

## Handout 13

Sebastian Millius, Sandro Feuz, Daniel Graf

**Thema:** Geometrische Algorithmen: Konvexe Hülle, Scanline, Intervall-Baum, Segment-Baum, Range Trees, DP auf Bäumen: Minimum Vertex Cover

**Referenz:** Widmayer, Kapitel 7, Cormen et. al, Kapitel 33

### Aus aktuellem Anlass

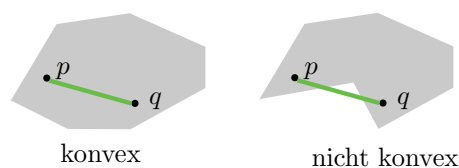
Prof. Dr. Robert Endre Tarjan, der Erfinder der Fibonacci-Heaps, kommt in zwei Woche an die ETH und hält einen Vortrag zum Thema "Concurrent Disjoint Set Union": Freitag 17. Juni um 12.15 Uhr im CAB G 11 (<http://goo.gl/ueXmyG>)

### Links

- Belfrage Slides: [http://www.belfrage.net/eth/d&a/pdf/uebung6\\_h.pdf](http://www.belfrage.net/eth/d&a/pdf/uebung6_h.pdf) (<http://goo.gl/Pvjjei>)
- Prof. Erickson Lecture Notes: <http://goo.gl/OZYSK>
- Konvexe Hülle: Visualisierung verschiedener Algorithmen  
<http://www.cs.princeton.edu/courses/archive/spring09/cos226/demo/ah/ConvexHull.html> (<http://goo.gl/QYkPY>)
- Eine ausführliche Darstellung findet sich in *de Berg, M., Cheong, O., van Kreveld and M., Overmars, M.: Computational Geometry: Algorithms and Applications*. Ist (via VPN) auf Springerlink frei verfügbar  
<http://springerlink.com/content/k18243/> (<http://goo.gl/bMGLr>)  
Belfrage eine Übersicht/Zusammenfassung der Darstellung über Range Trees zusammengestellt: <http://www.belfrage.net/eth/d&a/pdf/rangetrees.pdf> (<http://goo.gl/UKcsI>)
- Konkrete Implementationen von Geometrischen Algorithmen in C finden sich beispielsweise in *O'Rourke, J.: Computational Geometry in C*  
<http://maven.smith.edu/~orourke/books/compgeom.html> (<http://goo.gl/6KCMM>)
- Belfrage Slides: [http://www.belfrage.net/eth/d&a/pdf/uebung6\\_h.pdf](http://www.belfrage.net/eth/d&a/pdf/uebung6_h.pdf) (<http://goo.gl/Pvjjei>)
- Prof. Erickson Lecture Notes: <http://goo.gl/OZYSK>
- Scanline: Demo-Applet  
<http://www.cs.princeton.edu/courses/archive/spring09/cos226/demo/gishur/> (<http://goo.gl/tSKkU>)

### Konvexe Hülle

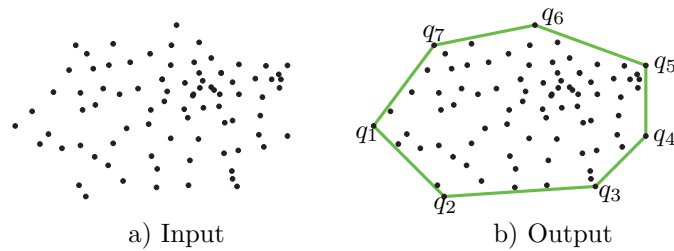
Eine Menge heisst *konvex* wenn für alle Paare von Punkten  $p, q$  ( $p \neq q$ ) in der Menge die Verbindungslinie vollständig in der Menge enthalten ist.



## Das Konvexe Hülle Problem in der Ebene ( $\mathbb{R}^2$ )

**Konvexe Hülle** Gegeben sei eine Menge  $\mathcal{P}$  von Punkten in der Ebene. Die konvexe Hülle von  $\mathcal{P}$  ist die kleinste konvexe Menge, in der  $\mathcal{P}$  enthalten ist

BEOBACHTUNG: Die Kanten der konvexen Hülle einer Menge von Punkten  $\mathcal{P}$  verbindet Punkte aus  $\mathcal{P}$ !



**Stop and Think!** Gegeben eine Menge von Punkten, finde möglichst effizient die beiden Punkte, die den grössten Abstand voneinander haben

Für das Problem der Bestimmung der konvexen Hülle wurde eine Vielzahl von verschiedenen Algorithmen vorgeschlagen.

### Jarvis March (Gift Wrapping)

Der natürlichste Algorithmus zur Bestimmung der konvexen Hülle, der der Art und Weise entspricht, wie ein Mensch die konvexe Hülle einer Punktmenge zeichnen würde, ist ein systematisches Verfahren des "Einwickelns" der Punktmenge. Man beginnt bei irgendeinem Punkt, der garantiert zur konvexen Hülle gehört (etwa bei dem Punkt mit der kleinsten  $x$ -Koordinate). Beginne die Punktmenge "einzuwickeln" von diesem Punkt aus, finde sukzessive den nächsten Punkt auf der Hülle als denjenigen, der am rechtesten von dem vorherigen Punkt ist.

