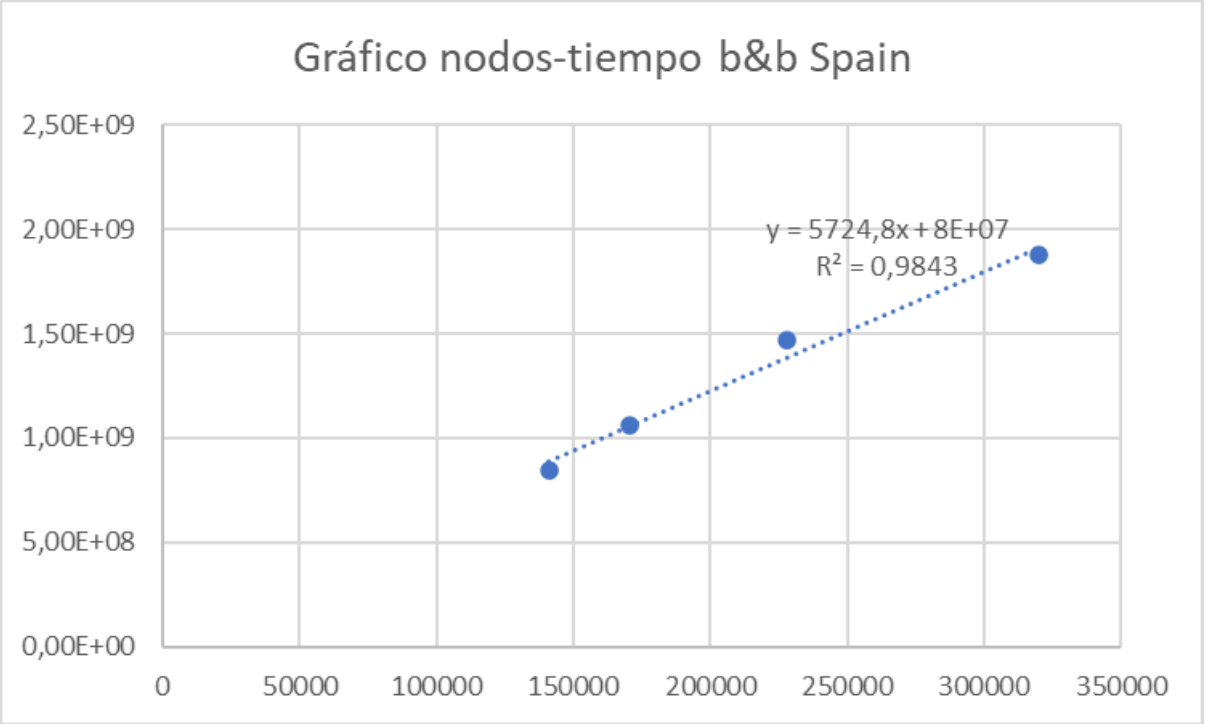
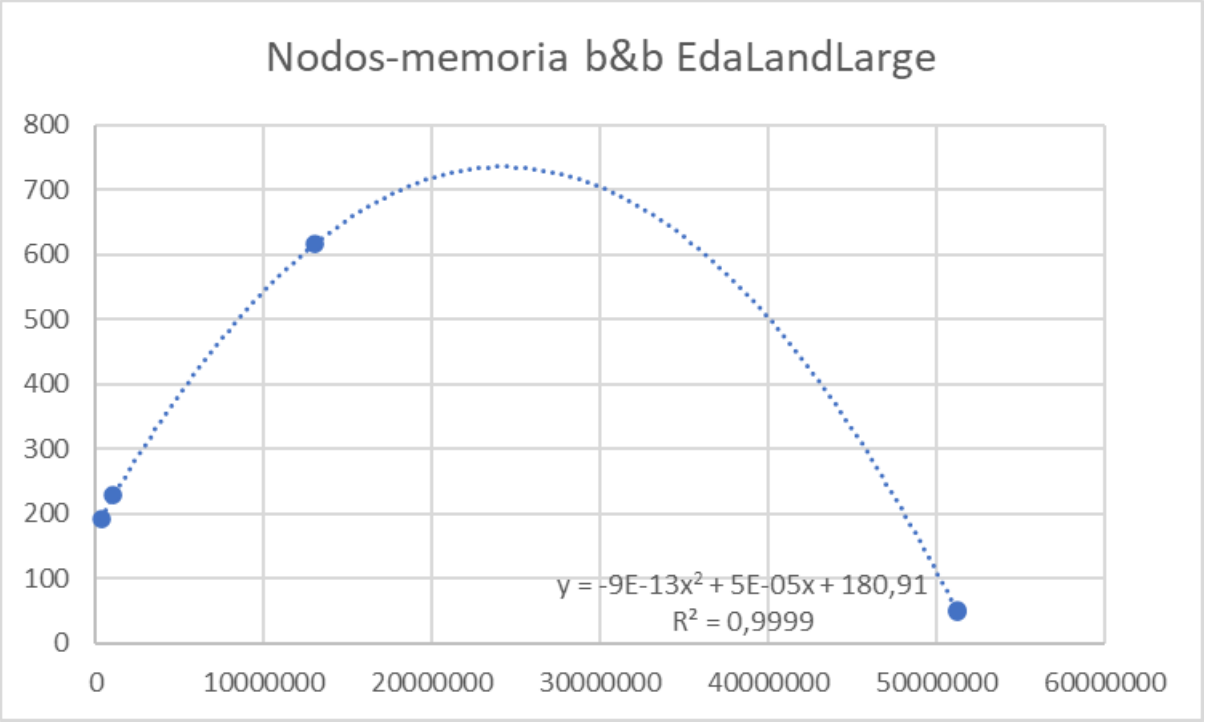
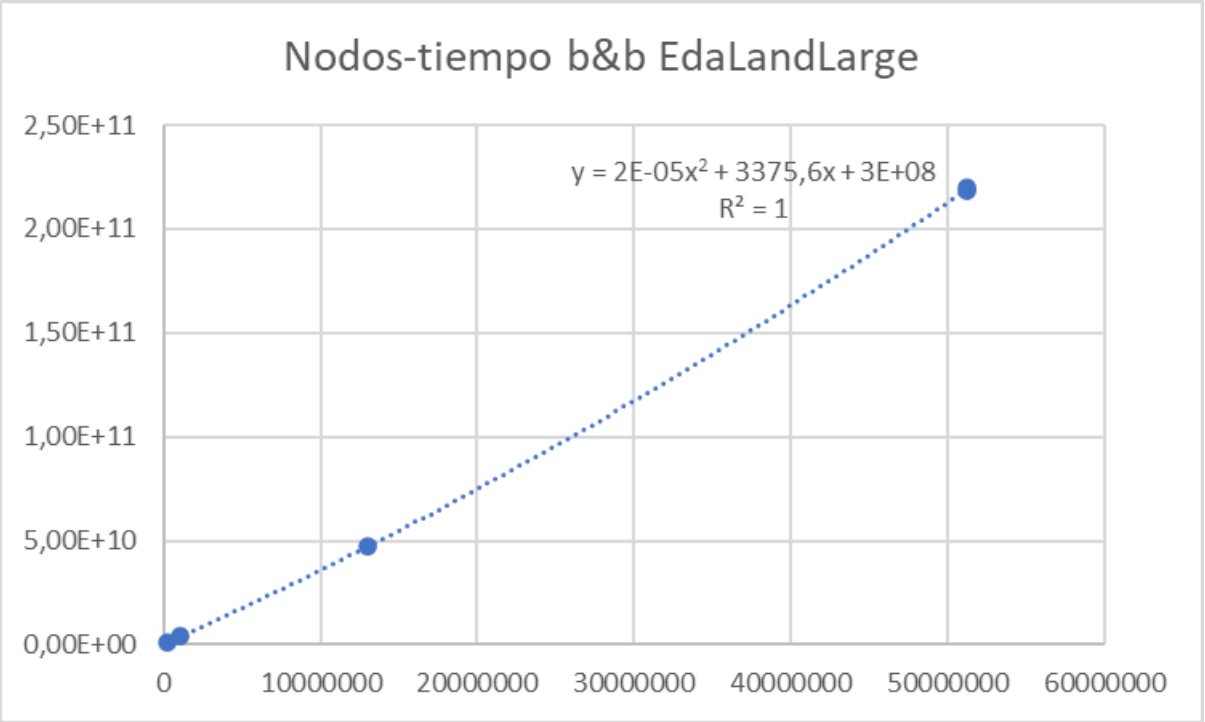


Gráfico EdaLandLarge

Nº nodos visitados	Tiempo	Memoria disponible
50366186	6,1867E10	144MB
51208415	2,1958E11	50MB
51204051	2,1856E11	51MB
295354	1,2684E9	191MB
1017411	3,8022E9	230MB
13023046	4,7236E10	616MB





Conclusión

En conclusión, podemos decir que en el peor de los casos branch and bound puede llegar a ser de orden de backtracking.