5.1.3 Pfaderhöhung: (Maxflow) s, t

Idee: Versuche im Restnetzwerk einen Pfad mit Restkapazität zu finden, indem z.B mit `explorefrom(s)`, `dfs(s)` oder `bfs(s)` alle von s aus erreichbaren Knoten gesucht werden, bis t gefunden wird.

Labeling-Algorithmus

```
void MF_Labeling(const graph& G, node s, node t,const
   edge_array<int>& cap, edge_array<int>& flow){
   node_array<bool> labeled; // besuchte Knoten
   node_array<edge> PRED; // Pfade

   // Initialisierung
   edge e;
   forall_edges(e,G){
      flow[e] = 0
   }
   node v;
   forall_nodes(v,G){
      labeled[v] = false
      pred[v] = NULL
   }

   while(true){
      queue<node> L; // Breitensuche
      labeled[s] = true;
      L.append(s);
      while(!L.empty()){
         node v = L.pop();
         // Iteriere über alle im Restnetzwerk adjazenten Kanten
         edge e;
         forall_out_edges(e,v){
            if(flow[e] == cap[e]) // Restkapazität ist null
               continue; // e nicht in G(x)
            node w = G.target(e);
            if(labeled[w])
               continue;
            labeled[w] = true;
            PRED[w] = e;
```

```
            L.append(w);
            }
            forall_in_edges(e,v){
            if(flow[e] == 0) continue;
            node w = G.source(e);
            if(labeled[w]) continue;
            labeled[w] = true;
            PRED[w] = e;
            L.append(w);
         }
         if(labeled[t])
            L.clear();
      } // Ende while-Schleife
      if(labeled[t])
         AUGMENT(G,s,t,PRED,cap,flow);
      else
         break; // ex. kein erhöhender Pfad
   }
}
```

Listing 5.1: MF_Labeling

```
void AUGMENT(const graph& G, node s, node t, const
   node_array<edge>& PRED, const edge_array<int>& cap,
   edge_array<int>& flow){
   int delta = MAXINT; // Restkapazität von P
   node v = t;
   while(v != s){
      int r;
      edge e = PRED[v];
      if(v == G.source(e)){ //Rückwärtskante
         r = flow[e];
         v = G.target(e);
      }
      else{ // Vorwärtskante
         r = cap[e] - flow[e];
         v = G.source(e);
      }
      if(r < delta)
         delta = r; // min
   }
```

```
    // Eigentliche Flusserhöhung
    v = t;
    while(v != s){
        edge e = PRED[v];
        if(v == G.source(e)){ //Rückwärtskante
            flow[e] = flow[e] - delta;
            v = G.target(e);
        }
        else{ // Vorwärtskante
            flow[e] = flow[e] + delta;
            v = G.source(e);
        }
    }
}
```

Listing 5.2: Augment

Erklärung:
Die äußere *while*-Schleife läuft so lange, bis explizit aus ihr herausgesprungen wird. Dies geschieht, wenn nach der inneren Schleife t nicht gelabelt ist, es also keinen erhöhenden Pfad mehr gibt.
Da s der Startknoten ist und jeder von s aus erreichte Knoten gelabelt und zur Menge L hinzugefügt wird, wird s zu Beginn immer gelabelt und in L eingefügt. In der inneren Schleife wird zuerst ein beliebiger Knoten u aus der Menge L entnommen. Von diesem Knoten u aus werden nun sowohl die ausgehenden, als auch die eingehenden Kanten betrachtet. Dabei werden diejenigen Kanten, die keine Restkapazität mehr besitzen, sofort aussortiert und nicht weiter betrachtet. Bei den restlichen wird überprüft, ob der durch die Kante erreichbare Knoten schon gelabelt wurde. Ist dies nicht der Fall wird er gelabelt, sein $PRED$-Verweis auf die Kante gesetzt, über die er erreicht wurde und der Knoten in die Menge L aufgenommen.
Die innere Schleife bricht ab, sobald L leer ist, was der Fall ist, wenn t gelabelt wurde, da dann ein erhöhender Pfad existiert und entlang diesem, im Algorithmus AUGMENT, Fluss geschickt wird.
Hierbei wird zuerst über den gefundenen Pfad von t nach s gelaufen, und dabei die maximale Flusserhöhung ausfindig gemacht. Ist diese gefunden, also s erreicht, wird erneut über den gefundenen Pfad von t nach s gelaufen und dabei die Flussänderung um den herausgefundenen Wert durchgeführt.
Es wird immer von t nach s gelaufen, da man nur den Vorgänger, nicht aber den Nachfolger eines Knotens kennt.

