Løsning af skyline-problemet

Keld Helsgaun
RUC, oktober 1999

Efter at have overvejet problemet en stund er min første indskydelse, at jeg kan opnå en løsning ved at tilføje en bygning til den aktuelle skyline en ad gangen. Jeg skal blot finde en repræsentation af den aktuelle skyline, som muliggør dette.

Jeg vil opfatte en skyline som en liste af rektangler med en venstre x-koordinat, en højde, og en højre x-koordinat. Listen skal være ordnet således, at et rektangels venstre x-koordinat altid er større end eller lig med et eventuelt foregående rektangels højre x-koordinat.

Min erfaring siger mig, at det nok vil være en god ide også at lade rektangler, hvis højde er 0, være repræsenteret i listen. Dermed er der ingen særtilfælde. Fra starten skal skylinen derfor bestå af et enkelt rektangel - med højde 0 og med venstre og højre x-koordinat lig med henholdsvis bygningernes minimale og maksimale x-koordinat.

Herefter er algoritmen simpel. Jeg skal blot for hver bygning løbe den aktuelle skyline igennem og justere denne passende.

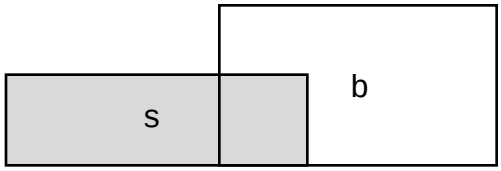
Der opstår en række tilfælde, når en bygning overlapper et rektangel i skylinen, men det skulle nok være til at håndtere.

For at lette programmeringen anvender jeg mig af Java-pakken simset, en pakke til håndtering af tovejslister, som jeg har tidligere udviklet (pakken er tilgængelig fra kursets hjemmeside).

En kommenteret version af det udviklede Java-program ses nedenfor. For at forstå programmet er det en god ide at tegne de situationer, der kan opstå.

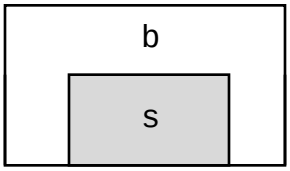
De 4 tilfælde, der hentydes til i kommentarerne er følgende:

Tilfælde 1:

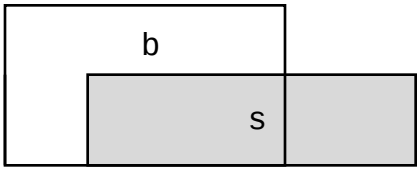


Tilfælde 2:

Tilfælde 3:



