rsanchez628@alumno.uned.es

Cuestiones teóricas de la práctica

1.- Análisis del coste computacional del algoritmo.

- Fundir Horizontes

```java
/*Toma dos horizontes y devuelve el horizonte que se genera combinandolos
 * @param: sky1, sky2 Horizontes que se combinaron
 * @return: el horizonte que resulta de combinarlos
 */
private static Sky mergeSky(Sky sky1, Sky sky2) {

    Integer uh = 0, ia = 0, ib = 0, cax = 0, cbx = 0, cay = 0, cby = 0;
    Integer nx, nh;
    List<Point> template = new LinkedList<Point>();
    List<Point> pa = sky1.getPoints();
    List<Point> pb = sky2.getPoints();
    Integer lastPointerIndexA = pa.size()-1;
    Integer lastPointerIndexB = pb.size()-1;

    while(ia≤lastPointerIndexA && ib≤lastPointerIndexB){
        //get Abscisas
        cax = pa.get(ia).getX();
        cbx = pb.get(ib).getX();

        //Abscisas iguales
        if(cax==cbx) {
            nx = cax;
            //get Ordenadas
            cay = pa.get(ia).getY();
            cby = pb.get(ib).getY();
            ia++;
            ib++;
            nh = Math.max(cay, cby);
            if(nh≠uh) {
                uh = nh;
                template.add(new Point(nx,nh));
            }
        //Abscisas distintas
        } else {
            //Abscisa de candidato A menor que abscisa de candidato B
            if(cax<cbx) {
                nx = cax;
                //get Ordenadas
                cay = pa.get(ia).getY();
                nh = Math.max(cay, cby);
                ia++;
                if(nh≠uh) {
                    uh=nh;
                    template.add(new Point(nx,nh));
                }
            }
            //Abscisa de candidato B menor que abscisa de candidato A
            if(cbx<cax) {
                nx = cbx;
                //get Ordenadas
                cby = pb.get(ib).getY();
                nh = Math.max(cay, cby);
                ib++;
                if(nh≠uh) {
                    uh=nh;
                    template.add(new Point(nx,nh));
                }
            }
        }
    }

    //No quedan puntos en uno de ellos

    while(ia≤lastPointerIndexA) {
        template.add(pa.get(ia));
        ia++;
    }
    while(ib≤lastPointerIndexB) {
        template.add(pb.get(ib));
        ib++;
    }

    return new Sky(template);
}
```

Coste lineal O(l + k)

l ≡ número de puntos del primer horizonte que se fusiona

k ≡ número de puntos del segundo horizonte que se fusiona

Tomando

l = 2m

m ≡ número de puntos del primer horizonte que se fusiona

k = 2p

p ≡ número de edificios del segundo horizonte que se fusiona

El coste de la función es: Coste lineal O(2m + 2p) → O(n)

n ≡ número de edificios que se fusionan

Algoritmo

```
// Caso base de la recursión
private static Sky baseCase(List<Building> buildings, boolean trace) {

    List<Point> empty = new LinkedList<Point>();
    Sky clear = new Sky(empty);
    if (buildings.size()==0) {
        return clear;}
    else if (buildings.size()==1) {
        return mergeSky(lift(buildings.get(0)),clear, trace);}
    else {
        return mergeSky(lift(buildings.get(0)),lift(buildings.get(1)),trace);}
}

//función recursiva
private static Sky mergeSkyline(List<Building> buildings, boolean trace) {
    if (buildings.size()<2) {
        return baseCase(buildings, trace);}
    else {
        return mergeSky(mergeSkyline(split(buildings).get(0),trace),mergeSkyline(split(buildings).get(1),trace),trace);}
}
```

El coste del algoritmo depende de cuántas descomposiciones se realicen y del tipo de decrecimiento de las sucesivas llamadas recursivas a los subproblemas.