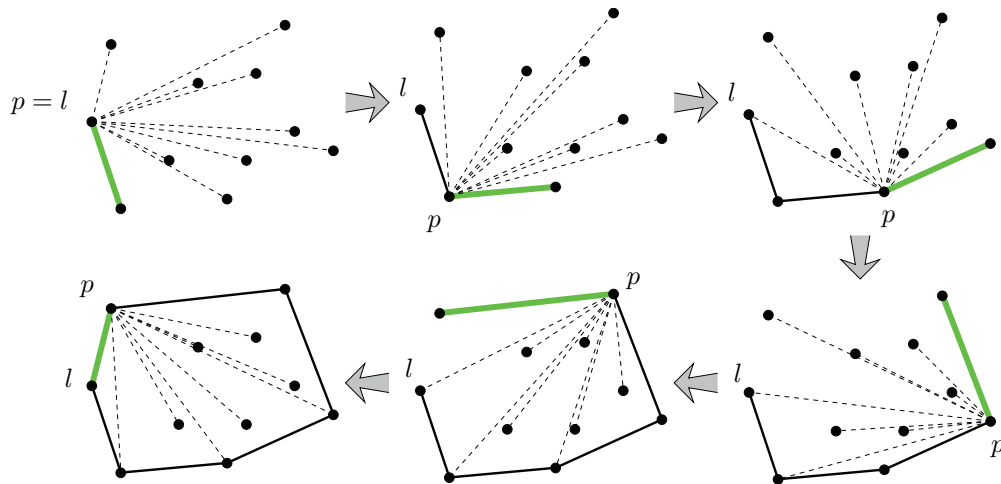


Abb.: Beispieldurchlauf von Jarvis March

Man kann den Algorithmus wie folgt implementieren, so dass neben dem Vergleich von  $x$ -Koordinaten nur eine zusätzliche geometrische Operation benötigt wird: ein *orientation test* (vgl. Anmerkungen): Es sei  $\text{CW}(p, q, r)$  für drei Punkte  $p, q, r \in \mathbb{R}^2$ , das Prädikat das TRUE ist wenn  $r$  (strikt) rechts des (orientierten) Liniensegments  $\overline{pq}$  ist.

```
// Eingabe: Menge von Punkten in der Ebene (P)
// Ausgabe: Konvexe Huelle (H)
void wrap (vector<Point2>& P, vector<Point2>& H) {
    // Finde den Startpunkt
    int left = 0;
    for (int i=1; i < P.size(); i++)
        if (P[i] < P[left]) left = i;
```

```

// Einwickeln
int p = left; // beginne mit linkestem Punkt
int next = (left+1)%P.size(); // irgendein anderer Punkt
do {
    H.push_back (P[p]);

    // Finde rechtesten Punkt
    for (int i=0; i < P.size(); i++)
        if (cw (P[p], P[next], P[i]))
            next = i;

    p = next; next = left;
} while (p != left);
}

```

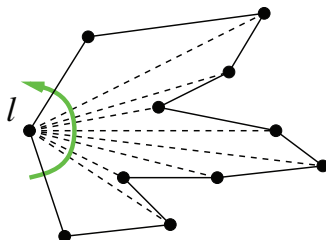
Für jeden der  $h$  Punkte auf der konvexen Hülle hat der Algorithmus einen Aufwand von  $\mathcal{O}(n)$ , das heisst die Gesamtlaufzeit ist  $\mathcal{O}(nh)$ . Der Algorithmus ist also *Output-sensitiv*: je kleiner der Output desto schneller der Algorithmus. Im Worst Case, wenn alle Punkte auf der Hülle liegen, also  $\mathcal{O}(n^2)$ .

## Graham Scan (Successive Local Repair)

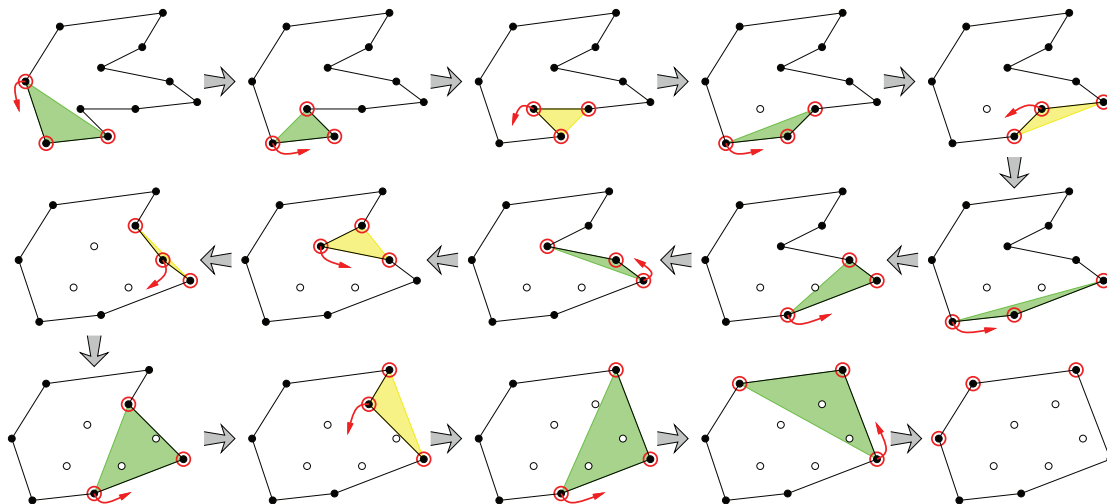
Folgendes Verfahren (Durchsuchen nach Graham), das 1972 von R.L. Graham entwickelt wurde ist interessant, weil der grösste Teil der erforderlichen Berechnungen das Sortieren betrifft:

### Graham scan

1. Wähle Punkt  $p$  mit der kleinsten  $x$ -Koordinate.
2. Sortiere Punkte nach Winkel bezüglich  $p$  um ein simples Polygon zu erhalten (radiale Sortierung)
3. Betrachte die Punkte in dieser Ordnung und verwirfe diejenigen die eine Drehung im Uhrzeigersinn provozieren würden



**Abb.:** Durch die radiale Sortierung betrachtet der Algorithmus die Punkte (implizit) entlang eines *simplem* Polygons



## Abb.: Die "Drei Groschen-Oper"

Eine C++ Implementation davon

---

```
void graham (vector<Point2>& P, vector<Point2>& H) {
    // Finde den Startpunkt
    int left=0;
    for (int i=1; i < P.size(); i++)
        if (P[i] < P[left]) left = i;

    // Sortiere radial um P[left]
    sort (P.begin(), P.end(), cmp_angle (P[left]));

    H.push_back (P[0]);
    H.push_back (P[1]);

    for (int i=2; i < P.size(); ) {
        if (H.size()>=2 && cw (H[H.size()-2], H[H.size()-1], P[i]))
            H.pop_back();
        else {
            H.push_back (P[i]);
            ++i;
        }
    }
}
```

---

## Scanline / Sweepline

- Sortiere Punkte
- Links-Rechts-Durchgang: Untere Hülle
- Rechts-Links-Durchgang: Obere Hülle
- Verwerfe jeweils diejenigen die eine Drehung im Uhrzeigersinn provozieren würden

