DET KONVEXA HÖLJET OCH ALGORITMEN QUICKHULL

Philip Arvidsson (S133686@student.hb.se)
Systemarkiktekturutbildningen (ASYAR14h), University of Borås

Färdigställd den 30:e maj 2015

Algoritm, konvexa höljet, quickhull, komplexitet

1. Inledning

Det konvexa höljet är enkelt att förklara på ett lättbegripligt vis: Vi kan till exempel börja med att föreställa oss en nåltavla (Levitin, 2011, se fig.1 t.h.) där man får placera nålar som man önskar och fästa dem genom att sticka dem rätt in i tavlan. När vi satt upp så många nålar som vi vill, tar vi en gummisnodd och fäster runt alla de yttersta nålarna, så att alla nålar på tavlan är innanför gummisnoddens omkrets. Gummisnodden utgör nu det konvexa höljet.

Mer matematiskt kan problemet förstås på följande sätt: Vi har ett antal punkter som alla ligger på samma plan. Det konvexa höljet utgörs av den serie linjesegment på planet som omsluter alla punkterna. Vinkeln mellan två givna segment får aldrig vara lägre än 180°; då är höljet inte längre konvext.

I denna rapport redogör vi för problemet och tittar på ett par algoritmer för att lösa det, samt analyserar algoritmerna. Det är viktigt att dessutom ha i åtanke att en teoretiskt mer effektiv lösning (av lägre komplexitet) inte nödvändigtvis är överlägsen i praktiken. Även andra aspekter måste begrundas, så som reell tid för exekvering, minneskrav med mera.

Avslutningsvis går vi igenom det inkluderade demoprogrammet som kan användas för att visualisera det konvexa höljet i form av ett interaktivt sandbox-läge, samt för att jämföra och testa de olika algoritmernas effektivitet genom körning i benchmark-läge.

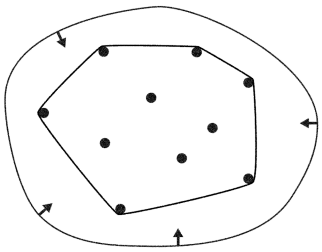

Fig. 1.

2. Algoritmerna

Nedan går vi igenom algoritmerna uttömmande sökning och quickhull, som ligger i fokus i denna rapport. Uttömmande sökning inkluderas endast som referenspunkt för att förstå hur effektiv quickhull-algoritmen är.

Uttömmande sökning (*eng. bruteforce*) innebär att vi går igenom alla möjliga lösningar, för att till slut hitta den korrekta lösningen. Det blir uppenbart att en bruteforcealgoritm sällan är särskilt effektiv. Algoritmen jobbar enligt följande:

1. Välj en punkt ur punktsamlingen.
   $p \in P$ *där* $P \subset \mathbb{R}^2$
2. Välj en till, annan punkt bland alla punkter.
   $q \in P$ *så att* $q \neq p$
3. Skapa en linje mellan dessa två punkter.
   $L = \overline{pq}$
4. Kontrollera att alla punkter ligger på

samma sida om linjen.

$f(x) = \overline{px} \cdot L_{\perp}$

$f(x) \geq 0$ *för alla* $x \in H$ *där* $H \subseteq P$

5. Om alla punkter ligger på samma sida så har vi hittat ett segment i höljet, varpå vi lägger till det i en lista, går till steg 1 och väljer en ny punkt. Annars går vi bara till steg 2, utan att spara segmentet i listan.

   $H = P \Rightarrow L$ *är ett segment i höljet*

Efter att ha tillämpat algoritmen ovan på alla möjliga par av punkter i punktsamlingen innehåller listan det konvexa höljets segment i form av en serie linjer.

Algoritmen går *inte* att förenkla sett till sin komplexitet[1]. Uttömmande sökning går ut på att generera alla möjliga lösningar och verifiera vilka som är riktiga. När det gäller konvexa höljet innebär detta att vi först måste generera alla möjliga par av punkter. Med n punkter kan vi först välja en av n punkter, varefter den andra punkten i paret kan väljas bland resterande $n-1$ punkter. Detta innebär att vi för n punkter har $n \cdot (n-1)$ möjliga par. Vi verifierar lösningen genom att titta på en punkt åt gången och ser vilken sida av linjen (mellan de två punkterna) de ligger på. Då vi måste göra detta för varje möjlig lösning, så får vi $n \cdot (n-1) \cdot n$ operationer.