Tilfælde 4:



```
import simset.*;
import IO.*;

//* An object of class Rectangle is used to represent a building
//* or an element of the current skyline. L and R denote the
//* left and right x coordinate, respectively. H is the height.
//* Rectangle is a subclass of class Link of the simset package.
//* Thus objects of the class can be kept in a list.

class Rectangle extends Link {
    double L, H, R;

    Rectangle(double l, double h, double r)
      { L = l; H = h; R = r; }
}

public class SkylineProblem {
    public static void main(String args[]) {
        // Create a list of buildings
        Head buildings = new Head();
        new Rectangle( 1, 11,  5).into(buildings);
        new Rectangle( 2,  6,  7).into(buildings);
        new Rectangle( 3, 13,  9).into(buildings);
        new Rectangle(12,  7, 16).into(buildings);
        new Rectangle(14,  3, 25).into(buildings);
        new Rectangle(19, 18, 22).into(buildings);
        new Rectangle(23, 13, 29).into(buildings);
        new Rectangle(24,  4, 28).into(buildings);
        // Find the minimum and maximum x coordinate of the
        // buildings
        double minL = Double.MAX_VALUE, maxR = Double.MIN_VALUE;
        for (Rectangle b = (Rectangle) buildings.first();
             b != null;
             b = (Rectangle) b.suc()) {
            if (b.L < minL) minL = b.L;
            if (b.R > maxR) maxR = b.R;
        }
        // Create the initial skyline
        // (a list of Rectangle objects)
        Head skyline = new Head();
        new Rectangle(minL, 0, maxR).into(skyline);
```

```
// For each building b
for (Rectangle b = (Rectangle) buildings.first();
     b != null;
     b = (Rectangle) b.suc()) {
    // For each rectangle s in the current skyline
    for (Rect s = (Rectangle) skyline.first();
         s != null;
         s = (Rectangle) s.suc()) {
        if (b.H > s.H && s.R > b.L && s.L < b.R) {
            // b is higher than s and its base line
            // overlaps the base line of s.
            // There are 4 cases:
            if (s.L < b.L) {
                if (s.R <= b.R)
                // (1) b overlaps the right part of s.
                    // Let a new skyline rectangle of
                    // height b.H follow s:
                    new Rectangle(b.L, b.H, s.R).
                        follow(s);
                else {
                // (2) s overlaps b totally.
                    // Split s into three rectangles:
                    new Rectangle(b.R, s.H, s.R).
                        follow(s);
                    new Rectangle(b.L, b.H, b.R).
                        follow(s);
                }
                // Adjust the right x coordinate of s.
                s.R = b.L;
            }
            else if (s.R <= b.R)
                // (3) b is "inside" s.
                s.H = b.H;
            else {
                // (4) b overlaps the right part of s.
                // Let a new skyline rectangle of
                // height s.H follow s:
                new Rectangle(b.R, s.H, s.R).follow(s);
                // Adjust the height and right
                // coordinate of s.
                s.H = b.H; s.R = b.R;
            }
            // For efficiency: collapse s with its
            // predecessor if it has the same height:
            Rectangle t = (Rectangle) s.pred();
            if (t != null && t.H == s.H)
              { t.R = s.R; s.out(); s = t; }
        }
    }
}
```

```
        // Print the solution:
        for (Rectangle r = (Rectangle) skyline.first();
             r != null;
             r = (Rectangle) r.suc())
            IO.print(r.L + " " + r.H + " ");
        Rectangle r = (Rectangle) skyline.last();
        if (r != null)
            IO.print(r.R + " " + 0);
        IO.println();
    }
}
```

Programmet er korrekt, men det er ikke særligt effektivt. I det værste tilfælde kræver det O(n) skridt for at placere den n'te by i skylinen. Dermed bliver det totale antal skridt lig med $O(n) + O(n-1) + ... + O(1) = O(n^2)$.

For at forbedre effektiviteten kan del-og-hersk-teknikken benyttes. Hvert problem opdeles i to (næsten) lige store dele, som løses hver for sig, hvorefter de to løsninger samles til én løsning. Dette princip benyttes rekursivt.

I dette problem kan mængden af bygninger opsplittes i to mængder, hver bestående af cirka n/2 bygninger. Derefter bestemmes de to skylines for hver af de to mængder, hvorefter de derved fremkomne skylines flettes til en skyline.

To skylines kan flettes efter samme princip som i det forrige program. De to skylines gennemløbes fra venstre mod højre, idet x-koordinaterne sammenholdes, og højderne justeres, når det er nødvendigt.