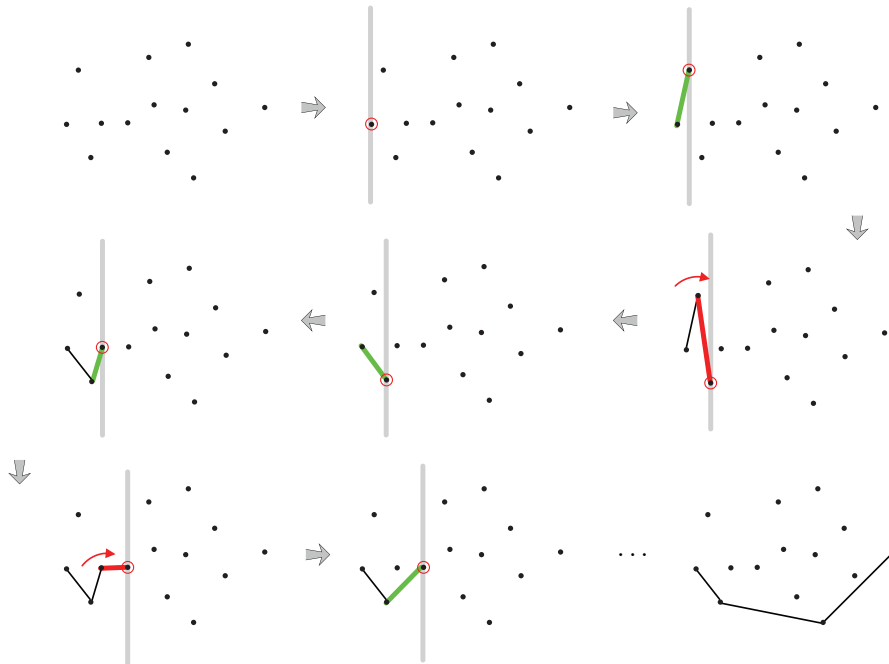


Abb.: Bestimmung der unteren Hülle

---

```
// Eingabe: Menge von Punkten in der Ebene (P)
// Ausgabe: Konvexe Huelle (H)
```

```

void scan (vector<Point2>& P, vector<Point2>& H)
{
    if (P.size() <= 2) { H=P; return; }

    sort (P.begin(), P.end());
    H.push_back (P[0]); //linkester Punkt ist auf der Huelle
    int h = 0;

    // Untere Huelle (links nach rechts Scan)
    for (int i=1; i<P.size(); i++) {
        while (h>0 && cw (H[h-1], H[h], P[i])) { H.pop_back(); --h; }
        H.push_back (P[i]); ++h;
    }

    // Obere Huelle (rechts nach links Scan)
    for (int i=P.size()-2; i>=0; --i) {
        while (cw (H[h-1], H[h], P[i])) { H.pop_back(); --h; }
        H.push_back (P[i]); ++h;
    }
}

```

---

Die Laufzeit des Scanline Algorithmus ist bestimmt durch das Sortieren und zwei lineare Durchgänge, also  $\mathcal{O}(n \log n)$ .

## Laufzeiten

$n$ : Anzahl Punkte,  $h$ : Anzahl Punkte auf der Hülle

- Jarvi's March  $\mathcal{O}(nh)$
- Graham Scan  $\mathcal{O}(n \log n)$
- Scanline  $\mathcal{O}(n \log n)$
- Chan's Algorithm  $\mathcal{O}(n \log h)$

Ein schnellerer Algorithmus (Melkman) ist möglich  $\mathcal{O}(n)$  falls die Punkte bereits als simples Polygon gegeben sind. (Erster Teil von Graham Scan entfällt (Sortierung))

Beachte, dass die untere Schranke von  $\Omega(n \log n)$  eine Worst-Case Schranke ist. Zum Beispiel schlägt Jarvis March diese Schranke für  $h = o(\log n)$ , also wenn nur sehr wenige Punkte auf der konvexen Hülle sind.

Die Frage blieb, ob gleichzeitig Outputsensitivität (wie Jarvis March) und optimale Worst Case Performance (wie Graham Scan, Scanline) erreicht werden kann. In der Tat erreicht dies Chan's Algorithmus diese Laufzeit in dem er geschickt die Vorteile von Jarvis March und Graham Scan kombiniert.

## Anmerkungen

Die verwendeten Hilfsfunktionen in den drei Implementationen können angegeben werden als

---

```

#define EPS 1e-9

struct Point2 {
    double x, y;
    Point2 (double _x=0, double _y=0) : x(_x), y(_y) {}
};

#define DET(a,b,c,d) ((a)*(d)-(b)*(c))
#define SGN(x) ((x)<=-EPS ? -1 : (x)>EPS ? 1 : 0)
#define COLLINEAR(a,b,c) (cw(a,b,c)==0)

double squaredDistance (Point2 p0, Point2 p1)

```

```

{
  double dx = p1.x-p0.x;
  double dy = p1.y-p0.y;
  return dx*dx+dy*dy;
}

// -1: cw, 0: collinear, 1: ccw
char orientation (Point2 p0, Point2 p1, Point2 p2)
{
  return SGN(DET(p1.x-p0.x,p2.x-p0.x,p1.y-p0.y,p2.y-p0.y));
}

bool cw (Point2 p0, Point2 p1, Point2 p2)
{
  int o = orientation (p0, p1, p2);
  if (o == 0) // collinear
    return (squaredDistance (p0, p1) < squaredDistance (p0, p2));
  else
    return (o == -1);
}
}



---


struct cmp_angle {
  Point2 p0;
  cmp_angle (Point2 p) { p0=p; }
  bool operator () (const Point2& a, const Point2& b) {
    return cw (p0, b, a);
  }
};

bool operator< (const Point2& a, const Point2& b) {
  if (fabs (a.x - b.x) < EPS) return (a.y<b.y);
  return (a.x<b.x);
}
}

```

---

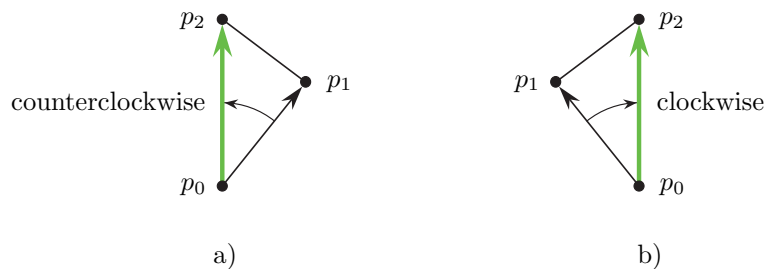
## Degenerierte Fälle

Folgende degenerierte Fälle können vorkommen:

- *Startpunkt*: Mehrere Punkte haben die kleinste  $x$ -Koordinate: Wähle lexikographische Ordnung:  $(p_x, p_y) < (q_x, q_y) \iff p_x < q_x \vee p_x = q_x \wedge p_y < q_y$ .
- Drei oder mehrere Punkte sind kollinear: Wähle Punkt der am weitesten ist (von denen die am rechtesten sind)

## Left/Right Turn - Orientation Test

Liegt ein Punkt  $p_2$  links von dem (gerichteten) Liniensegment  $\overrightarrow{p_0 p_1}$  wenn die Punkte  $p_0, p_1, p_2$  im Uhrzeigersinn durchlaufen werden:



Mit Hilfe der Determinante (think about it!) kann bestimmt werden, in welche Richtung drei Punkte  $p_0, p_1, p_2$  einen Knick machen. Wir überprüfen ob das (gerichtete) Liniensegment  $\overrightarrow{p_0 p_1}$  im oder gegen den Uhrzeigersinn relativ zu  $\overrightarrow{p_0 p_2}$  liegt. **a)** Wenn gegen den Uhrzeigersinn, so

machen die drei Punkte einen Knick nach links. b) Ansonsten einen nach rechts.

Deshalb können wir die (relative) Orientierung dreier Punkte angeben als

---

```
// -1: cw, 0: collinear, 1: ccw
char orientation (Point2 p0, Point2 p1, Point2 p2)
{
    return SGN(DET(p1.x-p0.x, p2.x-p0.x, p1.y-p0.y, p2.y-p0.y));
}
```

---

### Winkelvergleich

Falls  $dx$  und  $dy$  die Abstände von dem als Anker verwendeten Punkt zu einem anderen Punkt längs der  $x$ -Achse bzw.  $y$ -Achse bezeichnen, so ist der benötigte Winkel  $\tan^{-1} \frac{dy}{dx}$ . Obwohl der Arkustangens (oder `atan2`) meistens eine Standardfunktion ist, ist er gewöhnlich langsam und führt zu wenigstens zwei zusätzlichen Bedingungen, die überprüft werden müssen: ob  $dx$  gleich 0 ist und in welchem Quadrant der Punkt liegt.