Skriven i programkod (se fig. 2 t.h.) är det lättare att förstå algoritmens arbetssätt och komplexitet. Vi noterar att det finns tre stycken loopar inuti varandra. Den första och sista loopen arbetar fram till samma gränsvärde. Den mittersta loopen utför en iteration färre än de andra två. Detta innebär att vi teoretiskt får $n \cdot (n-1) \cdot n$ iterationer. Vi ska nu titta på programkoden:

```
numLines = 0;
for (int i=0; i < numPoints; i++)
{
  for (int j=(i+1); j < (i+numPoints); j++)
  {
    pointT *a, *b, *c;

    bool outside = FALSE;
    for (int k=0; k < numPoints; k++)
    {
      a = points[i];
      b = points[j % numPoints];
      c = points[k];

      float d = (b->x-a->x) * (c->y-a->y)
              - (b->y-a->y) * (c->x-a->x);
      if (d < 0.0f)
        outside = TRUE;
    }

    if (!outside)
    {
      lines[numLines].a = a;
      lines[numLines].b = b;
      numLines++;

      if (numLines >= maxLines)
        return;
    }
  }
}
```

Fig. 2.

Uträkningen av värdet på variabeln **d** får anses vara den kritiska operationen då det sker en gång för varje iteration i den innersta loopen. Detta ger algoritmen komplexiteten $O(n^3)$. D.v.s kubisk komplexitet, vilket inte alls är särskilt effektivt.

Algoritmen quickhull använder sig istället av en metod som kallas divide-and-conquer, som går ut på att dela upp problemet i mindre delproblem och lösa dem separat. Denna uppdelning görs om och om igen tills ett *basfall* nås (där uppdelning i mindre delproblem inte längre är möjligt). Basfallet löses och används sedan för att lösa större och större delproblem, till dess att den ursprungliga problemställningen är löst.

Quickhull arbetar såhär:

1. Hitta extrempunkterna längs x-axeln.

[1] Algoritmens *best case* går att förbättra till $O(n^2)$. Se implementation i filen bruteforce.c.

D.v.s den punkt som ligger längst till vänster, samt den punkt som ligger längst till höger.
2. Bilda en linje mellan de två punkterna. Resterande punkter kan nu delas upp i två separata samlingar; en för varsin sida om linjen.
3. För de båda samlingarna med punkter (på varsin sida om linjen): Hitta den punkt som är längst från linjen.
4. Bilda en triangel med de tre punkterna.
5. Välj de två punkterna i triangelns vänstra sida och gå till steg 2.
6. Välj de två punkterna i triangelns högra sida och gå till steg 2.

Efter att ha utfört algoritmen har vi samlat på oss alla punkter i höljet. Vi kan nu sammankoppla dem med varandra för att bilda höljets segment.

Implementationen av quickhull är långt mer komplex än den för bruteforce, varför den inte är inkluderad här. Se filen algorithms.c i src-mappen för implementationsdetaljer.

3. Designval

Programspråket C användes vid implementering av algoritmerna, då det specificerades som krav i uppgiften.

Quickhull implementerades med en hjälpfunktion för rekursionen, vilket förmodligen är nödvändigt med tanke på att det första steget i algoritmen skiljer sig något från resterande. D.v.s. vi behöver en yttre funktion som utför de initiala operationerna innan vi går ned i rekursionen.

Punkterna lagras i minnet i vektorer innehållandes element av typen **`pointT`**, som i själva verket är en 2-vektor (x- och y-komponenter). Höljet lagras i en typ (**`hullT`**) som innehåller en serie linjer. Varje linje (**`lineT`**) består av två pekare – en till vardera ändpunkt.

Ett återkommande problem med divide-and-conquer-algoritmer är minnesallokeringarna. Varje nivå i rekursionen kräver ofta nya allokeringar. Quickhull är inget undantag, utan kräver i vanlig ordning flera allokeringar för att ha reda på delproblemen. För att komma runt detta problem skapades en köstruktur (**`queueADT`**) som punktvektorerna lagras i. När quickhull behöver en ny vektor så tittar den först i kön. En ny vektor skapas endast om kön är tom. När algoritmen är klar med vektorn returneras den till kön, så att den kan återanvändas i en annan nivå eller körning. På detta vis kommer vi runt problemet med att ständigt utföra nya allokeringar. Detta ger en signifikant prestandaökning om ca 10%.

4. Experimentplanering

För att få en god insikt i algoritmernas effektivitet kommer ett stort antal körningar att göras. Experimentanalysens grund finner vi i antalet punkter (input-storlek n), som är den datapunkt vi utgår ifrån för att ställa upp resultatet.

