Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

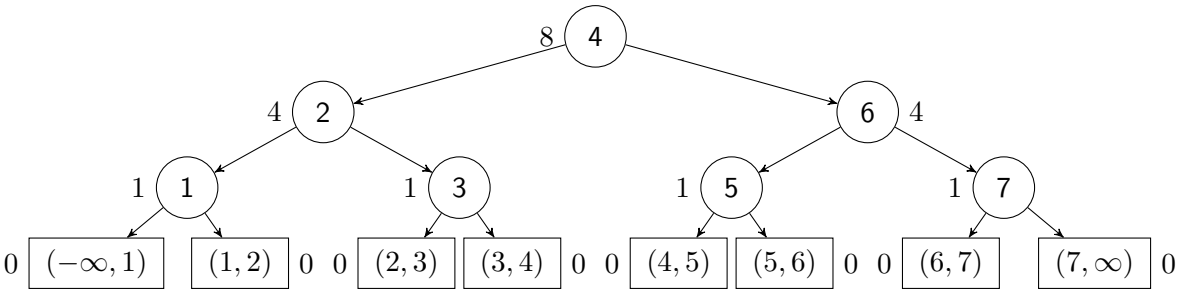
27. April 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 8 FS 16

Lösung 8.1 *Optimale Suchbäume und Splay-Bäume.*

Der optimale Suchbaum für die gegebenen Schlüssel und Häufigkeiten ist unten dargestellt. Neben jedem Knoten steht die entsprechende Zugriffshäufigkeit.

Die Anzahl der Vergleiche ändert sich in einem optimalen Suchbaum nicht mit der Reihenfolge der Abfragen. Für eine Abfrage von Schlüssel 4 ist beispielsweise lediglich ein Vergleich notwendig und für eine Abfrage von Schlüssel 3 sind drei Vergleiche nötig (unabhängig davon, was davor und danach gesucht wird). Für die 20 Abfragen fallen also $8 \cdot 1 + 2 \cdot 4 \cdot 2 + 4 \cdot 1 \cdot 3 = 36$ Vergleiche an.

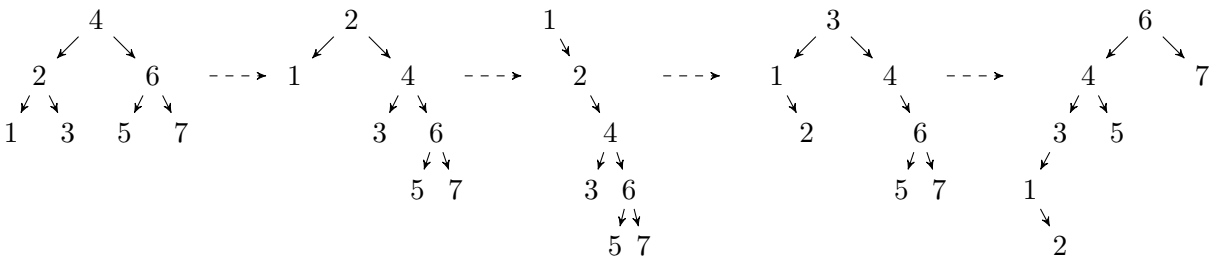


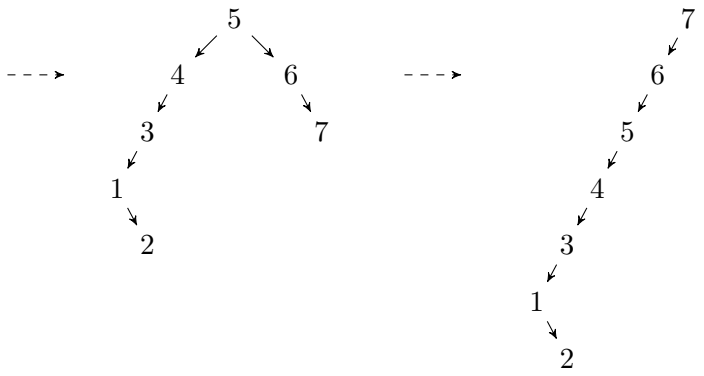
In einem Splay-Baum sind unter Umständen weniger Vergleiche notwendig, falls derselbe Schlüssel mehrmals nacheinander abgefragt wird. Nach der ersten Abfrage ist der Schlüssel nämlich in der Wurzel des Baumes gespeichert. Für weitere Abfragen desselben Schlüssel fällt dann nur noch je ein Vergleich an.