Da der Winkel nur für das Sortieren (Vergleichen) benutzt wird, ist es sinnvoll, eine Funktion zu benutzen die sich wesentlich leichter berechnen lässt, jedoch die gleichen ordnenden Eigenschaften hat wie der Arkustangens (so dass man beim Sortieren das gleiche Ergebnis erhält).

Sedgewick schlägt folgende Funktion vor:  $\frac{dy}{dy+dx}$ . Ein Test für Ausnahmbedingungen ist trotzdem notwendig, doch einfacher. Die folgende Funktion gibt einen Wert zwischen 0 und 4 zurück (multipliziere mit 90 einen Wert zwischen 0 und 360 zu erhalten) der *nicht* gleich dem Winkel ist, den  $p_0$  und  $p_1$  mit der Waagrechten bilden, aber die gleichen Ordnungseigenschaften wie dieser Winkel besitzt.

Sedgewick Theta Function

---

```
double theta (Point2 p0, Point2 p1)
{
    double dx = p1.x-p0.x, ax = abs (dx);
    double dy = p1.y-p0.y, ay = abs (dy);

    double t = (ax+ay == 0) ? 0 : dy/(ax+ay);
    if (dx < 0) t = 2-t; else if (dy < 0) t = 4+t;

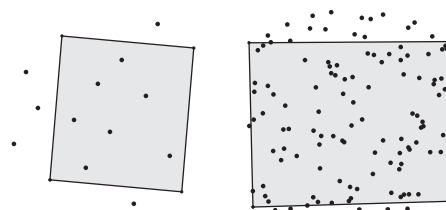
    return t;
}
```

---

Eine *einfachere und elegantere* Möglichkeit ist es, wieder die CW Funktion zu verwenden! Wenn wir radial um einen Punkt  $p_0$  sortieren, dann ist  $a <_{\text{rad}} b$  genau dann wenn  $\text{CW}(p_0, b, a)$  (d.h. wenn  $a$  rechts von  $\overrightarrow{p_0 b}$  liegt, again, think about it!). Dies wird in der obigen Graham-Scan Implementation verwendet.

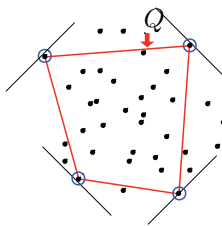
**Stop and Think:** Wie kann die CW Funktion verwendet werden, um zu entscheiden, ob sich zwei Liniensegmente  $\overline{ab}$  und  $\overline{cd}$  schneiden?

### Interior Elimination



Dies ist ein Verfahren, um schnell viele Punkte auszuschliessen, die nicht auf der konvexen Hülle liegen können. Idee: Wähle 4 Punkte und entferne alle Punkte, die im Inneren dieser vier Punkte liegen.

Gute Wahl: Wähle das Quadrilateral  $Q$ :  $\min(x + y), \max(x + y), \min(x - y), \max(x - y)$



**Stop and Think:** Wie kann die cw Funktion verwendet werden, um zu entscheiden, ob sich ein Punkt in einem konvexen Polygon befindet?

## Scanline

Viele geometrische Probleme können mittels *Scanline* Verfahren gelöst werden. Die Idee ist, dass man (Halte-)Punkte der zu bearbeitende Daten in einer Dimension sortiert und mit einer Scanline in Richtung dieser Dimension über den gesamten Raum fährt. Dabei wird in einer Datenstruktur immer ein bestimmter Zustand der momentanen Situation um die Scanline herum gespeichert. Die Haltepunkte sind auch in einer Datenstruktur gespeichert (sortiertes Array, min-Heap, ...) und definieren die Stellen an welchen die Scanline anhalten muss, da sich dort etwas an der Situation ändert. An jedem Haltepunkt wird diese Änderung bearbeitet und die Scanline-Datenstruktur aktualisiert. Eventuell werden auch neue Haltepunkte gefunden und der Haltepunkte-Datenstruktur hinzugefügt. Während dem verschieben der Scanline wird auch die Lösung des Problems stückweise aufgebaut.

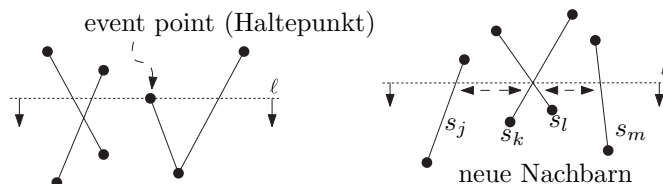
**Anmerkung:** Es gibt auch Situationen, in denen sich die Scanline nicht parallel fortbewegt, sondern z.B. als Halbgerade 360 Grad um einen Punkt herumgeht. In höheren Dimensionen kann man sich auch eine *scan plane* oder eine *scan hyperplane* vorstellen.

SWEEP/SCAN-LINE

$Q$  = Haltepunkte // Haltepunkt Struktur, meist nach einem Kriterium sortiert  
 $L = \emptyset$  // Scan-Line Struktur, Menge der aktiven Objekte

**while**  $Q \neq \emptyset$   
 Gehe zum nächsten Haltepunkt in  $Q$   
 AKTUALISIERE( $L$ )  
 Gebe Teillösungen aus // falls vorhanden

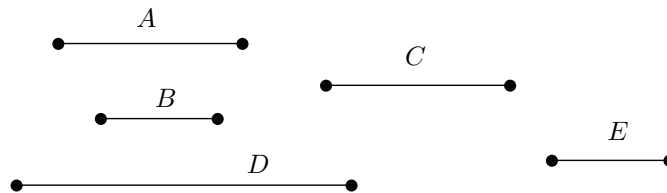
Beispielsweise sind beim Segment Schnitt Problem die Haltepunkte die Anfangs-, End- und Schnittpunkte von Segmenten.



BEISPIEL: Sichtbarkeitsproblem

Zwei Liniensegmente  $s_1$  und  $s_2$  in einer gegebenen Menge horizontaler Liniensegmente sind gegenseitig sichtbar, wenn es eine vertikale Gerade gibt, die  $s_1$  und  $s_2$  schneidet, aber kein weiteres Segment zwischen  $s_1$  und  $s_2$  liegend.





Wie können wir alle Paare von Segmenten ausgeben die sich gegenseitig sehen? (Think about it!) Im obigen Beispiel also die Paare  $\{A, B\}$ ,  $\{A, D\}$ ,  $\{B, D\}$ ,  $\{C, D\}$ .

Allgemeine Idee: Scanline. Alle Paare von Segmenten die sich gegenseitig sehen sind zu einem Zeitpunkt direkte Nachbarn in der  $y$ -Ordnung.

