Racunska geometrija

December 18, 2024

1 Računska geometrija

Računska geometrije je področje algoritmov in podatkovnih struktur, ki se ukvarja z učinkovitim reševanjem geometrijskih problemov. Ti vključujejo delo s točkami, daljicami, večkotniki in drugimi geometrijskimi objekti ter relacijami med njimi, kot sta npr. razdalja ali vsebovanost. Računska geometrija je očiten del računalniške grafike, vida, robotike. Manj očitno pa je prisotna tudi v številnih drugih problemih, ki dopuščajo geometrijsko formulacijo.

Omejili se bomo na reševanje problemov v ravnini, saj se problemi v višjih dimenzijah običajno dodatno zakomplicirajo. Poleg tega je reševanje ravninskih problemov enostavno za vizualizacijo. Kljub temu pa moramo biti pri reševanju pozorni na številne *posebne primere*, kot so kolinearne točke, sovpadanje točk, vzporedne daljice, … Pomembna ovira je tudi *računska natančnost*. Če imamo opravka s celoštevilskimi objekti, želimo rešiti problem z uporabo celih števil, da ne vpeljemo računske napake, ki bi lahko povzročila povsem napačen rezultat.

[1]:
```
#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;

typedef pair<int,int> PII;
typedef vector<PII> VII;
```

[2]:
```
template<class A, class B>
ostream& operator<<(ostream& os, pair<A,B> &p) {
    os << "(" << p.first << ", " << p.second << ")";
    return os;
}
```

[3]:
```
template<typename T>
void print(const vector<T> &sez) {
    for (T x : sez) cout << x << " ";
    cout << endl;
}
```

1.1 Razdalje in presečišča

Razdalje in presečišča so najbolj osnovni koncepti, ki jih moramo obvladati. Ne vključujejo kakšnih novih algoritmičnih prijemov, vendar služijo kot ponovitev geometrije in linearne algebre. Za naštete probleme seveda obstaja več formul. Ogledali si bomo najbolj enostavne, ki jih lahko izpeljemo brez prepisovanja iz kakšnega učbenika. Glede na predstavitev premic imamo različne pristope. Predpostavili bomo, da imamo premico P predstavljeno s točko P_0 in smernim vektorjem V_P.

Razdalje:

- Točki S in T: Pitagorov izrek poznamo še iz osnovne šole.
- Točka S in premica P: Izračunamo projekcijo S' točke S na premico P in izračunamo razdaljo med S in projekcijo S'. **Projekcijo** točke na premico izračunamo s pomočjo **skalarnega produkta**: $\text{proj}_b\, a = \frac{a \cdot b}{||b||^2} b$. Če delamo projekcijo na enotski smerni vektor premice, je dolžina projekcije kar enaka skalarnemu produktu.
- Točka S in daljica AB: Izračunamo projekcijo točke na nosilko (premico) daljice. Če je projekcija izven daljice, bo najkrajša razdalja do krajišča A ali B, sicer pa do projekcije S'.
- Daljici AB in CD: Predpostavimo, da se daljici ne sekata, sicer je odgovor 0. Najkrajša razdalja med daljicama bo enaka razdalji med enim izmed krajišč in drugo daljico. Izberemo najmanjšo izmed štirih možnosti.

Presečišča:

- Točka S in premica P: Če je vektorski produkt vektorja P_0S in smernega vektorja premice P enak 0, leži točka na premici.
- Točka S in daljica AB: Preverimo, ali točka leži na nosilki daljice in znotraj očrtanega pravokotnika (*bounding box*).
- Premici P in R: Če sta premici vzporedni, imamo neskončno ali nobenega presečišča. Sicer rešimo sistem enačb $P_0 + aV_P = R_0 + bV_R$ za obe koordinati.
- Premica in daljica: Izračunamo presečišče premice in nosilke daljice ter preverimo, ali leži presečišče na daljici.
- Daljici AB in CD: Ugotoviti moramo, ali se daljici sploh sekata, nato pa uporabimo rešitev za izračun presečišča nosilk obeh daljic. Daljici se sekata natanko takrat, ko sta krajišči prve daljice A in B na nasprotnih straneh nosilke daljice CD in obratno. **Stran/smer** ugotovimo s pomočjo **vektorskega produkta**. Točka A je na levi strani vektorja CD (v pozitivni smeri oz. nasprotni smeri urinega kazalca), če je vektorski produkt $CD \times CA$ pozitiven (na drugi strani bi bil negativen). Posebej pozorni moramo biti na primere, ko se daljici dotikata, kjer je vektorski produkt lahko 0.

