DD2458, Problemlösning och programmering under press

Föreläsning 11: Beräkningsgeometri

Datum: 2007-11-27
Skribent(er): Joel Palmert, Erik Markland, Christian Adåker
Föreläsare: Mikael Goldmann

I nedanstående sektioner beskrivs lösningar på ett antal specifika problem som har generella tillämpningar. Som alltid är det en fördel om man kan göra beräkningarna med heltal i stället för att använda de reella tal som oftast är naturliga utifrån matematiken de beskrivs med.

Jämför till exempel:

- a och b i stället för $\sqrt{a}$ och $\sqrt{b}$
- a och b i stället för $\arctan a$ och $\arctan b$
- $a \cdot b$ och $c \cdot d$ i stället för a/d och c/b

1 Polygoner

1.1 Area

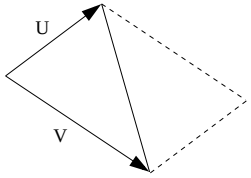
1.1.1 Triangel

Hur det är lämpligt att beräkna arean av en triangel beror på vilken form den är given. Ofta är det lätt att ta reda på två vektorer, $U = (x_1, y_1)$ och $V = (x_2, y_2)$, som representerar sidor i triangeln. Från linjär algebra kommer vi ihåg att determinanten av matrisen som har dessa vektorer som rader har samma belopp som arean av det parallellogram som spänns up av de båda vektorerna. Denna area är dubbelt så stor som triangeln som spänns up av samma vektorer så

$$2A_{\text{Triangel}} = \text{abs}\left(\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix}\right) = |x_1 y_2 - x_2 y_1|$$

Tecknet på denna determinant kan också användas. Om determinanten är positiv betyder det att vi valde punkterna (x_1, y_1) och (x_2, y_2) i motsols (positiv) riktning i triangeln. Om vi hade valt dem i medsols (negativ) riktning hade determinanten blivit negativ.

Figur 1:



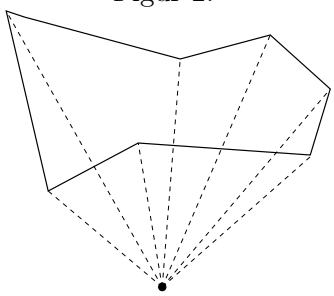
Beräkningen av parallellogram och därmed delar av sådana kan generaliseras till $\mathbb{R}^n$ då man räknar ut hyperarean av området som spänns upp av n st. vektorer.

1.1.2 Area för enkla polygoner

För att räkna ut arean av en enkel polygon[1] väljer vi först en godtycklig punkt. Sen räknar vi för varje kant ut area på den triangel som är bestämd av kanten och punkten vi valde. Tecknet här är av betydelse så det är viktigt att vi traverserar polygonens hörn i samma ordning hela tiden. Vi summerar nu alla dessa areor med olika tecken vilket ger hela polygonens area. Om polygonen traverseras motsols kommer arean att bli positiv.

$$A = \frac{1}{2}\begin{vmatrix} - & p_{n-1} & - \\ - & p_0 & - \end{vmatrix} + \frac{1}{2}\sum_{i=1}^{n-1}\begin{vmatrix} - & p_{i-1} & - \\ - & p_i & - \end{vmatrix}$$

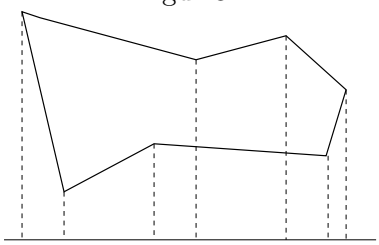
Figur 2:



I stället för att välja en godtycklig punkt kan vi välja en punkt nere i oändligheten men nu bara beräkna arean som finns mellan x-axeln och kanterna. Detta gör att areorna vi vill beräkna blir parallelltrapetser i stället för trianglar.

$$A = \frac{1}{2}(x_0 - x_{n-1})(y_0 + y_{n-1}) + \frac{1}{2}\sum_{i=1}^{n-1}(x_i - x_{i-1})(y_i + y_{i-1})$$

Figur 3:



1.1.3 Area för polygoner med heltalskoordinater

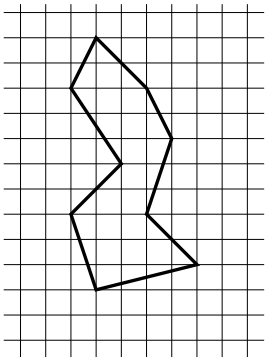
Om hörnen i en polygon P har heltalskoordinater eller om P kan transformeras så att detta gäller finns **Picks sats** som ger en annan identitet för arean:

[1] För polygoner som inte är enkla är det oklart hur man vill definiera arean. Vi begränsar oss därför till enkla polygoner.