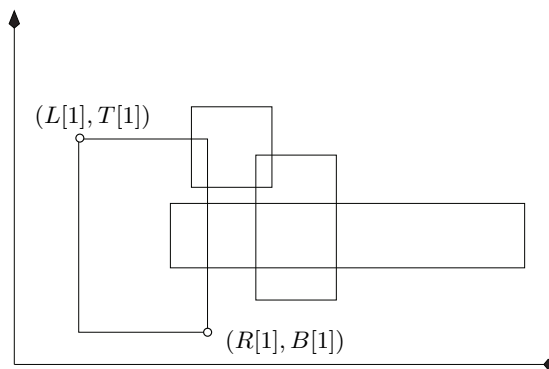
SICHTBARKEIT( $S$ )

```
// Scanline Algorithmus
Q = Folge der  $2n$  Anfangs- und Endpunkte von
Elementen aus der Menge  $S$  von Liniensegmenten in aufsteigender  $x$ -Reihenfolge
L =  $\emptyset$  // Menge der aktiven Segmente
```

```
while Q  $\neq \emptyset$ 
  p = SELECT_AND_REMOVE(Q)
  bestimme Nachbarn  $s_1$  und  $s_2$  von Segment  $s$ 
  if p ist linker Endpunkt von s
    füge s in L ein
    gib  $(s, s_1)$  und  $(s, s_2)$  aus
  else entferne s aus L
    gib  $(s_1, s_2)$  aus
```

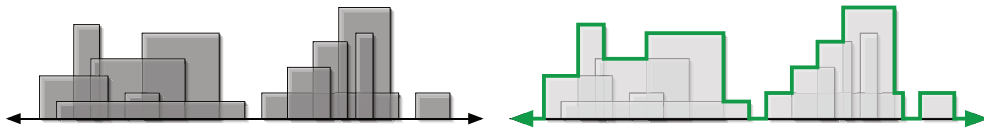
### Zusatzübungen

- Gegeben seien  $n$  Rechtecke in der Ebene, jedes spezifiziert durch eine linke-obere  $(L[i], T[i])$  und eine rechte untere Koordinate  $(R[i], B[i])$ .



Gegeben sind also die vier Arrays  $L[1 \dots n]$ ,  $R[1 \dots n]$ ,  $T[1 \dots n]$  und  $B[1 \dots n]$

- Beschreibe und analysiere einen Algorithmus der bestimmt ob sich irgendwelche zwei Rechtecke schneiden. (in  $\mathcal{O}(n \log n)$ )
  - Beschreibe und analysiere einen Algorithmus der einen Punkt  $P$  bestimmt, der in den meisten Rechtecken liegt (in  $\mathcal{O}(n \log n)$ )
- Beschreibe und analysiere einen Scanline Algorithmus der bestimmt, ob es in einer Menge von Kreisen zwei gibt, die sich schneiden (in  $\mathcal{O}(n \log n)$ ). Jeder Kreis ist spezifiziert durch die Koordinaten seines Zentrums  $(x_i, y_i)$  und seinen Radius  $r_i$ .
  - Beschreibe und analysiere einen Scanline Algorithmus, der die Skyline der folgenden Städte bestimmt. D.h. gebe die Eckpunkte der Skyline aus, in  $\mathcal{O}(n \log n)$ , wobei  $n$  die Anzahl der Häuser ist.
    - Manhattan: Jedes Gebäude ist ein Rechteck dessen untere Kante auf der  $x$ -Achse zu liegen kommt, und ist spezifiziert durch eine linke und rechte  $x$ -Koordinate sowie dessen Höhe.



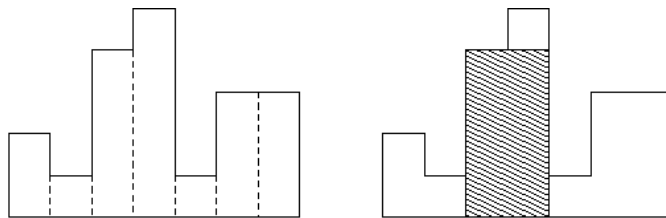
Die Manhattan Skyline

- b) Gizah: Jedes Gebäude ist ein gleichschenkliges rechtwinkliges Dreieck, dessen Hypotenuse auf der  $x$ -Achse zu liegen kommt, und ist spezifiziert durch die  $(x, y)$  Koordinate seiner Spitze.



Ägyptische Skyline

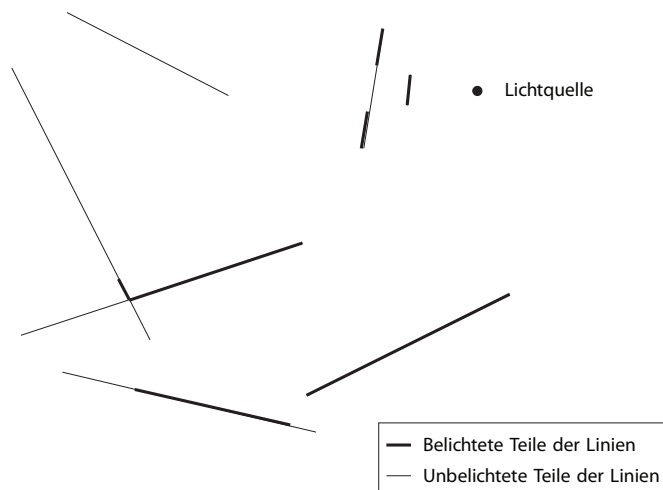
4. Ein Histogramm ist ein Polygon aus einer Folge von Rechtecken, die auf einer gemeinsamen Grundlinie ausgerichtet sind. Die Rechtecke haben alle die gleiche Breite können aber unterschiedliche Höhen haben. Zum Beispiel zeigt folgende Abbildung (links) ein Histogramm, das aus Rechtecken besteht mit den Höhen 2, 1, 4, 5, 1, 3, 3, in Einheiten gemessen, wobei 1 die Breite der Rechtecke ist:



Normalerweise werden Histogramme verwendet, um diskrete Verteilungen darzustellen, z. B. die Häufigkeiten von Zeichen in Texten. Gesucht ist die Fläche des grössten Rechtecks in einem Histogramm, das an den gemeinsamen Grundlinie ausgerichtet ist. Die Abbildung rechts zeigt dieses Rechteck. Gegeben  $n$  und die Höhen der Rechtecke eines Histogramms in dieser Reihenfolge  $h[1], \dots, h[n]$ , bestimme die Fläche des grössten Rechtecks in diesem Histogramm.

(möglich in  $\mathcal{O}(n)$ )

5. Gegeben sei eine Menge von  $n$  Liniensegmenten mit beliebiger Orientierung in der Ebene, in allgemeiner Lage. Sei weiter eine punktförmige Lichtquelle gegeben, die die Ebene belichtet. Jedes Liniensegment sei undurchsichtig und schirme das Licht ab. Gebe ein Verfahren an, das möglichst effizient feststellt, welche Teile jedes Segments belichtet sind. Speicherbedarf, Laufzeit? Welche Datenstrukturen?