För varje datapunkt (antal punkter) kommer algoritmerna att köras ett hundra gånger. Varje iteration (av de hundra) får en ny, slumpmässig uppsättning med punkter (med samma antal punkter). Vi räknar sedan ut snittvärden för antal kritiska operationer samt tidsåtgång, vilka utgör de grafer vi ritar upp för att analysera och förstå algoritmerna bättre. Därefter ökar vi antalet punkter och upprepar körningen. Totalt kommer algoritmerna att köras i spannet 10 till 1000 punkter, med 10 punkters ökning. Detta ger 100 datapunkter.

Punkternas positioner genereras enligt följande:

$p_x = r \cdot \cos a$, $p_y = r \cdot \sin a$ där $a \in [0...2\pi)$ och $r \in [0...1)$. Båda variablerna slumpas med hjälp av standardbiblioteket i C; funktionen `rand()`. [2]

5. Resultat och analys

Vi börjar med att titta på antalet kritiska operationer. De skiljer sig markant mellan de olika algoritmerna. Detta ger dock inte en komplett bild av skillnaden mellan algoritmerna i praktiken då den kritiska operationen är vald med viss subjektivitet samt ej avsedd för jämförelse algoritmer emellan. Quickhull-algoritmen innehåller dessutom så många olika delmoment att det krävs flera kritiska operationer för att ge en någorlunda rättvisande bild.

Nedan använder vi den kritiska operationen både för att jämföra algoritmerna (vilket vi sedan även gör med tidsmätning, något som ger en mer rättvisande bild) samt för att förstå deras komplexitet, för vilket den kritiska operationen är avsedd.

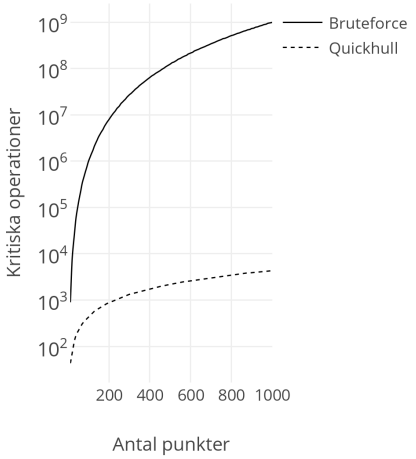

Fig. 3.

Antalet operationer ser ut att stämma överens med algoritmernas respektive teoretiska komplexiteter. För att kunna studera kurvornas utveckling närmare tittar vi på algoritmerna var för sig, på linjära skalor. Vi börjar med bruteforce-algoritmen:

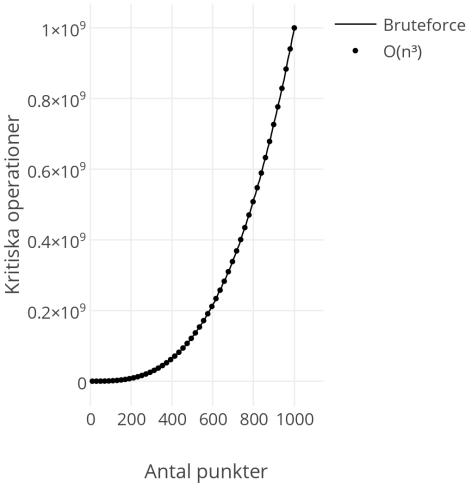

Fig. 4.

Den punktmarkerade kurvan ovan följer funktionen $f(x) = x^3$ vilket visar sig stämma överens exakt med empirisk data. Detta innebär att uttömmande sökning (som väntat) är av komplexiteten $O(n^3)$.

Vi fortsätter med quickhull-algoritmen och finner något anmärkningsvärt:

[2] Detta är ändrat i implementationen sedan experimenten utfördes, då den typen av slumpgenerering fungerar dåligt med sandbox-läget. Istället slumpas punkternas positioner längs x- och y-axlarna direkt.

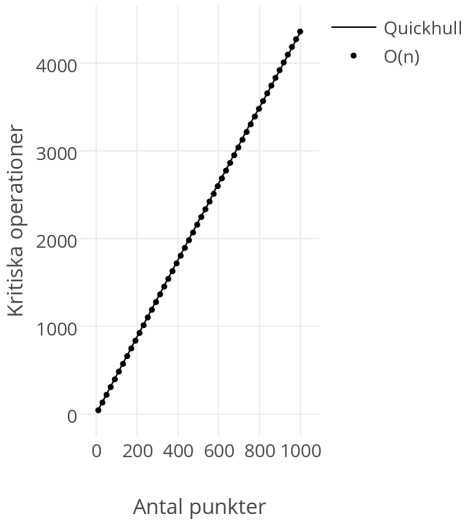

Fig. 5.