Nedenfor ses en udgave af et program, der benytter denne metode. Det kan vises, at køretiden er O(nlogn).

```
import simset.*;
import IO.*;

//* An object of class Rectangle is used to represent a building
//* or an element of the current skyline. L and R denote the
//* left and right x coordinate, respectively.
//* H is the height. Rectangle is a subclass of class Link of
//* the simset package.
//* Thus objects of the class can be kept in a list.

class Rectangle extends Link {
    double L, H, R;

    Rectangle(double l, double h, double r)
      { L = l; H = h; R = r; }
}

//* An object of class RectangleList is used to represent a list
//* of buildings or a the current skyline. RectangleList is a
//* subclass of class Head of the simse package.
//* Thus objects of the class can used as list headers.

class RectangleList extends Head {
    //* split() is used to split this list into two lists of
    //* (almost) equal length.
    //* The first n/2 elements of the list, where n is the
    //* original list length, are moved to a new list.
    //* This new list is the return value of split().
    RectangleList split() {
        RectangleList L = new RectangleList();
        for (int n = cardinal()/2; n > 0; n--) first().into(L);
        return L;
    }

    //* merge(L) is used to merge two skylines.
    //* The skyline represented by "this" is merged by the
    //* skyline represented by L. The result is stored in
    //* "this".
    //* At entry "this" list should not be empty.
    void merge(RectangleList L) {
        Rectangle s1 = (Rectangle) first();
        Rectangle s2 = (Rectangle) L.first();
        // Assure that all rectangles i L are fully contained in
        // the union of rectangles og "this":
        if (s2 != null) {
            if (s2.L < s1.L)
                new Rectangle(s2.L, 0, s1.L).precede(s1);
                s1 = (Rectangle) last();
                s2 = (Rectangle) L.last();
                if (s2.R > s1.R)
                    new Rectangle(s1.R, 0, s2.R).follow(s1);
                s1 = (Rectangle) first();
                s2 = (Rectangle) L.first();

        }
```

```
            // For each rectangle s2 in L:
            while (s2 != null) {
                if (s1.R <= s2.L)
                    // Move to next rectangle in "this" list:
                    s1 = (Rectangle) s1.suc();
                else if (s2.R <= s1.L)
                    // Move to next rectangle in L:
                    s2 = (Rectangle) s2.suc();
                // Otherwise, treat the following 4 cases:
                else if (s2.L <= s1.L) {
                    if (s1.R <= s2.R) {
                    // Case 1: s2 overlaps s1 totally
                        if (s2.H > s1.H) {
                            s1.H = s2.H;
                            Rectangle t = (Rectangle) s1.pred();
                            if (t != null && t.H == s1.H)
                                { t.R = s1.R; s1.out(); s1 = t; }
                        }
                        if (s2.R == s1.R) s2 = (Rectangle) s2.suc();
                        s1 = (Rectangle) s1.suc()
                    }
                    else {
                    // Case 2: s2 overlaps the left part of s1
                        if (s2.H > s1.H) {
                            new Rectangle(s2.R, s1.H, s1.R).
                                follow(s1);
                            s1.R = s2.R;
                            s1.H = s2.H;
                            Rectangle t = (Rectangle) s1.pred();
                            if (t != null && t.H == s1.H)
                                { t.R = s1.R; s1.out(); s1 = t; }
                        }
                        s2 = (Rectangle) s2.suc();
                    }
                }
                else if (s1.R < s2.R) {
                    // Case 3: s1 overlaps the right part of s1
                    if (s2.H > s1.H) {
                        new Rectangle(s2.L, s2.H, s1.R).follow(s1);
                        s1.R = s2.L;
                    }
                    s1 = (Rectangle) s1.suc();
                }
                else {
                    // Case 4: s2 is contained in s1
                    if (s2.H > s1.H) {
                        new Rectangle(s2.R, s1.H, s1.R).follow(s1);
                        new Rectangle(s2.L, s2.H, s2.R).follow(s1);
                        s1.R = s2.L;
                    }
                    s2 = (Rectangle) s2.suc();
                }
            }
        }
    }
```

```
public class SkylineProblem {
    //* Solve the problem by divide-and-conquer
    static RectangleList solve(RectangleList L1) {
        if (L1.first() != L1.last()) {
            // More than one element in L1.
            RectangleList L2 = L1.split();
            solve(L1);
            solve(L2);
            L1.merge(L2);
        }
        return L1;
    }

    public static void main(String args[]) {
        // Create a list of buildings:
        RectangleList buildings = new RectangleList();
        new Rectangle( 1, 11,  5).into(buildings);
        new Rectangle( 2,  6,  7).into(buildings);
        new Rectangle( 3, 13,  9).into(buildings);
        new Rectangle(12,  7, 16).into(buildings);
        new Rectangle(14,  3, 25).into(buildings);
        new Rectangle(19, 18, 22).into(buildings);
        new Rectangle(23, 13, 29).into(buildings);
        new Rectangle(24,  4, 28).into(buildings);
        // Solve the problem:
        RectangleList skyline = solve(buildings);
        // Print the solution:
        for (Rectangle r = (Rectangle) skyline.first();
             r != null;
             r = (Rectangle) r.suc())
            IO.print(r.L + " " + r.H + " ");
        Rectangle r = (Rectangle) skyline.last();
        if (r != null)
            IO.print(r.R + " " + 0);
        IO.println();
    }
}
```