# Interval-Baum

Ein Intervall-Baum ist eine Datenstruktur (Baum) in der Intervalle (mit Endpunkten in einer diskreten Menge von Endpunkten) mit nur linearem Speicherbedarf abgelegt werden können und mit der man Aufspiessanfragen effizient beantworten kann.

Eine Aufspiessanfrage (*stabbing query*) ist folgendermassen formuliert: "Gib alle Intervalle zurück, die im Baum gespeichert sind und eine gegebene Zahl  $x$  enthalten". Die Anzahl solcher Intervalle wird in den Laufzeitanalysen mit  $k$  bezeichnet.

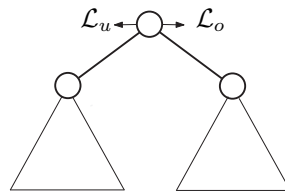
Mit einem Intervallbaum sind die Operationen *Einfügen* eines Intervalls, *Entfernen* eines Intervalls und *Aufspiess-Fragen* in Zeit  $\mathcal{O}(\log n)$  bzw.  $\mathcal{O}(\log n + k)$  möglich.

## Skelett

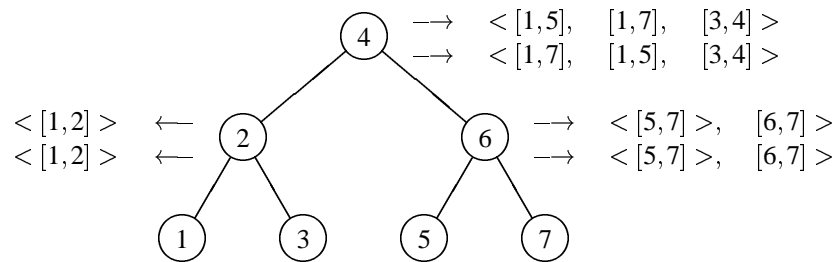
Der Intervallbaum ist eine sogenannte *halbdynamische Skelettstruktur*: anstatt Strukturen zu benutzen, deren Grösse sich der Menge der jeweils vorhandenen geometrischen Objekten voll dynamisch anpasst, schaffen wir zunächst ein anfänglich leeres Skelett über einem *diskreten Raster*, das allen im Verlauf des Verfahrens benötigten Objekten Platz bietet.

Der *Aufbau* geschieht indem man zuerst einen balancierten Binärsuchbaum mit allen unteren und oberen Intervallgrenzen aufbaut, inden man dann alle Intervalle darin einfügt.

Jedes Intervall  $i$  ist genau in einem Knoten  $p$  gespeichert, und zwar so, dass  $p$  der höchstmögliche Knoten ist, so dass  $p.value \in i$ . Für jeden Knoten gibt es dann eine *u-Liste* und eine *o-Liste* mit jeweils allen, dem Knoten zugeordneten Intervallen, sortiert nach der unteren bzw. oberen Grenze.



Folgendes Beispiel zeigt einen Intervall-Baum für die Menge  $\{[1, 2], [1, 5], [3, 4], [5, 7], [6, 7], [1, 7]\}$  von Intervallen mit Endpunkten in  $\{1 \dots 7\}$ :



## u-Liste/o-Liste

werden als balancierte Suchbäume implementiert. Daraus folgt, dass das Einfügen eines Intervalls in einer Anzahl von Schritten ausgeführt werden kann, die höchstens linear von der Höhe des Intervallbaum-Skeletts und logarithmisch von der Länge der einem Knoten zugeordneten u- und o-Listen abhängt.

## Einfügen eines Intervalls $I$

1. Suche bis Knoten  $p$  gefunden ist mit  $p.value \in I$
2. Füge  $I$  in  $\mathcal{L}_u$  und  $\mathcal{L}_o$  ein

```

EINFÜGEN( $I$  : Intervall,  $p$  : Knoten)
  // anfangs ist  $p$  die Wurzel des Intervall-Baumes
  //  $I = [l, r]$  ist ein Intervall mit linkem Endpunkt  $l$  und rechtem Endpunkt  $r$ 
  if  $value[p] \in I$ 
    Füge  $I$  entsprechend seinem unteren Endpunkt in  $\mathcal{L}_u$  von  $p$ 
    und entsprechend seinem oberen Endpunkt in  $\mathcal{L}_o$  von  $p$  ein und fertig!
  else if  $value[p] < l$ 
    EINFÜGEN( $I, right[p]$ )
  else EINFÜGEN( $I, left[p]$ )           //  $value[p] > l$ 

```

### Entfernen eines Intervalls $I$

analog zum Einfügen

1. bestimme (ausgehend von der Wurzel): Knoten  $p$  mit geringster Tiefe mit  $p.value \in I$
2. Entfernen des Intervalls aus den Suchbäumen  $\mathcal{L}_u$  und  $\mathcal{L}_o$  von  $p$

### Aufspiessanfrage

Bei der Aufspiessanfrage wird einfach nach dem Wert  $x$  gesucht, und bei jedem Knoten auf dem Pfad die u-Liste oder o-Liste (je nach dem auf welcher Seite des Knotens  $x$  liegt) so lange durchgegangen, bis  $x$  nicht mehr in dem aktuellen Intervall ist.

```

REPORT( $p, x$ )
  // Input: Die Wurzel  $p$  eines Intervallbaumes und ein Anfragepunkt  $x$ 
  // Output: Alle Intervalle die  $x$  enthalten

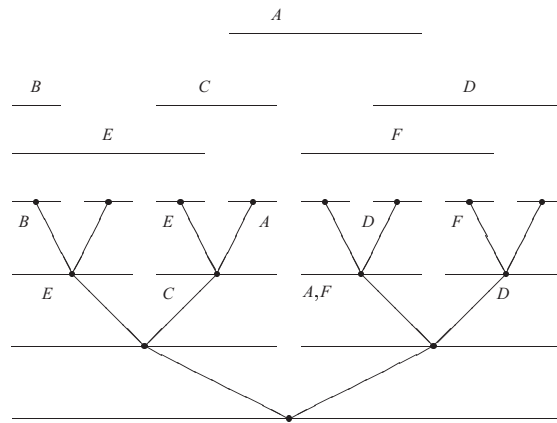
  if  $p$  ist kein Blatt
    if  $x < value[p]$ 
      Gehe startend mit dem Interall mit dem linkesten Endpunkt durch
       $\mathcal{L}_u$  von  $p$  und gebe alle Intervalle aus die  $x$  enthalten.
      Stoppe sobald ein Intervall  $x$  nicht enthält
      REPORT( $left[p], x$ )
    else Gehe startend mit dem Interall mit dem rechtesten Endpunkt durch
       $\mathcal{L}_o$  von  $p$  und gebe alle Intervalle aus die  $x$  enthalten.
      Stoppe sobald ein Intervall  $x$  nicht enthält
      REPORT( $right[p], x$ )

```

## Segment-Baum

Ein Segment-Baum hat grundsätzlich die gleichen Operationen wie ein Intervall- Baum, ist aber anders aufgebaut und braucht mehr Speicherplatz. Hier wird jedes Intervall  $I$  in  $\mathcal{O}(\log n)$  Teilintervalle (die exakt als Knoten im Baum repräsentiert werden) aufgeteilt.  $I$  wird dann in all diese Knoten eingefügt, die eines dieser Teilintervalle repräsentieren. Da jedes Intervall in  $\mathcal{O}(\log n)$  Knoten gespeichert wird, braucht der Segment-Baum auch  $\mathcal{O}(n \log n)$  Speicherplatz. Alle Laufzeiten bleiben aber gleich wie beim Intervall-Baum.