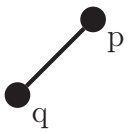
6 Algorithmische Geometrie

(Computational Geometry)

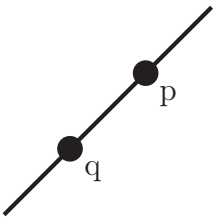
Die Objekte (Klassen / Typen) liegen hier in der Ebene $\mathbb{R}^2$
Hier werden die drei Typen *point*, *segment* und *line* genutzt. Ein Segment ist die Strecke zwischen zwei Punkten und *line* die Gerade durch zwei Punkte.

Beispiel 6.1
point $p(17.5, 1.3)$
point $q(x, y)$
$p \neq q$

segment $s(p, q)$



line $l(p, q)$



6.1 Operationen

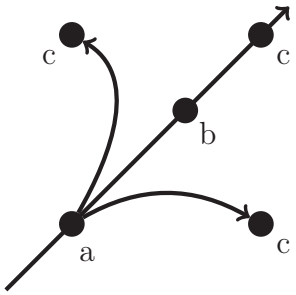
xy-Ordnung: $q \leq p$

$$p.comp_xy(q) = \begin{cases} -1, & p \leq_{xy} q \\ 0, & p = q \\ +1, & p \geq_{xy} q \end{cases}$$

Zuerst werden die x-Koordinaten der beiden Punkte verglichen. Sind diese gleich, werden die y-Koordinaten verglichen.

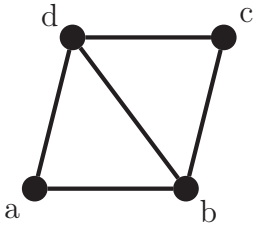
6.2 Orientierung

3 Punkte a, b, c



$$\text{Operation}(a,b,c) = \begin{cases} -1, & a,b,c \text{ Rechtskurve} \\ 0, & a,b,c \text{ co-linear} \\ +1, & a,b,c \text{ Linkskurve} \end{cases}$$

6.2.1 Lineare Algebra



Fläche des Parallelogramms hat Vorzeichen

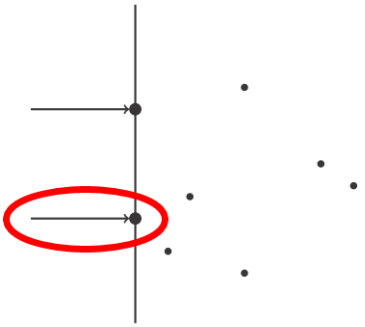
$$\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} \quad \begin{array}{ll} > 0, & a,b,c \text{ linksorientiert} \\ < 0, & a,b,c \text{ rechtsorientiert} \\ = 0, & a,b,c \text{ co-linaer} \end{array}$$

Fläche $\Delta = |\cdot|$

$\text{Operation}(a,b,c) = sign \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} \rightarrow$ einfach arithmetische Ausdruck

6.2.2 Minimum / Maximum in xy-Ordnung

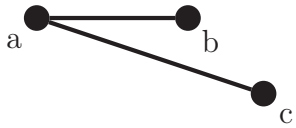
Min_{xy} ist der am meisten links liegende Punkt. Existieren mehrere, dann ist es der



$\rightarrow$ Extrempunkt

Analog: Max_{xy}, cmp_{xy}, ...

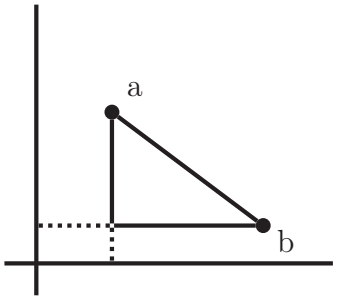
6.2.3 Vergleich von Entfernungen



dist(a.b) ? dist(a,c)
Ist a weiter von b oder von c entfernt?

dist Euklidischer Abstand
→ L_2-Norm

$dist(a, b) =$
$d_x \leftarrow a_x - b_x$
$d_y \leftarrow a_y - b_y$
$\sqrt{d_x^2 + d_y^2}$ (Hypothenuse)



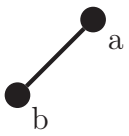
cmp_dist$(a, , b, c)$:= sign(dist(a,c) - dist(a,b))
$\sqrt{d_x^2 + d_y^2}$ - $\sqrt{d_x'^2 + d_y'^2}$
→ Vergleiche $(dist(a, c))^2$ mit $(dist(a, b))^2$
$= sign((c_x - a_x)^2 + (c_y - a_y)^2 - ((b_x - a_x) + (b_y - a_y)^2))$
→ Eliminierung der Quadratwurzel

Nur $+, -, *$
Vorteil:

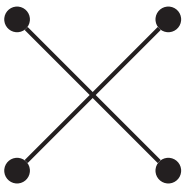
- schneller
- exakt (z.B. für int/long)