Wir wählen also eine Reihenfolge von Abfragen, bei der gleiche Schlüssel möglichst nacheinander abgefragt werden. Wir betrachten die Reihenfolge

$$\underbrace{4, \ldots, 4}_{8mal}, \underbrace{2, \ldots, 2}_{4mal}, 1, 3, \underbrace{6, \ldots, 6}_{4mal}, 5, 7.$$

Insgesamt fallen für diese Reihenfolge nur 31 Vergleiche im Splay-Baum an. Im Folgenden wird dargestellt, wie sich der Splay-Baum dynamisch anpasst.





Lösung 8.2 *Tiefensuche und Breitensuche.*

a) Tiefensuche: $A, B, C, I, F, D, E, H, G$; Breitensuche: $A, B, C, D, E, H, I, F, G$

b) Sowohl Breiten- als auch Tiefensuche müssen für jeden Knoten einmal alle Nachbarn besuchen. Ist der Graph durch Adjazenzlisten gegeben, so muss die Liste der Nachbarn für jeden Knoten abgearbeitet werden. Die Laufzeit ist in diesem Fall $\mathcal{O}(n+m)$, also für zusammenhängende Graphen $\mathcal{O}(m)$ (wie üblich bezeichnen wir die Anzahl Knoten mit n und die Anzahl Kanten mit m).

Wenn allerdings nur eine Adjazenzmatrix gespeichert ist, kann man die Nachbarn eines Knotens nur finden, indem man alle Einträge der entsprechenden Zeile durchgeht. Da dies für jeden Knoten getan werden muss, ist die Laufzeit $\Theta(n^2)$.

Bei einem sehr dichten Graphen mit $m \in \Theta(n^2)$ ist die asymptotische Laufzeit gleich. Bei «dünnen» Graphen, beispielsweise mit $m \in \mathcal{O}(n)$, führt die Verwendung einer Adjazenzmatrix allerdings zu einer erheblich schlechteren asymptotischen Laufzeit.

Lösung 8.3 *Dynamische Programmierung: Optimale Platzierung von Windrädern.*

a) Schon für $n = 3$ Windräder kann ein solches Beispiel konstruiert werden, z.B. mit den möglichen Positionen $d_1 = 0$, $d_2 = 1$, $d_3 = 2$ und einem Mindestabstand von $D = 2$. Ausserdem definieren wir die erzielten Energieeinheiten als $e_1 = 2$, $e_2 = 3$ und $e_3 = 2$. Die beschriebene Greedy-Strategie platziert ein Windrad auf der Position 2. Da der jeweilige Abstand zu den Positionen 1 und 3 kleiner als D ist, können keine weiteren Windräder platziert werden und die Energieausbeute beträgt genau 3. Optimal wäre aber die Wahl der Positionen 1 und 3 mit einer Energieausbeute von $2 + 2 = 4$.

b) *Definition der DP-Tabelle*: Wir verwenden eine eindimensionale (!) Tabelle T mit n Einträgen. Der Eintrag $T[k]$ beschreibt die maximal produzierbare Energie, wenn Windräder nur auf den Positionen $1, ..., k$ platziert werden können.

Berechnung eines Eintrags: Ist $d_k < D$ für eine Position $k \in \{1, ..., n\}$, dann wird der Mindestabstand zwischen der k-ten und der erstmöglichen Position nicht eingehalten, und es kann maximal eine Position aus den ersten k möglichen gewählt werden. Die maximale Energie wird dann produziert, wenn eine Position aus $\{1, ..., k\}$ mit maximaler Energieausbeute gewählt wird. Wir setzen

$$T[1] := e_1 \text{ und } T[k] := \max\{T[k-1], e_k\} \text{ für alle } k > 1 \text{ mit } d_k < D, \qquad (1)$$

und mit dieser Definition ist $T[k] = \max\{e_1, ..., e_k\}$ für alle k mit $d_k < D$.

