Datenstrukturen & Algorithmen 14. Mai 2014

Handout 10

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Konvexe Hülle, Intervall-Baum, Segment-Baum, DP auf Bäumen: Minimum Vertex Cover

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

Links

- Belfrage Slides: http://www.belfrage.net/eth/d&a/pdf/uebung6_h.pdf (http://goo.gl/Pvjei)
- 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)

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.

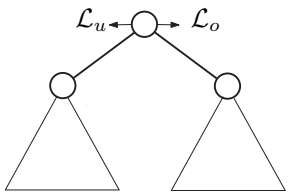
Mit einem Intervallbaum sind die Operationen *Einfügen* eines Intervalls, *Entfernen* eines Intervalls und *Aufspiess-Fragen* in Zeit $\mathcal{O}(\log n)$ bzw. $\mathcal{O}(\log n + k)$ möglich.

Skelett

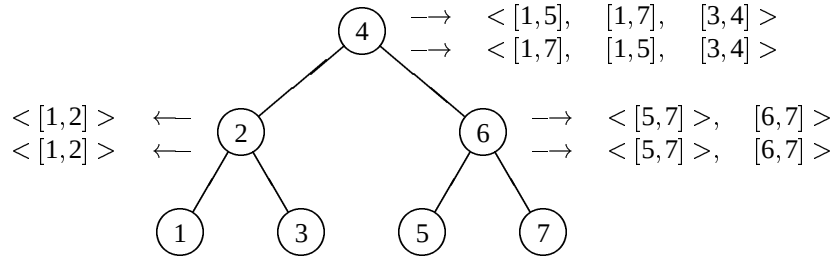
Der Intervallbaum ist eine sogenannte *halbdynamische Skelettstruktur*: anstatt Strukturen zu benutzen, deren Grösse sich der Menge der jeweils vorhandenen geometrischen Objekten voll dynamisch anpasst, schaffen wir zunächst ein anfänglich leeres Skelett über einem *diskreten Raster*, das allen im Verlauf 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 Konten 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 $\{1\ldots7\}$:



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.

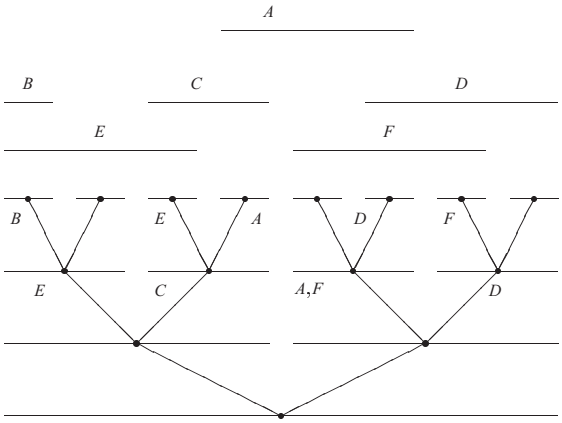
REPORT(p, x)
// Input: Die Wurzel p eines Intervallbaumes und ein Anfragepunkt x
// Output: Alle Intervalle die x enthalten

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 wir ein leeres Skelett aufgebaut, in welches anschliessend dynamisch Segmente (Intervalle) eingefügt und wieder entfernt werden können. Das Skelett ist ein vollst andiger Binärbaum. Intervalle mit Endpunkten aus einer fixen Menge $\{1, \ldots, n\}$ 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 Interval $[l, r]$ 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:

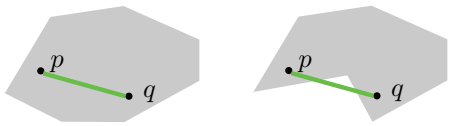
REPORT(p, x)

// 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$(right[p], x)$

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.

