Geometrie computațională III

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2015/2016

1. Triangularizarea unui poligon

2. Probleme de apropiere (Proximity problems)
   - Un mic catalog al problemelor apropiere
   - Cea mai îndepărtată pereche

Plan

1. Triangularizarea unui poligon

2. Probleme de apropiere (Proximity problems)
   - Un mic catalog al problemelor apropiere
   - Cea mai îndepărtată pereche

Definiția triangularizării (reamintire) 1/2

O diagonală a unui poligon L este un segment AB determinat de două vârfuri A și B cu proprietatea că orice punct Q din segmentul AB, $Q \notin \{A, B\}$, se află în interiorul lui L.

Proposition

Orice poligon cu $n \geq 4$ vârfuri are cel puțin o diagonală.

Definiția triangularizării (reamintire) 2/2

Theorem (Triangularizare)

Orice poligon cu $n \geq 3$ poate fi partiționat în triunghiuri ducând diagonale.

Demonstrația prin inducție după n.

Definition

O partiționare a unui poligon în triunghiuri prin ducerea de diagonale se numește triangularizare.

În general un poligon admite mai multe triangularizări. (exemplificare pe tablă)

Proprietăți

Proposition

Orice triangularizare a unui poligon cu n vârfuri utilizează $n-3$ diagonale și are $n-2$ triunghiuri.

Demonstrația prin inducție după n.

Definition

O ℓ-colorare a unui graf $G=(V,E)$ constă în colorarea vârfurilor cu cel ℓ-culori astfel încât nu există două vârfuri adiacente cu aceeași culoare.

Formal: există $c : V \to \{1,2,\ldots,\ell\}$ a.î. $(\forall \{i,j\} \in E) c[i] \neq c[j]$.

Theorem

Orice triangularizare a unui poligon admite o 3-colorare.

Demonstrația prin inducție după n.

Aplicație: Problema galeriei de artă

Podeaua unei galerii de arta este sub forma unui poligon. Se pune problema determinării numărului de paznici care să supravegheze complet întreaga galerie. Unghiul de vedere al unui paznic este de 360 de grade dar el nu poate vedea prin ziduri.

Theorem

Uneori sunt necesari $\lfloor \frac{n}{3} \rfloor$ paznici și totdeauna sunt suficienți $\lfloor \frac{n}{3} \rfloor$ paznici pentru o galerie cu n vârfuri.

Demonstrarea teoremei

- se triangularizează galeria
- se colorează triangularizarea cu 3 culori
- o soluție a problemei: plasarea paznicilor în vârfuri cu aceeași culoare
- există o culoare c care colorează cel mult $\left\lfloor \frac{n}{3} \right\rfloor$ vârfuri (se aplică "pigeon-hole principle")
- caz când sunt necesari $\left\lfloor \frac{n}{3} \right\rfloor$ paznici: galeria fierăstrău cu $\left\lfloor \frac{n}{3} \right\rfloor$ dinți îndepărtați

Algoritm pentru triangularizare bazat eliminarea urechii: ideea

- se bazează pe demonstrația teoremei de existență a triangularizării
- $\exists$ C_n^2 candidați la diagonale;
  testarea dacă două vârfuri formează diagonală: $O(n)$
  calculul celor $n-3$ diagonale ale triangularizării: $O(n^4)$
- reducere la $O(n^3)$: se observă că sunt numai $O(n)$ urechi, deci trebuie testați $O(n)$ candidați
- reducere la $O(n^2)$: se determină statusul de "ureche" pentru fiecare vârf și se actualizează la fiecare eliminare a unei urechi

Candidat la diagonală: definiția și operații auxiliare

- unește două vârfuri nealăturate ale poligonului
- nu intersectează alte laturi ale poligonului
- utilizăm iteratori pentru a referi vârfurile poligonului
- operație auxiliară: `isIntSeg(A, B, C, D)` - testează dacă segmentele AB și CD se intersectează

Candidat la diagonală: algoritm

```
/*
  tests if AB is a internal or external diagonal of the poligon L,
  where A = *a, B = *b, and a, b are iterators associated to L
*/
isDiagKind(a, b, L) {
  A = *a;  B = *b;
  for (p = L.first(); p != L.end(); ++ p) {
    q = p +% 1;
    if (( *p  == A && *q == B) ||
        ( *p  == B && *q == A))
      return false;
    if ( *p != A && *p != B &&
         *q != A && *q != B &&
         isIntSeg(A, B, *p, *q))
      return false;
  }
  return true;
}
```

Candidat la diagonală: test

```
test(out t) {
  L = < {x -> 1.0 y -> 1.0},
        {x -> 2.0 y -> 2.0},
        {x -> 3.0 y -> 1.0},
        {x -> 4.0 y -> 2.0},
        {x -> 5.0 y -> 1.0},
        {x -> 5.0 y -> 4.0},
        {x -> 3.0 y -> 5.0},
        {x -> 1.0 y -> 4.0} >;
  p0 = L.first();
  t[0] = isDiagKind(p0, p0+2, L);
  t[1] = isDiagKind(p0, p0+3, L);
}
t = [];
test(t);

$ krun ../tests/comp-geom3/isdiagkind.alk -cINIT=".Map"
<k> .K  </k>
<state>
    t |-> [ true, false ]
</state>
```

Candidat la diagonală: intern sau extern

Un candidat la diagonală poate fi

- inclus în interiorul poligonului sau
- inclus în exteriorul poligonului (detalii pe tablă)

O diagonală este un candidat inclus în interiorul poligonului.