Beispiel:



## Struktur

Zuerst wird ein leeres Skelett aufgebaut, in welches anschließend dynamisch Segmente (Intervalle) eingefügt und wieder entfernt werden können. Das Skelett ist ein vollständiger Binärbaum. Intervalle mit Endpunkten aus einer fixen Menge  $\{1, \dots, n\}$  können daher aufgenommen werden. Die Blätter repräsentieren elementare Intervalle (d.h. der Form  $[i, i + 1]$ ). Jeder innere Knoten repräsentiert die Vereinigung aller elementaren Intervalle in seinem Teilbaum (vgl. Abbildung).

Die Wurzel repräsentiert demzufolge das gesamte Intervall  $[1, n]$ .

## Einfügen

Um ein Intervall  $[l, r]$  einzufügen wird beginnend bei der Wurzel bei jedem Knoten entschieden, ob das von ihm repräsentierte Intervall vollständig in  $[l, r]$  liegt. Ist dies der Fall wird das Intervall bei diesem Knoten vermerkt. Ansonsten wird für beide Kinder überprüft, ob sie mit  $[l, r]$  überlappen und entsprechend wird rekursiv vorgegangen.

Da höchstens logarithmisch viele Knoten während des Einfügens besucht werden, ist die Laufzeit für eine Einfüge-Operation in  $\mathcal{O}(\log n)$ .

## Löschen

Das Löschen eines Intervalls erfolgt im Prinzip analog zum Einfügen. Um bei jedem Knoten zu überprüfen, ob das Intervall dort gespeichert ist sind schlimmstenfalls linear viele Schritte notwendig, wenn eine verkettete Liste verwendet wird. Deshalb wird z.B. ein balancierter Binärsuchbaum (z.B. AVL Baum) verwendet. Eine Struktur, die das Einfügen und Entfernen in logarithmischer Zeit ermöglicht. Damit ist auch das Entfernen aus dem Segment-Baum in logarithmischer Zeit möglich.

## Aufspiessanfragen

Aufspiessanfragen können in einer rekursiven Art für einen gegebenen Punkt  $x$  beantwortet werden:

```
REPORT( $p, x$ )
  // Input: Die Wurzel  $p$  eines Segmentbaumes und ein Anfragepunkt  $x$ 
  // Output: Alle Intervalle die  $x$  enthalten

  Gebe alle Intervalle von  $p$  aus
  if  $p$  ist kein Blatt
    if  $p$  hat linkes Kind  $left[p]$  mit Intervall, welches  $x$  enthält
      REPORT( $left[p], x$ )
    if  $p$  hat rechtes Kind  $right[p]$  mit Intervall, welches  $x$  enthält
      REPORT( $right[p], x$ )
```

## Range Tree ( $d$ -dimensional)

Range Trees sind Bäume, die es ermöglichen, Bereichsanfragen im  $d$ -dimensionalen Raum effizient zu beantworten.

Für  $d = 1$  sind das gewöhnliche (balancierte) Binäre Bäume, deren Blätter meist verkettet sind (Blattsuchbaum). Dadurch kann man die Liste durchgehen, um alle Elemente im Bereich zu ermitteln.

Für höhere Dimensionen lässt sich diese Idee verallgemeinern. Der Baum hat die  $x$ -Koordinate als Schlüssel. Jeder innere Knoten  $k$  zeigt auf einen  $y$ -Baum ( $y$ -Koordinate dient als Schlüssel), in dem alle im Teilbaum von  $k$  liegenden Werte enthalten sind. Für die dritte Dimension zeigt wiederum jeder innere Knoten jedes  $y$ -Baumes auf einen  $z$ -Baum. Dies kann in beliebige Dimensionen getrieben werden.

### Analyse

#### Query

$\mathcal{O}(\log^d n + k)$ ,  $k$ : inspizierte Elemente ( $\neq$  Anzahl Elemente im Bereich!)

Jede Dimension benötigt  $\mathcal{O}(\log n)$  für die Suche in jeweils einem Baum, und es werden jeweils auch  $\mathcal{O}(\log n)$  Bäume der nächsten Dimension durchsucht.

#### Speicherplatz

$\mathcal{O}(n \cdot \log^{d-1} n)$

Baum der 1. Dimension ( $x$ -Baum) benötigt  $\mathcal{O}(n)$  Speicherplatz. Für jedes Level dieses Baumes kommt jedes Element (also insgesamt  $n$  Elemente) in den entsprechenden  $y$ -Bäumen genau einmal vor. Daher beanspruchen alle  $y$ -Bäume zusammen  $\mathcal{O}(n \log n)$  Speicherplatz (es gibt  $\mathcal{O}(\log n)$  viele Levels).

Nun kann man dies wiederum rekursiv anwenden. Wir wissen also, dass ein  $2 - d$  Range-Tree  $\mathcal{O}(n \log n)$  Speicherplatz benötigt.

Bei einer dritten Dimension hat man wiederum eine obere Limite von  $\mathcal{O}(n \log n)$  pro Level was zu  $\mathcal{O}(n \log^2 n)$  führt, usw.

*Wichtig: Range-Trees sind statisch, Dynamik (insert/remove) kriegt man nicht einfach so hin und muss dafür andere Ansätze verfolgen*

## Structural DP: Minimum Vertex Cover

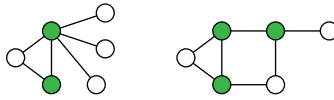
DP auf kombinatorischen Strukturen (rekursive Def.) anstatt einer "Sequenz"/Sequenzen. Beispielsweise Bäume: Azyklische Substruktur. Teilproblem: Teilbaum.

Ein Vertex Cover (eine Knotenüberdeckung) eines Graphen ist eine Menge von Knoten, sodass jede Kante des Graphen mindestens einen Endknoten im Cover hat. Das Problem ein Minimales Vertex Cover zu finden ist ein klassisches Optimierungsproblem in der Informatik und ist ein typisches Beispiel für ein NP-schweres Optimierungsproblem.

