Linguaggi di Programmazione

AA 2016-17
Progetto giugno 2017
Consegna 14 giugno 2017, h. 23:59 GMT+1

Line Patterns

Marco Antoniotti e Gabriella Pasi

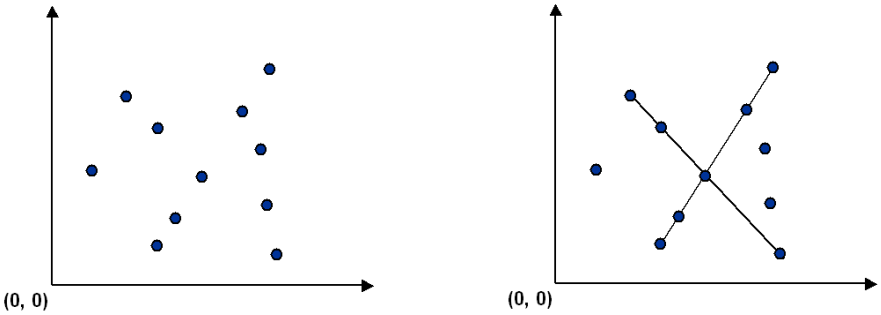
Introduzione

Molti problemi in *computer vision* richiedono di estrarre delle "feature" e dei "pattern" da un'immagine. Un esempio molto semplice riguarda il riconoscimento di "segmenti" da un insieme di punti. Questo problema si presenta in molti altri ambiti applicativi, ad esempio nell'analisi statistica di dati.

Questo progetto è tratto dal corso e dai libri di Sedgewick [S02] (figure incluse).

Il problema

Dato un insieme di N punti (features) **P**, disegnate ogni segmento che comprende almeno 4 punti collineari distinti, come nella figura di seguito[1].



Una soluzione basata sull'"ordinamento"

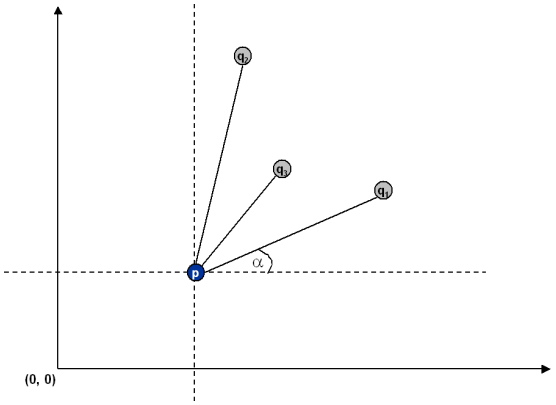
Una soluzione efficiente al problema precedentemente illustrato utilizza un algoritmo di ordinamento sulla base del quale trovare insiemi di punti collineari. Il collo di bottiglia della procedura diventa quindi l'algoritmo di ordinamento. La procedura complessiva è riassunta qui di seguito:

Per ogni punto p.

1. Si consideri p come l'origine.
2. Per ogni altro punto q si calcoli l'angolo che ha con p (rispetto alle coordinate standard; cfr., l'angolo α tra il punto p ed il punto q_1 nella figura dopo quest'algoritmo).

[1] Tutte le figure sono tratte dalle slides dei corsi di Sedgewick su Algoritmi e Strutture Dati di Princeton.

3. Si ordinino i punti rispetto all'angolo calcolato con *p*.
4. Scorrendo l'ordinamento si raccolgano sotto-sequenze, di lunghezza almeno 4, di angoli "uguali"; ovvero di punti collineari.



La figura precedente mostra quattro punti non collineari. Se tutti i quattro punti fossero collineari il coefficiente angolare (l'angolo α) dei segmenti che li collegano a due a due, sarebbe lo stesso.

La complessità di quest'algoritmo è $O(N\,(N + N \lg N + N)) = O(N^2 + N^2 \lg N) = O(N^2 \lg N)$. Perché?

L'algoritmo è molto semplice, ma richiede l'implementazione molto attenta di una serie di primitive geometriche oltre all'algoritmo principale (che abbiamo chiamato *Line Patterns*). Scopo del progetto è di implementare tutte queste primitive e l'algoritmo *Line Patterns* sia in *Common Lisp*[2] sia in *Prolog*.

Preliminari

Le primitive geometriche sono molto complesse da realizzare correttamente e ci sono alcuni caveat da tener presente nel caso generale.

L'interfaccia CL richiesta è la seguente

`make-point` *x y* ⇒ *point-2D*

`x` *point-2D* ⇒ <*integer coordinate*>

`y` *point-2D* ⇒ <*integer coordinate*>

La rappresentazione interna di un punto è lasciata alla vostra immaginazione. Comunque devono valere le seguenti relazioni:

```
cl-prompt> (x (make-point 0 42.01))
0

cl-prompt> (y (make-point 42 123.45))
123.45
```

Per il Prolog le cose sono relativamente più semplici. Potete usare direttamente il meccanismo di unificazione per gestire i punti ed estrarre le loro coordinate. Ovvero, potete usare il termine **`point(X, Y)`** per rappresentare un punto.