En este caso, el problema decrece de forma geométrica:

$T(n) = aT(n/b) + cn^k$

a ≡ número de subproblemas en los que se descompone el problema **= 2**

b ≡ factor de reducción del tamaño **= 2**

cn^k ≡ coste de la función de combinación de soluciones parciales **= n → k = 1**

Si **$a = b^k$**; $2 = 2^1$ → $\theta(n^k \text{ Log } n)$ con k = 1 → **$\theta(n \text{ Log } n)$**

Siendo n el número de edificios que compone el horizonte que se calcula.

2.- Discusión de las alternativas al esquema divide y vencerás.

Algoritmos Voraces

Este esquema se emplea con problemas que constan de *n* candidatos y buscan soluciones formadas por un subconjunto de los candidatos, o una secuencia ordenada, de forma que se optimice una función objetivo o se cumpla una restricción.

Si edificios y horizontes se toman como colecciones de puntos el esquema voraz resulta válido para resolver el problema.

Aplicación del esquema voraz en la resolución del problema Skyline:

1. El conjunto de edificios de toma como un conjunto de puntos que marcan el principio y final de cada edificio.
2. Se implementa una cola de prioridad de forma que el punto que se encuentre en primera posición sea siempre el del edificio de mayor altura mayor altura.
3. Se van recorriendo el conjunto de puntos que definen los edificios ordenados según abscisa y añadiendo en la cola de prioridad.
    a. Si un punto es el de comienzo del edificio de mayor tamaño, pasa a la primera posición.
    b. Cuando se añade a la cola de prioridad el punto que marca el final del edificio de más altura, cambia el punto que estaba en la primera posición al ser otro edificio el de mayor altura para esa coordenada x.
4. Cada vez que cambia la altura del punto en primera posición de la cola de prioridad, se tiene un punto que formará parte del subconjunto de puntos solución del horizonte.

El **coste temporal de aplicar esta solución resulta: O(n Log n)**, por lo que no representa beneficio al que se empleó por tener los dos el mismo coste.

Programación dinámica

Técnica que reduce el coste del algoritmo memorizando soluciones parciales que se necesitan para llegar a la solución final.

En el caso del Skyline implicaría abandonar la recursión e ir añadiendo edificios al horizonte de uno en uno.

Cada uno de los horizontes intermedios que se genere es un horizonte válido que servirá para generar el siguiente horizonte con el nuevo edificio que se considera.

Edificios $\{e_1, e_2, e_3, \ldots, e_n\}$	**Horizonte { … }**
• $\{ e_1, e_2\} \rightarrow \{h_1, h_2\}$ • $\{ e_1, e_2, e_3\} \rightarrow \{h_{12}, h_3\}$ • … • $\{e_1, e_2, e_3, \ldots, e_{n-1}\} \rightarrow \{h_{1n-2}, h_{n-1}\}$	• h_{12} • h_{13} • … • h_{1n-1}
$\{e_1, e_2, e_3, \ldots, e_n\} \rightarrow \{h_{1n-1}, h_n\}$	**h_{1n}**

El coste de la función que fusiona los edificios es lineal respecto al número de edificios: O(n)

Se recorre el vector de los n edificios añadiendo el nuevo edificio → al horizonte generado hasta ese momento: $O(n^2)$

Vuelta atrás

El esquema realiza un recorrido en profundidad del grafo implícito de un problema, podando aquellas ramas para las que el algoritmo puede comprobar que no puede alcanzar una solución al problema. El objetivo de la *vuelta atrás* puede ser encontrar la primera solución o todas las soluciones posibles del problema.