Formale Definition:

- Gegeben ein ungerichteter Graph  $G = (V, E)$
- Finde eine kleinste Menge  $C \subset V$ , so dass  $|C|$  minimal ist und  $\forall e = \{u, v\} \in E : \{u, v\} \cap C \neq \emptyset$ .

Beispiel:



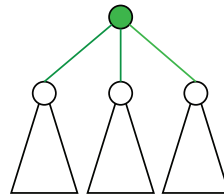
Z.B. Positioniere möglichst wenige Wachen um alle Korridore eines Museums zu bewachen.

Obwohl NP-schwer auf allgemeinen Graphen, kann ein Minimum Vertex Cover in polynomieller Zeit gefunden werden, wenn der Graph ein Baum ist.

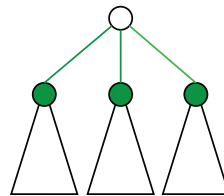
### Struktur einer optimalen Lösung

Folgendes gilt ziemlich offensichtlich für eine optimale Lösung: Entweder ist die Wurzel des Baumes im Vertex Cover oder nicht. Schauen wir beide Möglichkeiten an:

- Ist die Wurzel des Baumes im Vertex Cover, so sind alle Kanten von der Wurzel zu seinen Kindern abgedeckt. Die optimale Lösung muss desweiteren optimal auf allen Teilbäumen die an der Wurzel hängen sein. Übliches Widerspruchs-Argument (Cut & Paste, think about it): wäre dem nicht so, so könnte die Teillösung ersetzt werden und eine bessere Gesamtlösung wäre gefunden. Dies widerspricht der Annahme der Optimalität der Lösung von der wir ausgegangen sind.



- Ist die Wurzel des Baumes nicht im Vertex Cover, so sind, da alle Kanten von der Wurzel zu seinen Kindern abgedeckt werden müssen, alle Kinder im Vertex Cover.



Desweiteren, muss die optimale Lösung optimal auf allen Teilbäumen sein, mit der Bedingung, dass die Wurzel des Teilbaumes im Vertex Cover ist. Widerum ein Widerspruchsargument.

All dies legt nahe, dass um eine optimale Lösung für das gesamte Problem zu erhalten, wir die optimalen Lösungen von Teilproblemen (Teilbäume) betrachten müssen.

**Stop and Think:** Man kann die Struktur einer optimalen Lösung auch anders analysieren, in dem man im zweiten Fall (Wurzel ist nicht im Vertex Cover) die optimale Lösung auf den Teilbäumen mit einer Wurzel, die ein Kindeskind der Wurzel des gesamten Baumes ist, analysiert. Versuche eine DP-Lösung mit diesem Ansatz herzuleiten.

### Rekursion

Obige Zusammenhänge können wir nun direkt rekursiv formulieren:

- Es sei  $cost_{in}(v)$  die Anzahl Knoten in einem Minimum Vertex Cover auf dem (Teil)baum mit der Wurzel  $v$ , unter der Annahme, dass  $v$  im Vertex Cover ist.
- Es sei  $cost_{out}(v)$  die Anzahl Knoten in einem Minimum Vertex Cover auf dem (Teilbaum mit der Wurzel  $v$ , unter der Annahme, dass  $v$  nicht im Vertex Cover ist.

Die gesuchte Lösung ist deshalb  $\min\{opt_{in}(\text{root}), opt_{out}(\text{root})\}$

- *Verankerung*: Ist  $v$  ein Blatt so gilt:  $opt_{in}(v) = 1$  und  $opt_{out}(v) = 0$
- *Rekursion*: Mit einer analogen Argumentation wie oben für das Gesamtproblem ist klar, dass

$$opt_{in}(v) = 1 + \sum_{u \in \text{children}(v)} \min\{opt_{in}(u), opt_{out}(u)\}$$

und

$$opt_{out}(v) = \sum_{u \in \text{children}(v)} opt_{in}(u)$$

Man könnte dies nun so rekursiv implementieren. Dann würden aber die selben Teilprobleme mehrmals (und zwar sehr oft) gelöst werden (Wieso? think about it!), deshalb speichern wir die Lösung von bisher gelösten Teilproblemen ab um einen effizienten Algorithmus zu erhalten.

### Bottom-Up

Für die iterative Berechnung verwenden wir zwei Arrays  $IN[]$  und  $OUT[]$  die das Resultat für  $opt_{in}$  und  $opt_{out}$  speichern. Es ist nun eine Durchlaufordnung gesucht, so dass wenn wir ein Teilproblem lösen alle benötigten Teilresultate bereits berechnet worden sind. Dies ist zum Beispiel möglich, wenn wir die Knoten in *Post-Order* besuchen (think about it!).

Entweder diese Ordnung ist bereits gegeben, oder sonst können zuerst mit DFS (Tiefensuche) die Knoten in Post-Order enumeriert werden. Folgende Funktion implementiert dieses Verfahren

DP-DFS( $v$ )

```
if  $v$  ist Blatt
     $IN[v] = 1$ 
     $OUT[v] = 0$ 
else //  $v$  ist innerer Knoten
    // Berechnen  $IN$  und  $OUT$  Werte
     $OUT[v] = 0$ 
     $IN[v] = 1$ 
    foreach  $u \in \text{children}[v]$ 
        DP-DFS( $u$ )
         $IN[v] = IN[v] + \min(IN[u], OUT[u])$ 
         $OUT[v] = OUT[v] + IN[u]$ 
```

*Laufzeit*: Die Laufzeit ist linear in der Anzahl der Knoten des Baumes (also  $\mathcal{O}(n)$  bei  $n$  Knoten)

In der Basisprüfung 2008 wurde das Matching Problem auf Bäumen behandelt.

Siehe <http://goo.gl/vxuEqF> Aufgabe 4.

Die Lösung ist in <http://goo.gl/kibYsH> erklärt.

Ein paar Ausführungen zum Minimum Dominating Set Problem:

[http://www.cs.ucdavis.edu/~margulie/ecs122a\\_s08/hw6/hw6\\_s08\\_sol.pdf](http://www.cs.ucdavis.edu/~margulie/ecs122a_s08/hw6/hw6_s08_sol.pdf) (<http://goo.gl/pifv5>)

Zusätzliche Beispiele hats in

- [www.cs.berkeley.edu/~vazirani/s99cs170/notes/dynamic2.ps](http://www.cs.berkeley.edu/~vazirani/s99cs170/notes/dynamic2.ps)  
(<http://goo.gl/TJXqP>)
- <http://theory.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/03-dynprog.pdf>  
(<http://goo.gl/5KdYl>)
- [http://stellar.mit.edu/S/course/6/fa09/6.046/courseMaterial/topics/topic2/lectureNotes/L10\\_-\\_Dynamic\\_Programming/L10\\_-\\_Dynamic\\_Programming.pdf](http://stellar.mit.edu/S/course/6/fa09/6.046/courseMaterial/topics/topic2/lectureNotes/L10_-_Dynamic_Programming/L10_-_Dynamic_Programming.pdf)  
(<http://goo.gl/1Lz14>)