LINKÖPINGS UNIVERSITET

TDDI16: Datastrukturer och algoritmer

Lab 4: Mönsterigenkänning

Tommy Färnqvist, Rita Kovordanyi, Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2018

2018-06-27

1 Bakgrund

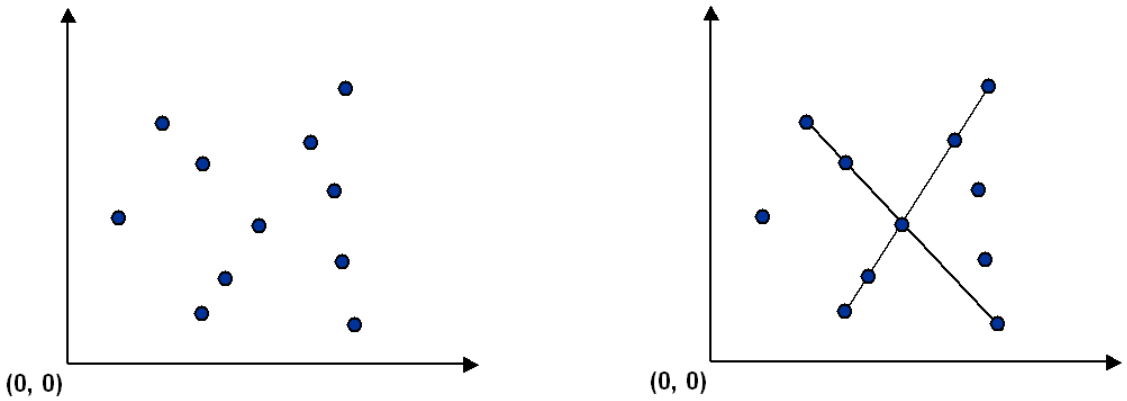
Du har kommit över en mängd filer som innehåller en mängd till synes slumpmässiga punkter. Du tror dock att vissa av dem innehåller gömda meddelanden, och du vill därför skriva ett program som kan hjälpa dig att hitta dessa. Du vet att de gömda meddelandena består av linjesegment, och att minst fyra 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

I de givna filerna finns programmet `brute`, som löser ovanstående problem. Programmet läser in en mängd diskreta punkter i planet från standard input, och hittar alla (maximala) linjesegment som innehåller en delmängd av fyra eller flera av punkterna, vilka ritas ut på skärmen.



Programmet löser problemet genom att göra en totalsökning över alla kombinationer av 4 punkter och för varje kombination kontrollera ifall de ligger på en linje. Om så är fallet ritas ett linjesegment mellan ändpunkterna ut. Detta upprepas tills alla kombinationer har undersökts. En liten optimering som är implementerad är att inte undersöka om 4 punkter är linjärt beroende ifall de 3 första punkterna som har valts ut inte är det.

1. Undersök programmet `brute` och gör en analys av dess tiskomplexitet. Redogör dina resultat i filen `readme.txt`.
2. Programmet `brute` är på tok för långsamt för att kunna användas för större datamängder. Du ska därför implementera en bättre lösning, `fast`, som ska klara av att hitta alla linjesegment i en mängd

med 12800 punkter på under 2 minuter. Se avsnitt 5 för idéer. Analysera också din snabbare lösning på samma sätt som för `brute` och fyll i `readme.txt`.

3. Jämför även energianvändningen av brute jämfört med din lösning enligt vad som beskrivs i slutet av `readme.txt`.

3 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. Punkter kan också jämföras med `<`-operatorn, vilken ger en lexiografisk ordning av punkterna (undersök gärna implementationen). Utöver det finns medlemsfunktionen `slopeTo`, vilken beräknar lutningen från denna punkten till en annan punkt (som en lutningskoefficient, kan vara $\pm\infty$). Till sist finns också den fria funktionen `render_line` som ritar en linje mellan två punkter på skärmen.

`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.

`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 4 för detaljer.

4 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
```

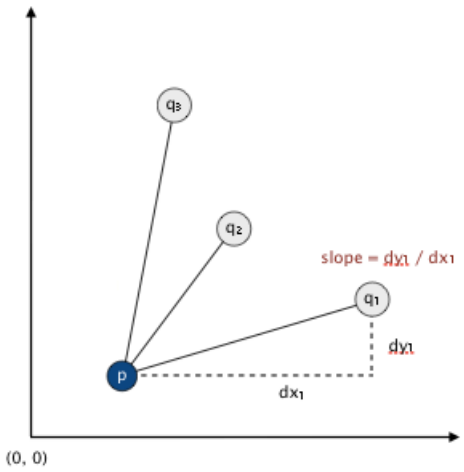
En del av testfallen har en `png`-fil som innehåller den förväntade utdatan från en korrekt lösning. Så länge `brute` inte har modifierats kan även `brute` användas för att generera en referensbild.

Följande fall kan vara intressanta att titta närmare på:

- `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)

5 En snabbare, sorteringsbaserad, lösning

Vi kan lösa 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.
4. Kontrollera om 3 (eller fler) på varandra följande punkter i den sorterade ordningen har samma lutning gentemot p. Om så är fallet ligger dessa punkter, tillsammans med p, i en linje.

Dessa steg gör att vi hittar alla linjesegment som går igenom p, så 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`). Fundera på om sorteringen behöver någon ytterligare egenskap.