1.2 Površina večkotnika

Začnimo s trikotnikom ABC. Če imamo podane koordinate oglišč, si lahko izberemo oglišče A za izhodišče in izračunamo polovico absolutne vrednost vektorskega produkta $p = \frac{1}{2}|AB \times AC|$.

Konveksen večkotnik lahko enostavno razbijemo na trikotnike in uporabimo prejšnji rezultat.

Na težave naletimo pri večkotnikih, ki niso nujno konveksni. Uporabimo formulo s predznačenimi vsotami trapezov $p = |\sum_{i=1}^{n} \frac{1}{2}(y_i + y_{i+1})(x_i - x_{i+1})|$. Predpostavimo lahko, da se večkotnik v celoti nahaja nad x-osjo (formula deluje tudi brez te predpostavke). Postavimo se nekam na x-os in opazujmo ozek vertikalen stolpec. Vsakič, ko bomo pri obhodu večkotnika prečkali stolpec

v desno stran, bomo območje pod njim odšteli, pri prehodu v levo pa prišteli. Marsikaj se bo izničilo in ostala bodo samo območja, ki imajo nad seboj liho število prečkanj (ta se izmenjujejo v levo in desno), kar je ravno notranjost večkotnika. Če naredimo obhod v drugo smer, bo rezultat negativen, po absolutni vrednosti pa enak.

Omenimo še, da deluje enak argument, če si izberemo poljubno izhodišče (npr. (0,0)) in seštevamo predznačene površine trikotnikov, ki jih z izbranim izhodiščem formirajo stranice na robu večkotnika.

1.3 Vsebovanost točke

Začnimo z najenostavnejšim primerom točke T, ki se nahaja v trikotniku ABC ali pač ne. Točka se nahaja v trikotniku, če se pri obhodu trikotnika ves čas nahaja na isti strani, kar preverimo z vektorskim produktom. Ali je to pozitivna ali negativna stran, je odvisno od smeri obhoda. Sledeči vektorski produkti morajo imeti enak predznak: $AB \times AT$, $BC \times BT$ in $CA \times CT$. Pozorni moramo biti, kaj problem zahteva v primeru, da se točka nahaja točno na robu trikotnika.

Naslednji primer je vsebovanost točke v konveksnem večkotniku. Enostavno ga lahko razbijemo na trikotnike (ki imajo skupno izbrano oglišče) in prevedemo problem na vsebovanost točke v trikotniku. Deluje pa tudi prej omenjeni pristop z lokacijo točke na isti strani obhoda večkotnika.

Kako pa rešimo problem za poljuben večkotnik (*point in polygon*), ki ni nujno konveksen? V tem primeru uporabimo tehniko metanja žarka (*ray casting*). Če sledimo poltraku iz točke T v poljubno smer, se ob vsakem križanju z robom večkotnika spremeni lokacija znotraj/zunaj. Če je število križanj liho, je točka znotraj večkotnika, sicer je izven. Pomembna podrobnost je, kaj se zgodi, če žarek seka večkotnik v enem od oglišč. Sprememba je namreč odvisna od sosednjih oglišč. Če več sosednjih oglišč leži na žarku, nas zanima prvo oglišče, ki ne. Če sta obe na isti strani žarka, ni spremembe, sicer pa je. Prikladna izbira smeri je npr. $z = (-1, 0)$. Lahko pa se tej komplikaciji izognemo s tako izbiro smeri (naključno), da do tega ne pride.

V spodnji implementaciji bomo predpostavili, da se točka ne nahaja na robu večkotnika. Če to ni res, bi lahko to posebej preverili. Pretvarjali se bomo, da ima točka za ϵ večjo y koordinato. To ne spremeni rešitve, vendar poenostavi razmislek, ker so vsa oglišča nad ali pod njo, ne pa na isti višini (oglišča z enako višino bodo obravnavana kot nižja). Problem bi z malo več truda lahko rešili tudi v celih številih, vendar zaradi preglednosti ne bomo dodatno kompliciral.