Låt I vara antalet heltalspunkter som ligger strikt inuti P, R antalet heltalspunkter som ligger på randen till P och A arean av P. Då gäller $A = I + R/2 - 1$.

I figuren nedan är $I = 20$, $R = 12$ och $A = 25$.

Figur 4: Polygon med heltalskoordinater.



Vi vet redan ett sätt att beräkna arean så detta ger oss ett samband mellan I och R.

1.2 Punkt i polygon

Vi vill ta reda på om en punkt p ligger i en given polygon P. Det finns två enkla sätt att lösa detta.

1.2.1 Antalet skärningar

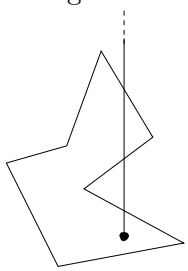
Om punkten ligger inuti polygonen kommer en linje L som dras från punkten rakt upp oändligt långt att skära polygonen ett udda antal gånger (se figur 5).

I pseudokoden nedan används funktionen INTERSECTS(L_1, L_2) som är sann om L_1 korsar L_2. Funktionen beskrivs närmare i 2.3. För att det ska stämma om vår linje L passerar genom ett hörn i polygonen så kommer intersects att betrakta linjers högra men inte vänstra ändpunkt som en del av linjen.

Algoritm 1: Algoritm för att se om en punkt ligger i en polygon.
Input: En polygon P och en punkt p.
Output: Huruvida p ligger inuti P.

```
INSIDE(P, p)
(1)   isIn ← false
(2)   L ← [p, (p_x, ∞)]
(3)   foreach edge e of P
(4)     if INTERSECTS(e, L)
(5)       isIn ← not isIn
(6)   return isIn
```

Figur 5:



1.2.2 Antalet cirklingar

En annan metod går ut på att vi går igenom polygonen längs dess rand i någon riktning och summerar vinkelskillnaderna mellan P_i, p och P_j för varje kantvektor $P_i \to P_j$. Om denna summa blir $2k\pi$ så går polygonen k varv runt punkten p. Om summan är 0 är punkten alltså utanför polygonen. Summan blir alltid en multipel av 2π.

$$2k\pi = \sum_{i=2}^{n} \angle(P_i, p, P_{i-1}) + \angle(P_1, p, P_n)$$

Den här metoden kommer även att ge rätt svar för polygoner som inte är enkla.

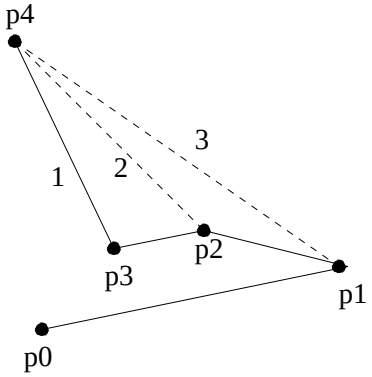
1.3 Konvext hölje

Ett konvext hölje för en mängd punkter är den minsta konvexa polygon så att alla punkter i mängden innesluts. Detta kan beräknas med en metod som kallas Graham Scan.

Algoritm 2: Grahams algoritm för konvexa höljen.
Input: En mängd punkter P.
Output: Den minsta konvexa polygon som innehåller alla punkter i P.
GRAHAM SCAN(P)

(1) $p_0 \leftarrow$ den punkt i P med lägst y-koordinat (och lägst x-koordinat om det finns flera val).
(2) Sortera de övriga punkter med avseende på vinkeln till x-axeln, sett från p_0. Om två har samma vinkel, ta bort punkten närmast p_0.
(3) Lägg p_0, p_1, p_2 på en stack.
(4) **for** i=3 **to** $n-1$
(5) PUSH(p_i)
(6) **while** stackens tre översta punkter inte bildar vänstersväng
(7) Ta bort näst översta elementet i stacken.

Figur 6: Illustration av Graham scan.