Ist $d_k \geq D$, dann gibt es zwei Möglichkeiten:

- Es wird ein Windrad auf Position k platziert. Wenn wir

$$v_k := \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\} \tag{2}$$

setzen, dann ist v_k der Index der Position links von k, die am nächsten an k liegt und den Mindestabstand D einhält. Auf den Positionen k' mit $v_k < k' < k$ können also *keine* Windräder platziert werden, da der Mindestabstand zu k nicht eingehalten wird.

Die maximal produzierbare Energie mit den Positionen $1, ..., k$ besteht nun aus der Summe von e_k (Energieausbeute der Position k) und der maximal produzierbaren Energie, wenn nur Positionen aus $\{1, ..., v_k\}$ verwendet werden.

- Es wird kein Windrad auf Position k platziert. In diesem Fall ist die mit den Positionen $1, ..., k$ maximal produzierbare Energie genauso gross wie die maximal produzierbare Energie mit den Positionen $1, ..., k-1$.

Für alle k mit $d_k \geq D$ setzen wir also

$$v_k := \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\} \tag{3}$$
$$T[k] := \max\{T[k-1], e_k + T[v_k]\}. \tag{4}$$

Berechnungsreihenfolge: Da ein Tabelleneintrag nur von Einträgen mit kleinerem Index abhängt, können die Einträge $T[k]$ für aufsteigendes k berechnet werden. Analog können die Werte v_k für aufsteigendes k berechnet werden.

Auslesen der Lösung: Die Lösung steht am Ende im Eintrag $T[n]$.

c) Eine sorgfältige Implementierung des obigen Verfahrens benötigt nur Zeit $\Theta(n)$, wenn v_k nicht in jedem Schritt naiv berechnet wird. Es ist klar, dass v_k nur für k mit $d_k \geq D$ berechnet werden muss. Für das erste solche k kann v_k wie in obiger Formel berechnet werden. Für alle weiteren k wissen wir, dass v_{k-1} bereits berechnet wurde, und es gilt

$$\begin{aligned} v_k &= \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\} \\ &= \max\{i \in \{v_{k-1}, ..., k-1\} : d_i \leq d_k - D\}. \end{aligned} \tag{5}$$

Zur Berechnung von v_k aus v_{k-1} kann also initial $v_k = v_{k-1}$ gesetzt und dieser Index so lange um 1 erhöht werden, bis $d_{v_k+1} > d_k - D$ gilt. Auf diese Art können *alle* v_k in Zeit $\Theta(n)$ berechnet werden. Da die Laufzeit zur Berechnung eines einzelnen Eintrags $T[k]$ nur konstant ist, beträgt die Gesamtlaufzeit $\Theta(n)$.

d) Wir füllen zunächst die gesamte Tabelle aus und speichern auch alle Werte von v_k. Eine optimale Positionierung von Windrädern kann nun durch Rückverfolgung der Werte in der Tabelle wie folgt erreicht werden. Wir setzen initial $k = n$ und wiederholen die folgenden Schritte, solange $k \geq 1$ ist.

- Falls $d_k \geq D$ und $e_k + T[v_k] \geq T[k-1]$ ist, dann gib die Position k aus und setze $k := v_k$.
- Falls $d_k \geq D$ und $e_k + T[v_k] < T[k-1]$ ist, dann wird mehr Energie produziert, wenn die Position k *nicht* benutzt wird. In diesem Fall wird nichts ausgegeben, und $k := k - 1$ gesetzt.

- Falls $d_k < D$ ist, dann kann nur ein einziges Windrad auf den Positionen $1, ..., k$ platziert werden. In diesem Fall wird die Position $i \in \{1, ..., k\}$ mit maximaler Energieausbeute e_i ausgegeben, und der Algorithmus terminiert.

Die Laufzeit dieses Verfahrens ist weiterhin $\Theta(n)$.

Lösung 8.4 *Zyklen finden (**Programmieraufgabe**).*

In dieser Aufgabe soll entschieden werden, ob ein gegebener Graph $G = (V, E)$ einen Zyklus enthält. Sei $V = \{0, 1, \ldots, n-1\}$ die Menge der Knoten. Der Graph ist in Adjazenzlistendarstellung gegeben und wird in der Code-Vorlage in einem Array `graph` von ArrayLists gespeichert. Der Eintrag `graph[i]` ist also eine ArrayList mit allen Knoten, die Nachbarn des Knoten `i` sind.