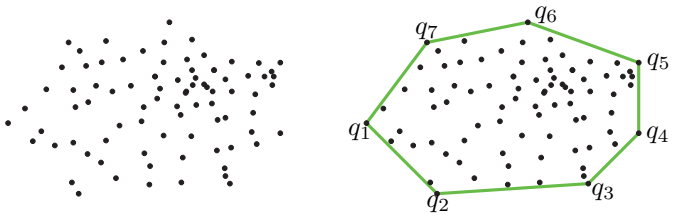


konvex nicht konvex

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 $\mathcal{P}$!



a) Input b) Output

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 nächten Punkt auf der Hülle als denjenigen, der am rechtesten von dem vorherigen Punkt ist.

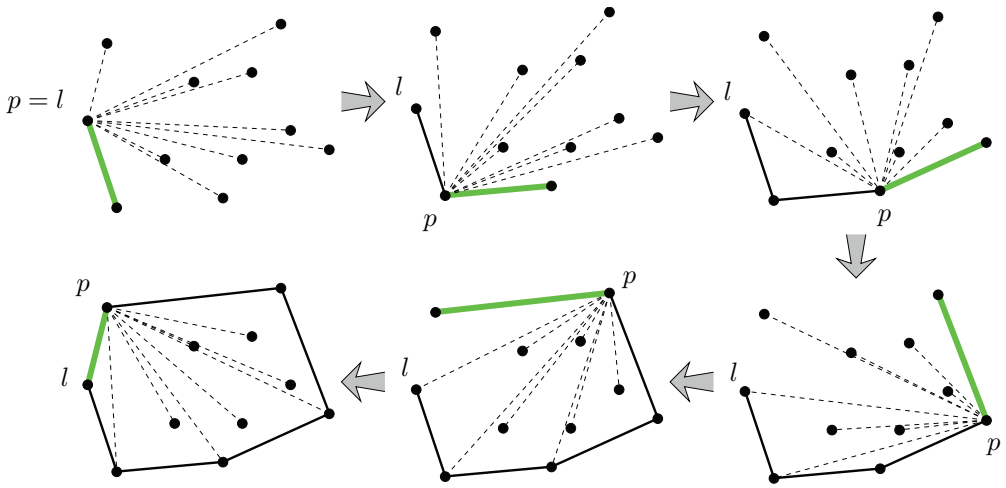


Abb.: Beispieldurchlauf von Jarvis March

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 $\overline{p\,q}$ 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 verwerfe diejenigen die eine Drehung im Uhrzeigersinn provozieren würden

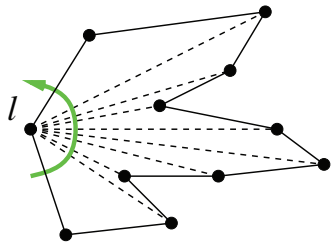


Abb.: Durch die radiale Sortierung betrachtet der Algorithmus die Punkte (implizit) entlang eines *simplen* Polygons

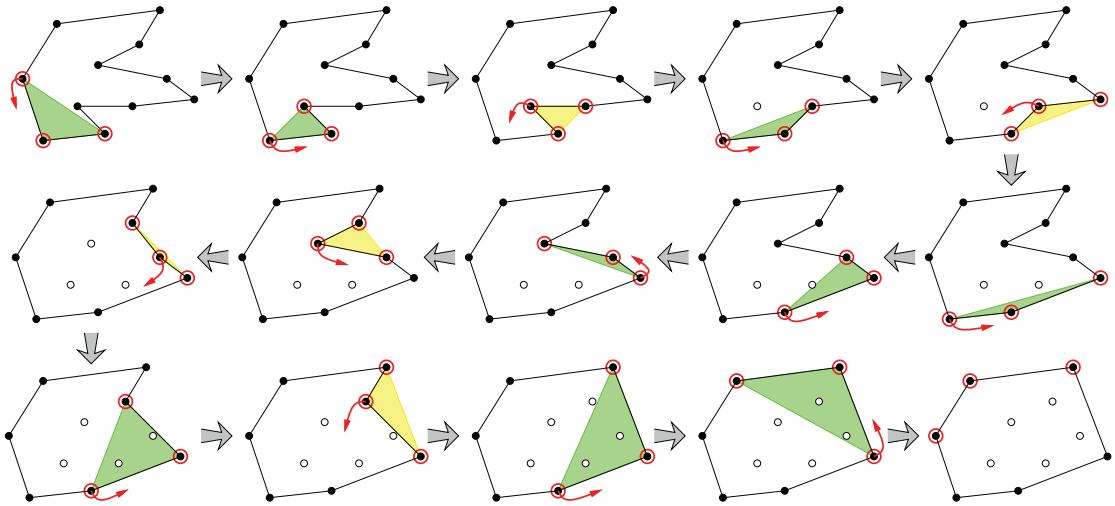


Abb.: Die "Drei Groschen-Oper"

