LINKÖPINGS UNIVERSITET

TDDI16: Datastrukturer och algoritmer

Lab 4: Mönsterigenkänning

Filip Strömbäck, Tommy Färnqvist, Rita Kovordanyi

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2024

2024-08-29

1 Bakgrund

Du har kommit över en mängd filer som innehåller en mängd till synes slumpmässiga punkter. Du tror dock att vissa av dem innehåller gömda meddelanden, och du vill därför skriva ett program som kan hjälpa dig att hitta dessa. Du vet att de gömda meddelandena består av linjesegment, och att minst fyra punkter från varje linje finns med i punktmängden (de två ändpunkterna av linjesegmentet, och två till). För att rekonstruera linjerna vill du alltså hitta alla uppsättningar om fyra eller fler punkter som ligger på samma linje och rita ut dessa.

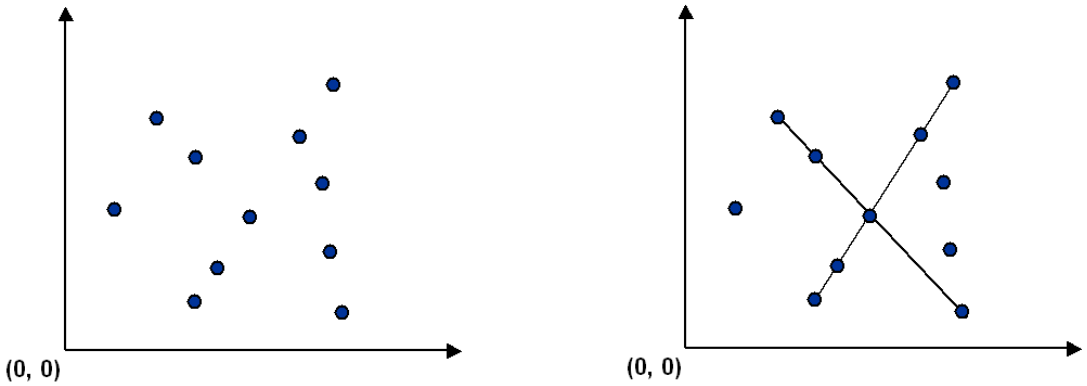
1.1 Tillämpningar

Viktiga komponenter i datorseende är att använda mönsteranalys av bilder för att rekonstruera de verkliga objekt som genererat bilderna. Denna process delas ofta upp i två faser: *feature detection* och *pattern recognition*. I *feature detection* väljs viktiga områden hos bilden ut; i *pattern recognition* försöker man känna igen mönster i områdena. Här får ni chansen att undersöka ett särskilt rent mönsterigenkänningsproblem rörande punkter och linjesegment. Den här typen av mönsterigenkänning dyker upp i många andra tillämpningar som t.ex. statistisk dataanalys.

2 Uppgift

Målet är att skriva ett program som hittar linjesegment i en stor mängd punkter på ett effektivt sätt. Programmet ska läsa punkterna från standard input, och rita ut linjesegmenten i ett fönster.

De givna filerna innehåller programmet `brute` som gör detta. Programmet är dock på tok för långsamt. Det undersöker helt enkelt alla kombinationer av fyra punkter för att se om de ligger på en linje. Om så är fallet ritas linjen ut. En liten optimering görs dock. Om de tre första punkterna som har valts inte ligger på en rät linje letar programmet inte efter en fjärde punkt.



Uppgiften är alltså:

- Undersök programmet `brute` och analysera dess tidskomplexitet. Redogör dina resultat i filen `readme.txt`.
- Implementera sedan en bättre lösning, `fast`, som ska klara av att hitta alla linjesegment bland 128000 punkter på under 2 minuter. Se avsnitt 7 för idéer.
- Analysera `fast` på samma sätt som för `brute`, och fyll i `readme.txt`.
- Jämför energianvändningen av `brute` jämfört med `fast` enligt vad som beskrivs i slutet av `readme.txt`.

3 Indata

Programmen `brute` och `fast` ska läsa indata från standard input. Formatet ser ut som exemplet nedan:

```
0.01
4
10000.0     0.0
 9999.3 10001.2
10000.2 19998.2
10001.0 30000.1
```

Första raden (`0.01`) anger den tolerans, ε, som ska användas vid jämförelser av vinklar. Nästa rad (`4`) antal punkter, n, som finns i resten av filen. Resterande n rader innehåller x- och y-koordinater för de n punkter som ska analyseras. Som syns i exemplet utgörs koordinaterna av flyttal.

4 Jämförelser av flyttal

När man arbetar med flyttal (ex.vis `float`, `double`) är det sällan meningsfullt att göra jämförelsen `x == y`. I och med att vi inte kan lagra oändligt många decimaler så har flyttal begränsad precision. Detta gör ibland att tal som vi betraktar som ”lika” skiljer sig lite i sista decimalplatsen på grund av avrundningsfel i beräkningar. Ett klassiskt exempel på detta är att uttrycket: `0.1 + 0.1 + 0.1 == 0.3` ger resultatet `false`. Detta gäller för de flesta språk som använder flyttal, exempelvis C++, Python, Javascript, med flera.

En annan sak vi måste ta hänsyn till är att indata till programmet inte är exakt. Som vi kan se i exemplet i avsnitt 3 ovan har inte alla x-koordinater exakt samma värde. Detta kan bero dels på mätfel, men också på avrundningsfel i instrumentet som samlade in punkterna från första början.