6.2.4 Anwendung von orientation (→ Schnitt-Test)

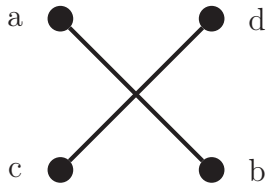
segment $s(a, b)$




```
point s.start();
point s.end();
```
Schnitt $s_1 \cap s_2$



Test mit orientation



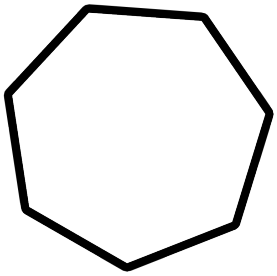
Schnitt $\Leftrightarrow$
orientation(a,b,c) $\neq$ orientation(a,b,d)
$\wedge$ orientation(c,d,a) $\neq$ orientation(c,d,b)

6.2.5 Problem: Konvexe Hülle

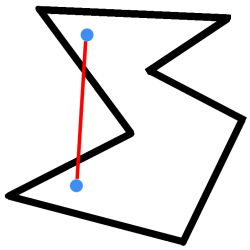
Eingabe: Liste von Punkten `list<point> L;`

Ausgabe: kleinstes Polynom P, das alle Punkte aus L enthält.

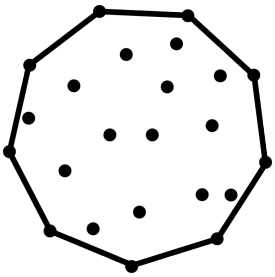
Konvex



Nicht konvex



Menge S kovex $\Leftrightarrow \forall\ a, b \in S : \overline{ab} \subseteq S$



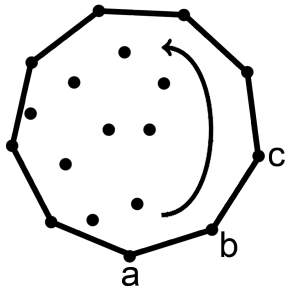
Man kann es sich wie ein Gummiband vorstellen, das außen um eine Menge an Nägeln gelegt wird.

Menge S kovex $\Leftrightarrow (a, b \in S : \overline{ab} \subseteq S)$
$\rightarrow$ Polygon $p = CH(L)$

Die Ecken vom Polygon p sind Punkte aus L.

Ausgabe: Ecken von P gegen den Uhrzeigersinn sortiert → Rand von P

Funktion: `list<point> convexHull(list<point>& L)`



konvex $\Leftrightarrow a.orientation(b,c) > 0$
(a,b,c machen einen links-Knick)

Einfacher Algorithmus: Gift-Wrapping

Im $\mathbb{R}^2$ einwickeln mit einer Schnur.

$L = \{q_1, .., q_n\}$ mit Ecken $p_1, \ldots$

1.) Startpunkt $p_1 \leftarrow min_{xy}(L)$ (Punkt unten links).

2.) Wie findet man p_2?
Betrachte horizontalen Strahl (l) nach rechts, der in p_1 startet.
Drehe l so lange gegen den Uhrzeigersinn bis er auf einen Punkt von L trifft.
$p_2 \leftarrow$ der Punkt q auf l mit max $dist(p_1, q)$ } *

3.) Wiederhole Schritt 2 mit Ecke p_2 bis wir wieder bei p_1 angelangt sind.

* Lineare Suche in L nach der nächsten Ecke

```
list<point> ConvexHull(list<point>& L){
   list<point> CH; // Resultat
   // Startpunkt (Ecke) -> lineare Suche
    point q0 = L.first();
    point p;
    forall(p,L){
      if(p.cmp_xy(q0) == -1){
         q0 = p;
      }
   }
   CH.append(q0);
   L.del(q0);
   while(true){
      point q = L.first()
      forall(p,L){ // lineare Suche nach max. Winkel
         if(CH.last().orientation(q,p) == -1
            || ((CH.last().orientation(q,p) == 0)
            && CH.last().cmp_dist(q,p) == -1)){
            q = p;
         }
      }
      L.del(q);
      if(q == q0) break;
      CH.append(q);
   }
   return CH;
}
```

orientation(p,q,r) und

$$cmp\_xy(a,b) = \begin{cases} +1 & a >_{xy} b \\ 0 & a =_{xy} b \\ -1 & a <_{xy} b \end{cases} \text{ und}$$

$$cmp\_dist(a,b,c) = \begin{cases} +1 & dist(a,c) > dist(a,b) \\ 0 & dist(a,c) = dist(a,b) \\ -1 & dist(a,c) < dist(a,b) \end{cases} \text{ stehen zur Verfügung.}$$