A questo punto avete bisogno di primitive geometriche più raffinate. In particolare dovrete implementare un test di collinearità (non per l'algoritmo principale, ma per il debugging). Avrete bisogno delle funzioni/predicati descritti di seguito.

Area di un triangolo

Implementate una funzione `area2` che calcoli l'area di un triangolo, date le coordinate dei vertici. Dati tre punti, *a*, *b*, e *c* (vertici del triangolo) si suggerisce di usare la formula:

[2] *Common Lisp*. Non Scheme o Autolisp o Emacs Lisp o Some Other Lisp!

$$2 \times A(a,b,c) = \Big(\big(x(b) - x(a)\big) \times \big(y(c) - y(a)\big)\Big) - \Big(\big(y(b) - y(a)\big) \times \big(x(c) - x(a)\big)\Big)$$

Notate che il valore può essere negativo (quando i 3 punti sono ordinati in senso orario), il che è in realtà molto utile: se il valore è positivo, allora $\angle abc$ è un angolo a sinistra, se è negativo allora $\angle abc$ è un angolo a destra. Il caso 0 è degenere e capita quando *a*, *b* e *c* sono *collineari*. Il capitolo 1 di [O98] spiega bene tutti i dettagli. Dato che il nostro interesse non è calcolare l'area del triangolo ma capire se un certo angolo è legato a una "svolta a destra" non è necessario fare la divisione per 2 finale.

`area2` *a b c* ⇒ *real*

In Prolog dovrete implementare il predicato

`area2(A, B, C, Area)`

`left`, `left-on`, `is-collinear`, `left/3`, `left_on/3`, `collinear/3`

Data la funzione `area2`, si possono scrivere i predicati:

`left` *a b c* ⇒ *boolean*

`left-on` *a b c* ⇒ *boolean*

`is-collinear` *a b c* ⇒ *boolean*

che ci dicono se i tre punti rappresentano una svolta a sinistra oppure no.

In Prolog avrete i predicati:

`left(A, B, C)`

`left_on(A, B, C)`

`collinear(A, B, C)`

Angolo tra due punti

Infine avrete bisogno di una funzione `angle2d` che ritorni l'angolo (in radianti) tra due punti. Avete bisogno di una funzione standard delle librerie Common Lisp e Prolog. NELLA FIGURA SOPRA PERO' MOSTRIAMO GLI ANGOLI MISURATI IN GRADI…. FORSE DOVREMMO SPIEGARE MEGLIO

`angle2d` *a b* ⇒ *real* ∈ $[-\pi, \pi]$.

In Prolog il predicato diventa:

`angle2d(A, B, R).`

"Sorting" e altri suggerimenti

L'algoritmo *Line Patterns* richiede l'utilizzo della funzione di ordinamento normalmente chiamata `sort`. La questione è che cosa questa funzione o predicato devono ordinare. Il suggerimento è di ordinare insiemi (liste o vettori) di coppie <*angolo*, *punto*> che potete rappresentare come volete (ma alcune rappresentazioni sono più comode di altre); chiamiamo queste coppie *slope*. E.g., in Prolog potete usare termini come **`slope(Angle, Point)`** per codificare queste coppie.

Sia in Common Lisp sia in Prolog l'utilizzo della funzione o predicato di libreria `sort` richiede molta attenzione. In particolare, l'operazione di confronto di due *slope*s va implementata con cura e in Prolog, il predicato di ordinamento va scelto in maniera appropriata tra `sort/2`, `sort/4`, `msort/2`, `keysort/2` etc. etc., al fine di ottenere l'effetto desiderato.

NB. NON dovete implementare una funzionalità di ordinamento.

Sia in Common Lisp che in Prolog avete bisogno della funzione `atan` (o simili).

NB. Non è detto che tutte le funzioni richieste (utili però per il debugging) siano necessarie nelle funzioni principali.

Funzione e predicato principali

Il vostro programma (Common Lisp) dovrà fornire una funzione principale chiamata:

`line-patterns` *points* ⇒ *lines*

Il risultato *lines* è un insieme di linee. Attenzione! Questo insieme (che può essere rappresentato come una lista) può contenere anche sotto segmenti. A voi la decisione se riportare solo segmenti massimali in *lines*.

In Prolog il predicato deve essere

`line_patterns(`*Points*`,` *Lines*`)`

Input (ed Output)

Il vostro progetto deve prevedere anche funzioni (predicati) per l'I/O di punti e segmenti.

Lettura punti

`readpoints` *filename* ⇒ *points*

filename è un nome di file e *points* è una lista di punti.

Il contenuto di filename è costituito da un numero **pari** di numeri, due per riga, separati da un carattere di *tabulazione* (il carattere "Tab": in Common Lisp `#\Tab`, in Prolog `0'\t`, in C e derivati `'\t'`) o una spaziatura (il carattere "Space": in Common Lisp `#\Space`, in Prolog `0' `, in C e derivati `' '`). Ad esempio questo è un file di test corretto:

```
0.1   6
3     7
4     6
4     5
3.4   4
2     4
5     0.42
42    123
-1    0
1024  -42
0     -2.0003
1     2
```

Attenzione ai punti duplicati[3].