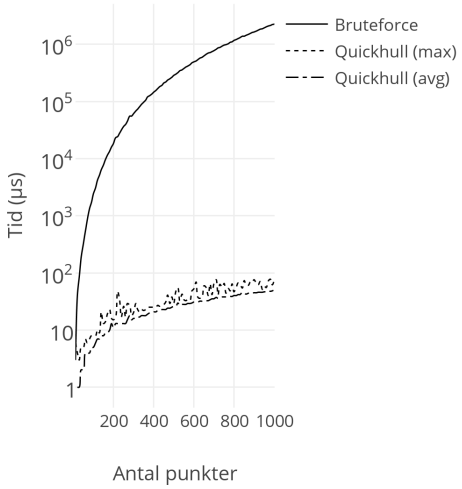

Fig. 6.

Den punktmarkerade linjen följer nu funktionen $f(x) = x$ vilket återigen ger en mycket exakt passform. Vi noterar alltså en linjär komplexitet för quickhull: $O(n)$.

Härnäst jämför vi algoritmerna på ett mindre teoretiskt sätt, nämligen genom att mäta den tid det tar för algoritmerna att exekvera för olika inputs. Detta ger en mer rättvisande bild av algoritmernas tillämpning i praktiken.

Vi börjar med att betrakta hur mycket tid de olika algoritmerna behöver på sig för att lösa problemet[3]:

Notera att vi vid låga n får så mycket *overhead* (i form av funktionsanrop, minneshantering m.m.) att quickhull kan vara långsammare än uttömmande sökning.

Algoritmerna var för sig, på linjär skala:

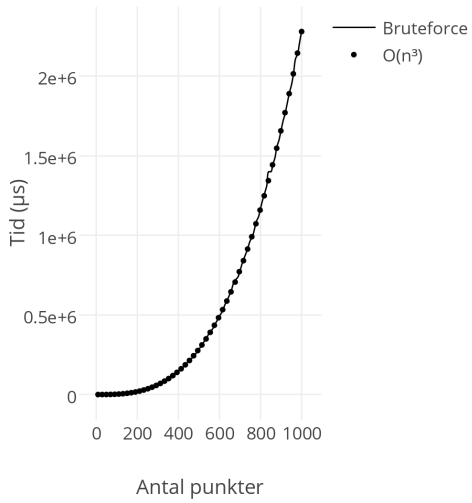

Fig. 7.

[3] Då uttömmande sökning inte är avhängig input utan jobbar på samma sätt hela tiden, har vi endast en kurva i grafen för bruteforce-algoritmen. Snitt och max är en och samma kurva.

Detta stämmer överens väl med komplexiteten för antalet kritiska operationer för uttömmande sökning. Utvecklingen visar sig vara av kubisk komplexitet även här.

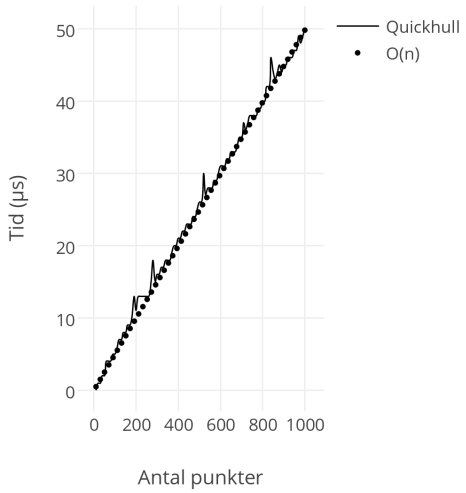

Fig. 8.

Den punktmarkerade linjen stämmer väl överens med algoritmens tidsåtgång, och quickhull visar sig än en gång vara linjär för slumpmässiga inputs.

6. Slutsatser/diskussion

Efter att ha analyserat datan framgår att quickhull – i enlighet med Overmars & van Leeuwen (1980) – är linjär för slumpmässiga uppsättningar med punkter. Detta kan verka anmärkningsvärt eftersom quickhull – i likhet med ex. quicksort (som ju är av komplexitet $O(n \cdot log_2\, n)$ i *average case*) – förefaller dela upp problemet i två delar (rekursivt) och hanterar dessa var för sig. I själva verket delar quickhull upp problemet i *fyra* delar. Detta förklarar den linjära komplexiteten. Vi kan illustrera detta med en bild som beskriver upp- delningen i mindre delproblem och sedan tillämpa Master Theorem (Cormen, Leiserson, Rivest & Stein, 2001).