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

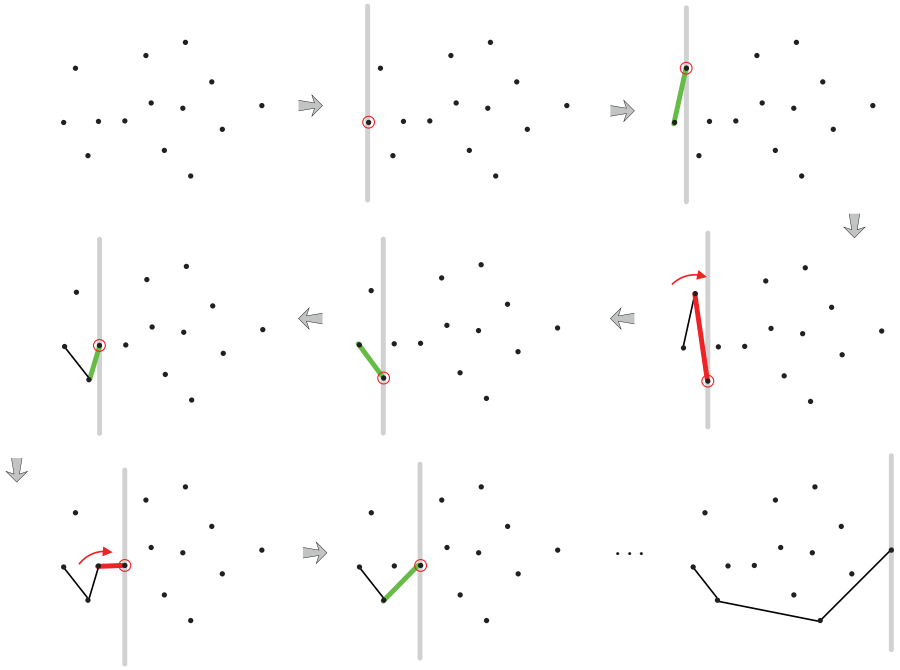


Abb.: Bestimmung der unteren Hülle

```
// 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);
}
```

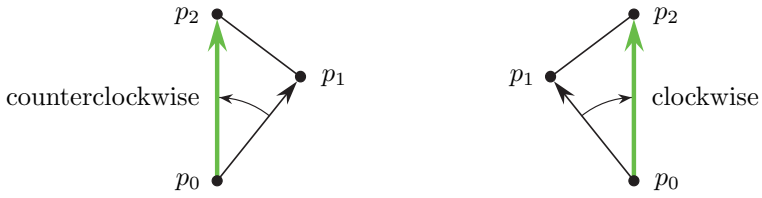
Degenerierte Fälle

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 liegt rechts von dem (gerichteten) Liniensegment $\overrightarrow{p_0\,p_1}$ wenn die Punkte p_0, p_1, p_2 im Uhrzeigersinn durchlaufen werden:



a) b)

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 $\overrightarrow{p_0\,p_2}$ im oder gegen den Uhrzeigersinn relativ zu $\overrightarrow{p_0\,p_1}$ 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 Ausnahmebedingungen 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 Ordungseigenschaften wie dieser Winkel besitzt.