Um einen Zyklus in G zu finden, genügt es eine Tiefensuche bei Knoten 0 zu starten. Wir merken uns für jeden Knoten, ob wir ihn bereits besucht haben. Sobald wir einen Knoten v besuchen, markieren wir ihn zuerst als besucht und fahren dann mit der Tiefensuche für jeden Nachbarn des Knoten fort, wobei wir denjenigen Nachbarn, von dem aus wir zu v gekommen sind, ignorieren. Der Graph hat einen Zyklus, falls wir bei der Tiefensuche auf einen bereits besuchten Knoten treffen.

Im folgenden sind eine rekursive und eine nicht-rekursive Lösung gegeben.

Rekursive Lösung

```
class Main {
    boolean[] visited;
    ArrayList<Integer>[] graph;

    //return true iff there is a cycle in graph.
    boolean dfs_cycle() {
        //A boolean array that holds if a vertex has been visited
        visited = new boolean[graph.length];
        Arrays.fill(visited, false);

        return dfs_rec(0,0);
    }

    //recursive function. Returns true iff there is a cycle.
    //v is the current vertex on the DFS
    //u is the previous vertex, i.e. the vertex from which v was called.
    boolean dfs_rec(int v, int u) {
        visited[v] = true;

        for (int w : graph[v]) {
            if(!visited[w]) {
                //if the recursive call returns true then we want it to return true
                //but if the recursive call returns false then we want the for loop to
                    continue
                if (dfs_rec(w,v)) return true;
            }
            //if you see a visited vertex that is not the previous one then there is a
                cycle
            else if (w != u) {
                return true;
            }
        }
        return false;
    }
    ...
```

Nicht-Rekursive Lösung

Diese Lösung verwendet einen Stack und vermeidet dadurch eine Rekursion.

```
class Main {
    boolean[] visited;
    ArrayList<Integer>[] graph;

    //return true iff there is a cycle in graph.
    boolean dfs_cycle() {
        int n = graph.length;

        //A boolean array that holds if a vertex has been visited
        visited = new boolean[n];
        Arrays.fill(visited, false);

        Stack<Integer> stack = new Stack();
        stack.push(0); //0 is treated as the root

        //An array that holds which is the previous vertex
        //w.r.t. dfs for each vertex
        int[] previous = new int[n];
        Arrays.fill(previous, -1);

        while (!stack.empty()) {
            int v = stack.pop();
            visited[v] = true;

            for (int w : graph[v]) {
                if (visited[w]) {
                    //if w is visited and it's not the
                    //previous of v then we found a cycle
                    if (w != previous[v]) return true;
                }
                else {
                    stack.push(w);
                    //we set the previous of w to be v
                    //only if the previous of v is not w
                    if (previous[v]!=w) previous[w] = v;
                }
            }
        }
        //if this point is reached the whole graph has
        //been explored and no cycles have been found
        return false;
    }
    ...
```

Die Funktion Main

```
    ...
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int test = in.nextInt();

        for (int t=0; t<test; t++) {
            Main main = new Main();
            int n = in.nextInt();
            in.nextLine(); //read the extra new line character
            //The above is because nextInt() ignores the \n character
            // that comes after the integer. nextLine() reads and
            //in this case discards this character.

            main.graph = new ArrayList[n];
            for (int i = 0; i < n; i++)
                main.graph[i] = new ArrayList<Integer>();
```

```
            for (int j=0; j<n; j++) {
                String adj = in.nextLine();

                //The split() functions splits the string
                //into an array of strings according the
                // split character " "  (space)
                for (String s : adj.split(" ")) {
                    //tries to parse each string to an integer
                    //and adds this integer to the adjacency list
                    //of the right vertex
                    try {main.graph[j].add(Integer.parseInt(s));}
                    catch (java.lang.NumberFormatException nfe) {
                        //do nothing, just ignore garbage characters
                        //for example the new line character \n
                    }
                }
            }

            if (main.dfs_cycle()) System.out.println("y");
            else System.out.println("n");
        }
    }
}
```