In Prolog:

`readpoints(`*Filename*`,` *Points*`).`

In SWI Prolog potete utilizzare la libreria standard `csv` per leggere il file di punti.

Scrittura punti

Ovviamente, il vostro programma dovrà anche fornire una funzione (predicato) che scriva un file nel formato suddetto.

In Common Lisp:

`writepoints` *filename points* ⇒ *boolean*

In Prolog:

`writepoints(`*Filename*`,` *Points*`)`

Scrittura segmenti

Il vostro programma dovrà anche prevedere una funzione (predicato) per scrivere segmenti di linea su un file. In questo caso il formato dovrà essere "un segmento per linea di file", con ogni linea di file contenente un numero pari di numeri. Ad esempio:

[3] Nota bene. Il vostro programma dovrà essere in grado di leggere centinaia di punti.

```
…
0       0       1       1.5     2       3
0       6       1       3
…
```

La funzione Common Lisp dovrà essere:

`write-segments` *filename segments* ⇒ *boolean*

Il predicato Prolog dovrà essere:

`write_segments`(*Filename*, *Segments*)

Da consegnare

Dovrete consegnare un file `.zip` (i files `.tar`, `.7z`, `.tgz`, `.tar.gz`, `.rar` etc. **non sono accettabili**!!!) dal nome

```
Cognome_Nome_Matricola_lp_LP_201706.zip
```

Cognomi e nomi multipli dovranno essere scritti sempre con il carattere "underscore" ('_'). Ad esempio, "Gian Giacomo Pier Carl Luca De Mascetti Vien Dal Mare" che ha matricola 424242 diventerà:

```
De_Mascetti_Vien_Dal_Mare_Gian_Giacomo_Pier_Carl_Luca_424242_lp_LP_201706
```

Questo file deve contenere una sola directory con lo stesso nome. Al suo interno ci devono essere due sottodirectory chiamate '`Lisp`' e '`Prolog`'. Al loro interno queste cartelle devono contenere i files caricabili e interpretabili, più tutte le istruzioni che riterrete necessarie. Il file Lisp si deve chiamare '`linepatterns.lisp`'. Il file Prolog si deve chiamare '`linepatterns.pl`'. La cartella deve contenere un file chiamato `README.txt`. In altre parole questa è la struttura della directory (folder, cartella) una volta spacchettata.

```
Cognome_Nome_Matricola_lp_LP_201706
        Lisp
                linepatterns.lisp
                README.txt
        Prolog
                linepatterns.pl
                README.txt
```

Potete anche aggiungere altri file, ma il loro caricamento dovrà essere effettuato automaticamente al momento del caricamento ("loading") dei files sopracitati.

Ogni studente deve consegnare, anche se lavora in gruppo. Le prime linee dei files devono contenere la composizione del gruppo di lavoro.

```
prompt$ unzip -l Antoniotti_Marco_424242_lp_LP_201706.zip
Archive:  Antoniotti_Marco_424242_lp_LP_201706.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
        0  12-02-14 09:59   Antoniotti_Marco_424242_lp_LP_201706/
        0  12-04-14 09:55   Antoniotti_Marco_424242_lp_LP_201706/Lisp/
     4623  12-04-14 09:51   Antoniotti_Marco_424242_lp_LP_201706/Lisp/linepatterns.lisp
    10598  12-04-14 09:53   Antoniotti_Marco_424242_lp_LP_201706/Lisp/README.txt
        0  12-04-14 09:55   Antoniotti_Marco_424242_lp_LP_201706/Prolog/
     4623  12-04-14 09:51   Antoniotti_Marco_424242_lp_LP_201706/Prolog/linepatterns.pl
    10598  12-04-14 09:53   Antoniotti_Marco_424242_lp_LP_201706/Prolog/README.txt
 --------                   -------
    30442                   7 files
```

Valutazione

Il vostro programma sarà controllato con set di files assolutamente arbitrari (di cui almeno uno totalmente casuale), quindi, dovrete essere pronti a ogni evenienza.

Abbiamo a disposizione una serie di esempi standard che saranno usati per una valutazione oggettiva dei programmi. In particolare, i files di test che useremo hanno un numero di punti dell'ordine di 10^2 e sono generati in modo casuale. Farete bene a testare i vostri programmi in modo similare.

Se i files sorgente non potranno essere letti/caricati negli ambienti Lisp e Prolog (NB: Lispworks e SWI-Prolog), il progetto non sarà ritenuto sufficiente.

Il mancato rispetto dei nomi indicati per funzioni e predicati, o anche delle strutture proposte e della semantica esemplificata nel testo del progetto, oltre a comportare ritardi e possibili fraintendimenti nella correzione, può comportare una riduzione nel voto ottenuto o la non correzione dell'elaborato.

Riferimenti

[CLR+01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 2nd Edition, MIT Press, 2001

[O98] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1998.

[S02] R. Sedgewick, Algorithms in C++ (1-5), Addison-Wesley Professional, 2002.