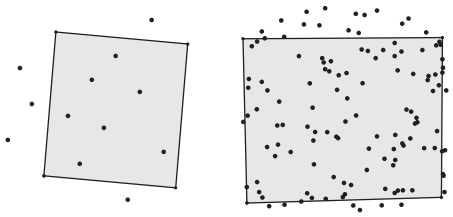
Sedgewick Theta Function

```
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.

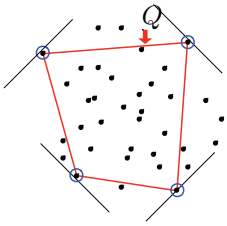
Stop and Think: Wie kann die CW Funktion verwendet werden, um zu entscheiden, ob sich zwei Liniensegmente $\overline{a\, b}$ und $\overline{c\, d}$ schneiden?

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?

Structural DP: Minimum Vertex Cover

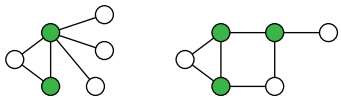
DP auf kombinatorischen Strukturen (rekursive Def.) anstatt einer "Sequenz"/Sequenzen. Beispielsweise Bäume: Azyklische Substruktur. Teilproblem: Teilbaum.

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:



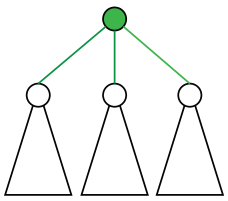
Z.B. Positioniere möglichst wenige Wachen um alle Korridore eines Museums zu bewachen.

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

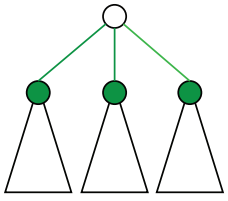
Struktur einer optimalen Lösung

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 $cost_{\text{in}}(v)$ 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 $cost_{\text{out}}(v)$ 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 $\min\{opt_{\text{in}}(\text{root}), opt_{\text{out}}(\text{root})\}$

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

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

und

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

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 ∈ 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://www.ti.inf.ethz.ch/pw/teaching/d+a/exercises/exams/ExamHE2008.pdf
(http://goo.gl/5uClL) Aufgabe 4.

Die Lösung ist in http://www.ti.inf.ethz.ch/pw/teaching/d+a/exercises/exams/SolutionsHE2008.pdf
(http://goo.gl/EbxYe) erklärt.

Ein paar Ausführungen zum Minimum Dominating Set Problem:
http://wwwcsif.cs.ucdavis.edu/~margulie/ecs122a_s08/hw6/hw6_s08_sol.pdf (http://goo.gl/pifv5)

Zusäztliche Beispiele hats in

- 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://goo.gl/lLz14)

IO in Eiffel

Folgende Klasse kann für ein gepuffertes Einlesen verwendet werden

```
class
        SCANNER
create
        make

feature {SCANNER}
        length: INTEGER = 1048576
        buffer: STRING
        at: INTEGER

feature
        make
                do at := 1; create buffer.make_empty end

        nextChar: CHARACTER
                do
                        if at > buffer.count then
                                io.read_stream (length)
                                buffer := io.last_string
                                at := 1
                        end
                        Result := buffer[at]
                        at := at+1
                end

        nextInt: INTEGER
                local
                        c: CHARACTER
                do
                        from until c.is_digit loop c := nextChar end
                        from Result := 0 until not c.is_digit loop
                                Result := 10*Result + (c.code - ('0').code)
                                c := nextChar
                        end
                end
end
```

Ansonsten kann auch die C scanf Funktion verwendet werden

```
readInteger : INTEGER
external
        "C++ inline use <cstdio>"
alias
        "[
                int t; scanf ("%d ", &t);
                return t;
        ]"
end
```

Brain Teaser: Wine

Zu Ehren des Monarchen wird ein Fest abgehalten. Da dabei immer gerne reichlich getrunken wird, stehen 240 Fässer Wein bereit. Der Monarch misstraut aber seinen Untertanen und hat den Verdacht, das (genau) eines der Fässer mit Gift versetzt wurde. Jeder der vom vergifteten Wein trinkt stirb innerhalb von 24 Stunden. Der Monarch hat 5 Sklaven die er bereit ist zu opfern um zu entscheiden, welches das vergiftete Fass ist. Wie kann erreicht werden, das dies nach (spätestens) 48 Stunden herausgefunden ist?