Notera hur trianglarna vi bildar i algoritmen kan delas upp i fyra olika sektioner, $\mathbf{S_0}$, $\mathbf{S_1}$, $\mathbf{S_2}$ och $\mathbf{S_3}$. Punkterna **A** och **B** är de punkter vi bygger algoritmens skiljelinje mellan. Punkten **P** är den punkt av alla punkter ovanför linjen, som är längst bort från den.

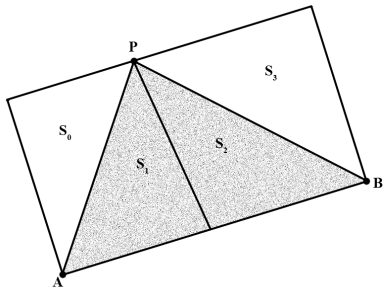

Fig. 9.

De fyra trianglarna kommer i snitt att ha samma area om alla punkter i uppsättningen är slumpmässiga. Detta innebär att vi i själva verket delar upp problemet i fyra delproblem, kastar bort två av dem och löser resterande två fjärdedelar var för sig, rekursivt. Vi tillämpar nu Master Theorem:

$$T(n) = aT(n/b) + f(n) \text{ där } f(n) \in \Theta(n^k)$$
$$a < b^k \Rightarrow T(n) \in \Theta(n^{k)}$$
$$a = b^k \Rightarrow T(n) \in \Theta(n^k \cdot log_2\, n)$$
$$a > b^k \Rightarrow T(n) \in \Theta(n^{log_b\, a})$$

Vi har $a = 2,\ b = 4,\ k = 1$ (där a är antalet delproblem som ska lösas och n/b är storleken på de mindre problemen) d.v.s. $a < b^k$ vilket ger $T(n) \in \Theta(n^{k)}$, vilket innebär att quickhull är av linjär komplexitet (för slumpmässiga inputs) precis i enlighet med våra observationer.

7. Referenser

[1] Levitin, A., 2011. *The Design & Analysis of Algorithms*. 3rd ed., Upper Saddle River: Pearson Education, pp.110–111

[2] Overmars, M.H. & van Leeuwen, J. 1980, *Further comments on Bykat's convex hull algorithm*. Information Processing Letters, vol. 10, no. 4/5, pp.209–212

[3] Cormen, T., Leiserson, C., Rivest, R. & Stein, C., 2001, *Introduction to Algorithms*, 2nd ed., pp.73–90

[4] Devroye, L. & Toussaint, G., 1981, *A note on linear expected time algorithms for finding convex hulls*, Computing, vol. 26, pp.361–366

8. Demoprogram

Börja med att starta programmet. Du blir tillfrågad hur många punkter du vill ha i punktsamlingen. Vid körning i sandbox-läge kan prestandan testas med högt antal (1000–10,000), men den visuella pedagogiken blir tydligare vid ett lägre antal (5–15). Om programmet ska köras i benchmark-läge är det lämpligt med ett högt antal (5000–10,000) för mer precis mätning av exekveringen.

Efter att du angett antal punkter frågar programmet om du vill köra det i benchmark-läge. Om du vill testa algoritmernas prestanda svarar du med "yes", sedan enter. Vill du köra det i sandbox-läge trycker du bara enter utan att ange något.

I benchmark-läge körs programmet i trettio sekunder och mäter tiden för att exekvera de olika algoritmerna med det specificerade antalet punkter. Därefter presenteras datan på skärmen.

Om du istället startar programmet i sandbox-läge kan du laborera med olika inställningar och betrakta hur det konvexa höljet förhåller sig till de olika punkterna. Titta i konsolfönstret för att se de olika tillgängliga knapparna du kan använda, samt vilka effekter de har. Tryck ex. på **q** för att byta fram och tillbaka mellan bruteforce och quickhull. Notera hur mycket snabbare quickhull är för större punktsamlingar! Jämför även med Akl-Toussaint-heuristik genom att trycka på **a**.

9. Kommentarer

Tyvärr ska rapporten hållas kort, varför det inte finns utrymme att resonera kring Akl-Toussaint-heuristik och hur den påverkar algoritmernas exekvering. Det visar sig att algoritmer (för att lösa konvexa höljet) som är kvadratiska i *worst case* blir linjära om Akl-Toussaint först tillämpas på punkterna (Devroye & Toussaint, 1981), då förutsatt att punkterna är slumpade.

Experimentera gärna med funktionerna i demoprogrammet. Jämför quickhull med bruteforce, samt slå på och av Akl-Toussaint-heuristik för de båda algoritmerna.

Bruteforce blir långt mycket snabbare med Akl-Toussaint påslaget, men hur mycket snabbare beror bland annat på punkternas spridning.