[4]:
```
#define OPERATOR_SUBTRACT operator- // workaround for a bug of cling
```

[5]:
```
PII OPERATOR_SUBTRACT(PII a, PII b) {
    return {a.first-b.first, a.second-b.second};
}
```

[6]:
```
int point_in_polygon(vector<PII> poly, PII t) {
    int n=poly.size(), cnt=0;
    auto [x,y] = t;
    for (int i=0;i<n;i++) {
        int j=(i+1)%n;
        if ((poly[i].second<=y) != (poly[j].second<=y)) {  // stranica seka␣
 ↪vodoravno premico
            PII s = poly[j]-poly[i];  // vektor stranice: i -> i+1
```

```
            PII v = t-poly[i];  // vektor do tocke: i -> t
            double k = (double)v.second/s.second;
            double xp = poly[i].first + k*s.first;  // presecisce z vodoravno␣
 ↪premico
            if (xp < x) cnt++;
        }
    }
    return cnt%2;
}
```

[7]:
```
vector<PII> poly = {{0,0}, {1,1}, {3,1}, {4,2}, {5,1}, {6,2}, {7,0}, {8,1},␣
 ↪{9,0}, {10,1}, {10,3}, {0,3}};
cout << point_in_polygon(poly, {9,1}) << endl;
cout << point_in_polygon(poly, {9,2}) << endl;
cout << point_in_polygon(poly, {6,1}) << endl;
cout << point_in_polygon(poly, {5,0}) << endl;
```

```
1
1
0
0
```

1.4 Konveksna ovojnica

Konveksna ovojnica/ogrinjača/lupina (*convex hull*) množice točk v ravnini je najmanjša konveksna množica, ki vsebuje vse podane točke. Običajno nas zanima rob konveksne ovojnice, ki je najkrajša sklenjena črta, ki vsebuje vse točke. Včasih tudi lomljeni črti, ki predstavlja rob konveksne ovojnice, rečemo kar konveksna ovojnica. Predstavljamo si jo lahko kot elastiko, ki se skrči okoli množice točk.

Iščemo ekstremne (robne) točke na robu ovojnice, ki jo definirajo. V primeru več kolinearnih točk na robu, je stvar definicije problema, ali želimo poročati samo oglišča ali tudi točke vzdolž stranic konveksne ovojnice. V nadaljevanju se bomo omejili na primere, kjer ni treh kolinearnih točk.

Ogledali si bomo par najbolj klasičnih algoritmov, obstaja pa jih še veliko več. Problem seveda postane težji, če ga rešujemo v treh ali še več dimenzijah.

[8]:
```
vector<PII> points = {{4,0}, {2,3}, {5,2}, {6,1}, {8,4}, {6,6}, {5,4}, {4,5},␣
 ↪{2,6}, {1,1}, {1,5}, {3,2}};
```

1.4.1 Identifikacija stranic

Rob konveksne ovojnice je sestavljen iz daljic med pari točk. Če lahko za posamezen par točk oz. daljico med njima ugotovimo, ali je del konveksne ovojnice, lahko zgradimo konveksno ovojnico. Daljica je del roba konveksne ovojnice, če se vse ostale točke nahajajo na isti strani (recimo na levi/pozitivni). Časovna zahtevnost takega postopka je $O(n^3)$, kjer je n število točk.

[9]:
```
int cross(PII u, PII v) {
    return u.first*v.second - u.second*v.first;
```

```
}
```

[10]:
```
int n=points.size();
for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) if (i!=j) {  // vektor daljice i-j
        PII d=points[j]-points[i];
        int ok=1;
        for (int k=0;k<n;k++) if (k!=i && k!=j) {
            PII v=points[k]-points[i];
            if (cross(d,v)<0) ok=0;
        }
        if (ok) cout << char('A'+i) << " " << char('A'+j) << endl;
    }
}
```

```
A D
D E
E F
F I
I K
J A
K J
```

Stranice seveda niso izpisane v vrstnem redu, kot si sledijo na konveksni ovojnici, vendar bi jih lahko uredili, če bi bilo treba.

1.4.2 Zavijanje darila

Pri iskanju konveksne ovojnice smo lahko bolj učinkoviti. Naraven pristop zavijanja darila (*gift wrapping, Jarvis march*) začne z izbiro točke, ki je gotovo del konveksne ovojnice. V ta namen lahko izberemo npr. najbolj levo točko A (najnižjo med najbolj levimi) in raztegnemo ovojni papir navzgor. Papir ovijamo v smeri urinega kazalca dokler se ne dotakne naslednje točke. Postopek ovijanja ponavljamo, dokler ne pridemo do začetne točke.