Vänstersväng defineras i det här fallet som att arean av p_{i-2}, p_{i-1}, p_i är > 0. Om man tittar på exemplet i figur 6 ser man att att flera punkter redan är adderade. Men man har nu kommit till punkten p_3 där man upptäcker att man måste ta en högersväng längs linje 1. Därmed vet man att p_3 inte ska ingå i höljet, så den plockas bort och linje 2 kommer att väljas istället. Även här är det en högersväng och istället så kommer slutligen linje 3 att väljas.

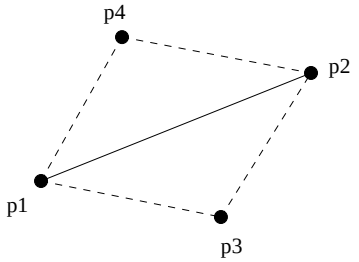
Sorteringen av punkterna i Graham Scan tar tid $O(n \log n)$. I algoritmen kollar vi om vi måste göra en vänstersväng och en sådan koll riskerar att ta $O(n)$ tid. Men detta kan inte hända ofta. Mer specifikt kommer koden i den inre while-loopen bara köras totalt $O(n)$ gånger. Det kommer förstås for-loopen också att göra så tiden för själva svepet är $O(n)$. Tiden för hela algoritmen domineras därför av sorteringen och blir $O(n \log n)$.

2 Linjer

2.1 Orientering av en punkt

Triangelareor kan även användas för att bestämma huruvida en punkt ligger till vänster eller höger om en linje, i linjens rikting. Om man bildar triangeln $p_1p_3p_2$ och sedan beräknar dess area så kommer den att bli positiv eftersom, som tidigare

Figur 7: Punkter på var sin sida om en linje.



nämnts, triangelns hörn ligger motsols. Det innebär att p_3 ligger till höger om linjen. Om man däremot gör samma sak för triangeln $p_1p_4p_2$ så kommer den att få en area

med negativt tecken eftersom triangeln ligger medsols. Punkten p_4 ligger således till vänster om linjen. Om arean i någon av beräkningarna skulle bli noll så ligger alla tre punkter på samma linje.

2.2 Punkt på linjesegment

Vi är intresserade av huruvida en punkt q ligger på ett linjesegment med ändpunkter p_1 och p_2. Det första vi kan undersöka är arean av triangeln p_1qp_2 (enligt ovan). Om den är nollskild så ligger q inte ens på linjen mellan p_1 och p_2.

När vi vet att q ligger på rätt linje så kan vi enkelt undersöka om $\text{DIST}(p_i, q) \leq \text{DIST}(p_1, p_2)$. Om båda villkoren är uppfyllda så ligger q mellan p_1 och p_2.

Ett krav för detta är naturligtvis att $p_1 \neq p_2$ (annars blir arean alltid 0). Fallet $p_1 = p_2$ är trivialt.

2.3 Korsande linjesegment

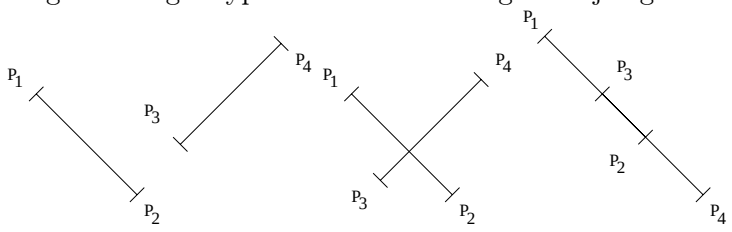
Antag att vi har två stycken linjesegment, det ena med ändpunkterna p_1, p_2 och det andra med ändpunkterna p_3, p_4. Vi är intresserade av om de skär varandra i någon punkt.

Att faktiskt hitta skärningspunkten mellan två linjer innebär i det allmänna fallet att lösa ett linjärt ekvationssystem. Det kan innebära problem om linjerna är näst intill parallella, eftersom skärningspunkten kan komma att ligga väldigt nära oändligheten. Om man däremot bara vill veta om de skär varandra så finns det andra metoder.

En enkel algoritm är att se efter om båda segmenten har sina ändpunkter på varsin sida om det andra segmentets linje. Om varje intervall har sina ändpunkter på varsin sida om det andra, så måste segmenten skära varandra. Enligt metoden ovan undersöker vi triangelareor för att undersöka vilken sida punkterna ligger. Villkoret blir $\text{SIGN}(A_1) \neq \text{SIGN}(A_2)$ [2] (ett test per linje och båda måste uppfyllas).