Lösningen på båda problemen ovan är att inte kräva exakt likhet för att anse att punkterna ligger på en linje. I stället accepterar vi att de skiljer sig med det antal radianer som *tolerans*-värdet anger. Detta kallas ofta för att man använder ett *epsilon* (ε) vid jämförelsen.

Denna jämförelse finns implementeras som funktionen `sameSlope(x, y, tolerance)` i den givna koden. Implementationen kontrollerar i princip `abs(x - y) <= tolerance`, men tar också hänsyn till att 0 radianer är det samma som 2π radianer. Om `sameSlope`-funktionen används kommer punkterna i exemplet i avsnitt 3 anses ligga på samma linje, och exemplet ska därmed producera en lång vertikal linje.

5 Givna filer

Följande filer är givna för labben:

`point.{h,cpp}` Dessa filer innehåller en implementation av en punktdatatyp som används för att lagra punkterna undersöks. Punkten har i vanlig ordning medlemsvariabler för x- och y-koordinater. Punkten har en medlem `slopeTo` som beräknar vinkeln mellan två punkter. Det finns också en medlem `sameSlope` som bestämmer om vinkeln mellan olika punkter är densamma. `sameSlope` finns också som en fri funktion.

`brute.cpp` Implementation av en bruteforce-lösning till problemet. Beskriven i avsnitt 2.

`fast.cpp` Kodskelett där du kan implementera en snabbare lösning till problemet. Koden sköter inläsning av data, utritning av punkter och tidtagning.

`window.h` Klass som innehåller funktionalitet för att rita ut punkter och linjer till fönstret.

`readme.txt` Readme-fil som ska fyllas i med resultaten av din undersökning av `brute`, samt en analys av din snabbare lösning.

`Makefile` Makefil som underlättar kompilering av den givna koden. Kör `make` för att bygga både `brute` och `fast`, eller kör `make brute` eller `make fast` för att bara bygga den ena.

`data/` Innehåller givna testfall, se avsnitt 6 för detaljer.

Notera: Filen `window.h` innehåller två varianter av funktionen `draw_line`:

`draw_line(Point a, Point b)` Ritar en linje mellan de två givna punkterna.

`draw_line(vector<Point> points)` Tar emot en array av punkter. Dessa punkter antas ligga på samma linje. Funktionen undersöker punkterna och ritar *en* linje mellan de punkter som ligger längst ifrån varandra. Denna variant är tänkt att förenkla utritandet av en linje när programmet har hittat ett antal punkter som ligger på samma linje.

Det finns inget krav på vilken av funktionerna som ska användas. Däremot är det antagligen enklare att använda versionen som tar emot `vector<Point>`.

6 Testfall

Mappen `data/` innehåller ett stort antal testfall i form av textfiler som du kan använda för att testa din lösning. Varje indatafil börjar med ett heltal som anger antalet punkter i filen, följt av x- och y-koordinater för punkterna. I och med att programmen förväntar sig att läsa indata från standard input körs testfallen lämpligt enligt följande:

```
./brute < data/input20.txt
```

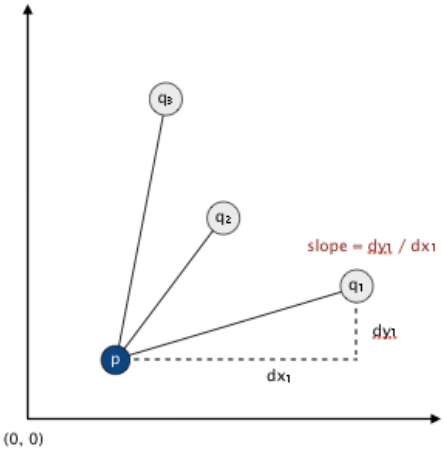
En del av testfallen har en `png`-fil som innehåller den förväntade utdatan från en korrekt lösning. Så länge `brute` inte har modifierats kan även `brute` användas för att generera en referensbild.

Följande fall kan vara intressanta att titta närmare på:

- `cardinal` (punkter längs koordinataxlarna)
- `grid4x4`, `grid5x5`, `grid6x6` (jämför med utdatan från `brute`)
- `input12800.txt` (för att testa prestandan hos er lösning, innehåller 12800 punkter)
- `mystery10089.txt` (bra prestandatest med utfall som är enkelt att verifiera)

7 En snabbare, sorteringsbaserad, lösning

Vi löser problemet mycket snabbare än vad `brute` gör genom att använda oss av sortering. Idén är som följer:



1. Välj en punkt p av alla punkterna i datamängden.
2. Tänk på p som origo.
3. Sortera alla andra punkter efter lutningen de har gentemot p (titta ex.vis på `PolarSorter` för inspiration).
4. Kontrollera om 3 (eller fler) på varandra följande punkter i den sorterade ordningen har tillräckligt lika lutning gentemot p med hjälp av en av `sameSlope`-funktionerna. Om så är fallet ligger dessa punkter, tillsammans med p, i en linje och ska ritas ut.

Dessa steg gör att vi hittar alla linjesegment som går igenom p. Om vi upprepar dessa steg för *alla* möjliga p så hittar vi alla linjesegment.

En komplicerande faktor är att jämförelsefunktionen i den sorteringsbaserade lösningen behöver ändras från sortering till sortering. En variant är att skapa ett funktionsobjekt som överlagrar `operator()` och som kan användas för att göra jämförelser (se `PolarSorter` i `point.h`).