Naslednjo točko, ki se jo dotakne papir pri ovijanju, lahko poiščemo na več načinov. Ker so med gradnjo konveksne ovojnice vedno vse točke na isti strani zadnje točke A (del neke polravnine skozi A), lahko med njimi poiščemo najbolj levo z uporabo vektorskega produkta $AC \times AB$ za primerjavo, ali je točka C bolj levo (oz. v nasprotni smeri urinega kazalca) od točke B.

[11]:
```
VII gift_wrapping(VII points) {
    int n=points.size();
    PII start=*min_element(points.begin(), points.end());
    vector<PII> hull;
    PII a=start;
    while (a!=start || hull.empty()) {
        hull.push_back(a);
        PII b = (a!=points[0])?points[0]:points[1];  // katerakoli tocka, ki ni␣
↪a
        for (PII c : points) if (c!=a) {
```

```
            PII ac=c-a, ab=b-a;  // vektorja AC, AB
            if (cross(ab,ac)>0) b=c;
        }
        a = b;
    }
    return hull;
}
```

[12]:
```
auto hull = gift_wrapping(points);
print(hull);
```

```
(1, 1) (1, 5) (2, 6) (6, 6) (8, 4) (6, 1) (4, 0)
```

Časovno zahtevnost algoritma lahko analiziramo v odvisnosti od velikosti rezultata (*output-sensitive*) - število točk h na konveksni ovojnici. V tem primeru je časovna zahtevnost $O(hn)$. Če pa velikosti rezultata ne upoštevamo, so lahko v najslabšem primeru vse točke na robu ovojnice, zato je časovna zahtevnost $O(n^2)$.

1.4.3 Grahamov pregled

Konveksno ovojnico točk v ravnini lahko poiščemo bolj učinkovito kot v kvadratnem času in sicer z uporabo Grahamovega pregleda (*Graham scan*). Ponovno si izberimo neko ekstremno točko T, ki je zagotovo del konveksne ovojnice (npr. najnižjo med najbolj levimi točkami). Uredimo preostale točke glede na kote vektorjev iz točke T (od tistih, ki kažejo navzdol, proti vodoravnim in tistim, ki kažejo navzgor). Naj bo ta urejen seznam točk $P_1, P_2, \ldots$ Če jih povežemo, dobimo ovojnico, ki vsebuje vse točke (kar v ogliščih), vendar ni konveksna. Vemo tudi, da bo konveksna ovojnica neka podmnožica tega seznama točk. Vrstni red točk je že pravilen, samo izbrati moramo prave.

Algoritem gradi konveksno ovojnico postopno z dodajanjem novih točk v izbranem vrstnem redu po kotih. Po vsaki dodani točki popravi konveksnost zgrajene ovojnice, če je nova točka podrla konveksnost z obratom v napačno smer. To naredi z odstranjevanjem točk s konca zgrajene ovojnice, dokler zaključek ovojnice z novo točko ni konveksen.

Grahamov pregled pravzaprav gradi vedno večjo konveksno ovojnico z dodajanjem posameznih točk po kotih. V i-tem koraku doda točko P_i in iz konveksne ovojnice točk $T, P_1, P_2, \ldots, P_{i-1}$ izračuna konveksno ovojnico točk $T, P_1, P_2, \ldots, P_i$.

Časovna zahtevnost algoritma je zaradi urejanja $O(n \log n)$. Preostanek algoritma vključuje dodajanje in odstranjevanje točk z ovojnice, vendar je vsaka točka lahko dodana in odstranjena kvečjemu enkrat. Zato je ta del algoritma linearen v odvisnosti od števila točk.

[13]:
```
VII graham_scan(VII points) {
        int n=points.size();
    PII t=*min_element(points.begin(), points.end());

    vector<pair<double,PII>> angles;
    for (PII p : points) if (p!=t) {
        PII v = p-t;
        angles.push_back({atan2(v.second, v.first), p});
    }
```

```
    sort(angles.begin(), angles.end());

    vector<PII> hull = {t};  // stack
    for (auto [_,c] : angles) {
        while (hull.size()>=2) {  // restore convexity
            PII a=hull[hull.size()-2], b=hull[hull.size()-1];
            PII ab=b-a, ac=c-a;
            if (cross(ab,ac)>0) break;
            hull.pop_back();
        }
        hull.push_back(c);
    }
    return hull;
}
```

[14]:
```
auto hull2 = graham_scan(points);
print(hull2);
```

```
(1, 1) (4, 0) (6, 1) (8, 4) (6, 6) (2, 6) (1, 5)
```