Ett specialfall som kan uppstå är om alla areor blir noll, då ligger båda intervallen på samma linje. Det fallet är i princip ett endimensionellt problem och man behöver bara se om ett av segmenten har en ändpunkt inom det andra (också enligt ovan).

Figur 8: Några typiska fall för skärning av linjesegment.



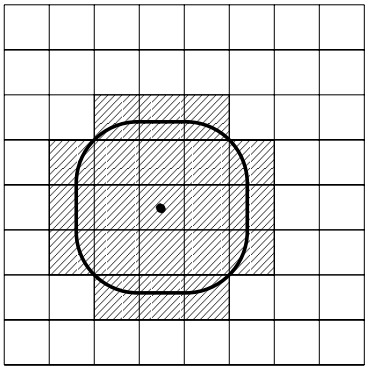
(a) korsar inte (b) korsar (c) specialfall

[2] vi definierar $\text{SIGN}(0) = 0$

3 Kortaste avståndet mellan punkter

Ur en mängd punkter vill vi hitta det kortaste avståndet mellan ett par av punkter. För detta ändamål finns två algoritmer som lämpar sig för olika typer av indata. Om punkterna är likformigt fördelade på en yta finns en algoritm som går i tid $O(n)$ för att hitta det kortaste avståndet.

Figur 9: Punkter på avståndet max $\sqrt{2}d$.



Idén är att vi börjar med att stoppa alla punkter i $n/2$ kvadrater av dimension $d \times d$ i ett tvådimensionellt plan beroende på var de befinner sig. Eftersom det måste finnas någon kvadrat med minst två punkter kommer det sökta minstaavståndet vara mindre eller lika med diagonalen i rutorna: $d_m \leq \sqrt{2}d$. Vi jämför sedan alla punkter med endast de punkter som ligger i rutor som är nära den aktuella punktens ruta. Vi returnerar det kortaste avståndet. I figur 9 är det område markerat där det skulle kunna finnas punkter på minimiavstånd. För att kolla alla dessa punkter kollar vi alla punkter i rutorna som berörs av gränsen. Dess rutor är också markerade. Om punkterna är likformigt fördelade så kommer det i genomsnitt att finnas $O(1)$ punkter i varje ruta och specifikt i de 21 närliggande rutorna markerade i figuren så arbetet för varje punkt är konstant. Alltså är hela arbetet $O(n)$.

Detta gäller inte om punkterna är allt för skevt fördelade. För det allmänna fallet finns en dekompositionsalgoritm som går i $O(n \log n)$.

Börja med att sortera punkterna på x-koordinat. Dela upp planet i två delar och finn det minimala avståndet i varje del. Sen måste vi också kolla om det finns några par av punkter som båda ligger nära linjen vi delade upp planet i som är nära varandra. Vi utnyttjar också att vi bara behöver undersöka par av punkter som är så nära varandra i y-led att vi vet att det inte kan finnas fler än fyra punkter där.

Algoritm 3: Algoritm för kortaste avståndet mellan två punkter.
Input: En punktmängd S, sorterad på x-koordinat.
Output: d, det minsta avståndet mellan två (skilda) punkter i S.

```
MINDIST(S)
(1)   if |S|=2
(2)     SORT(S) //i y-led
(3)     return DIST(S[0], S[1])
(4)   d_1 ← MINDIST(S[0, ..., n/2]) //dela upp punktmängden i två halvor
(5)   d_2 ← MINDIST(S[1 + n/2, ..., n − 1])
(6)   d ← MIN(d_1, d_2)
(7)   foreach p i S[0, ..., n/2], i stigande y-ordning
(8)     if p_x är max d ifrån uppdelningen
(9)       Jämför nu p med alla värden, i den halva som p
          inte kom från, som är tillräckligt nära p i y led.
          Eftersom p är den koordinat längst ner som inte
          ännu är testad behöver vi bara titta uppåt. I den
          lilla kvadrat d × d där de möjliga värdena kan
          vara får det plats högst fyra punkter eftersom de
          måste var på avstånd minst d från varandra. Om
          vi hittar något par som är på avstånd mindre än
          d så ersätter vi d.
(10)  S = MERGE(S[0, ..., n/2], S[1 + n/2, ..., n]) //på y
```

Figur 10: Bara ett fåtal punkter ligger på avstånd $\leq d$.