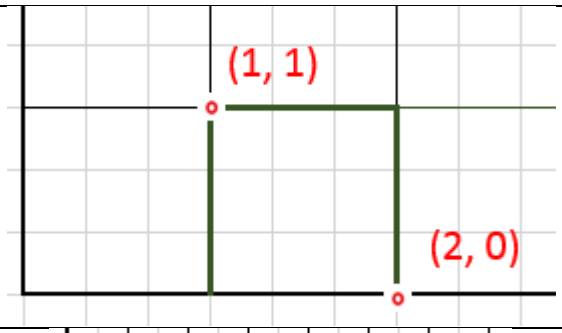
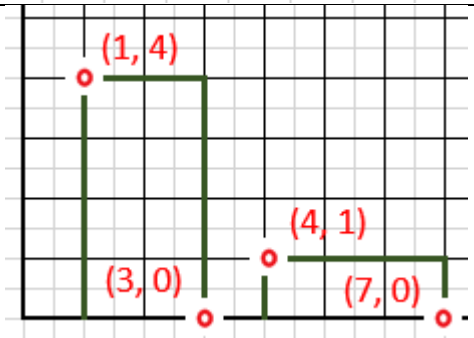
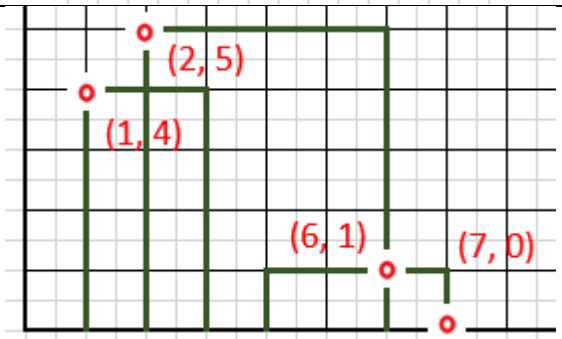
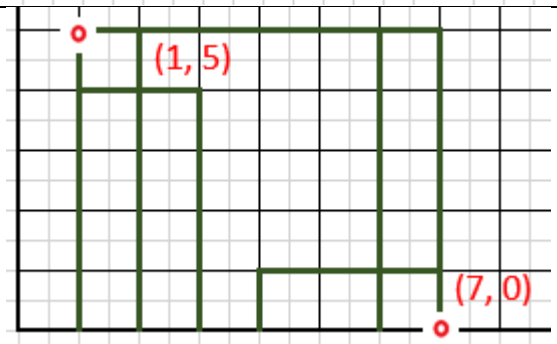
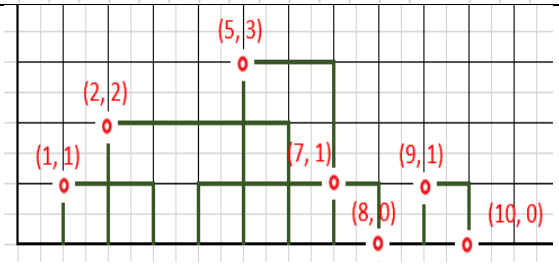
El horizonte que genera una colección de edificios existe y es único por lo que la aplicación de este esquema carecería de sentido.

Ramificación y poda

Es un esquema para explorar un grafo dirigido implícito buscando la solución óptima.

Requiere de una función que se quiera optimizar, lo que no existe en el problema de generar el Skyline.

Ejemplos de ejecución para distintos tamaños del problema

Entrada – Edificios	**Salida – Horizonte**	**Grafico**
Fichero vacío		
• (1, 2, 1)	• (1, 1) • (2, 0)	
• (1, 3, 4) • (4, 7, 1)	• (1, 4) • (3, 0) • (4, 1) • (7, 0)	
• (1, 3, 4) • (4, 7, 1) • (2, 6, 5)	• (1, 4) • (2, 5) • (6, 1) • (7, 0)	
• (1, 3, 4) • (4, 7, 1) • (2, 6, 5) • (1, 7, 5)	• (1, 5) • (7, 0)	
• (1, 3, 1) • (2, 6, 2) • (4, 8, 1) • (5, 7, 3) • (9, 10, 1)	• (1, 1) • (2, 2) • (5, 3) • (7, 1) • (8, 0) • (9, 1) • (10, 0)	

• (1, 3, 3) • (1, 6, 2) • (2, 4, 4) • (6, 7, 3)	• (1, 3) • (2, 4) • (4, 2) • (6, 3) • (7, 0)	Ejemplo extraído del libro base