Un candidat AB se află în interior dacă B este interior unghiului din A și dacă A este interior unghiului din B.

Testul de diagonală: algoritm

```
/*
  tests if AB is a diagonal in L, where A = *a, B = *b
*/
isDiag(a, b, L) {
  return isDiagKind(a, b, L) &&
         inCone(a, *b) &&
         inCone(b, *a);
}
```

Test pentru interior unghiului: definiția

Problema: test dacă A este interior vârfului P referit de p în poligonul L.

Se disting două cazuri:

- vârful P este convex: vecinii lui P se găsesc de o parte și de alta a lui PA
- vârful P este concav (reflex): se reduce la primul (A nu se află în unghiul convex din exterior)

Detalii pe tablă.

Test pentru interior unghiului: algoritm

```
/*
  tests if A is interior to the cone defined by
  the vertex *p of the poligon L
  (p is a iterator associated to L)
*/
inCone(p, A) {
  P = *p;
  Pprev = *(p -% 1);
  Pnext = *(p +% 1);
  if (ccw(P, Pnext, Pprev) == 1) // P is a convex vertex
    return ccw(P, A, Pprev) + ccw(A, P, Pnext) == 2;
  else // P is a concav (reflex) vertex
    return ccw(P, A, Pnext) + ccw(A, P, Pprev) != 0;
}
```

Test pentru interior unghiului: testare algoritm 1/2

```
test(out t) {
  A[0] = {x -> 1.0 y -> 1.0};
  A[1] = {x -> 2.0 y -> 2.0};
  A[2] = {x -> 3.0 y -> 1.0};
  A[3] = {x -> 4.0 y -> 2.0};
  A[4] = {x -> 5.0 y -> 1.0};
  A[5] = {x -> 5.0 y -> 4.0};
  A[6] = {x -> 3.0 y -> 5.0};
  A[7] = {x -> 1.0 y -> 4.0};
  L = emptyList;
  for (i = 0; i < A.size(); ++i)
    L.pushBack(A[i]);

  p = L.first();
  p = p + 1;  // *p is A[1]
  B1 = {x -> 2.0 y -> 0.0};
  t[0] = inCone(p, A[6]);
  t[1] = inCone(p, A[7]);
  t[2] = inCone(p, B1);
```

Test pentru interior unghiului: testare algoritm 2/2

```
  p = p + 1; // *p is A[2]
  t[3] = inCone(p, A[4]);
  t[4] = inCone(p, A[5]);
  t[4] = inCone(p, A[6]);
}

t = [];
test(t);

krun ../tests/comp-geom3/incone.alk -cINIT=".Map"
<k>
    .K
</k>
<state>
    t |-> [ true, true, false, false, true ]
</state>
```

Algoritmul de triangularizare scris în termenii din domeniul problemei

- se determină statusul de "ureche" pentru fiecare vârf
- cât timp poligonul procesat are mai mult de 3 vârfuri
  - caută un vârf ureche
  - adaugă diagonala determinată de vârfurile vecine
  - elimină vârful ureche
  - actualizează statusul de "ureche"

Statusul de "ureche"

Un vârf este ureche dacă vecinii lui formează o diagonală.

```
/*
  set the ear status for each vertex in L
*/
setEarStatus(out L) {
  for (p = L.first(); p != L.end(); ++ p)
    *p.ear = isDiag(p -% 1, p +% 1, L);
}
```

Fiecare vârf are un câmp suplimentar `ear`, care va memora valoarea booleană ce caracterizează statusul de "ureche".

Caută un vârf ureche

```
for (p = L.first(); !(*p.ear); ++p) {}
```

Execuția se termină întotdeauna deoarece orice pligon cu $n \geq 3$ vârfuri are cel puțin o ureche.

Actualizarea statusului de "ureche"

- numai vecinii vârfului urechii își pot modifica statusul
  Detalii pe tablă

```
*(p -% 1).ear = isDiag(p -% 2, p +% 1, L);
*(p +% 1).ear = isDiag(p -% 1, p +% 2, L);
```

Triangularizare: algoritm

```
triangulate(L, out D) {
  D = emptySet;
  setEarStatus(L);
  while(L.size() > 3) {
    // search for an ear
    for (p = L.first(); !(*p.ear); ++p) {}
    // add the diagonal
    d.A.x = *(p -% 1).x;  d.A.y = *(p -% 1).y;
    d.B.x = *(p +% 1).x;  d.B.y = *(p +% 1).y;
    D.pushBack(d);
    // reset ear status for p-1 and p+1
    *(p -% 1).ear = isDiag(p -% 2, p +% 1, L);
    *(p +% 1).ear = isDiag(p -% 1, p +% 2, L);
    // remove *p
    p->delete();
  }
}
```

Triangularizare: testare

```
test(out D) {
  L = < {x -> 1.0 y -> 1.0},
        {x -> 2.0 y -> 2.0},
        {x -> 3.0 y -> 1.0},
        {x -> 4.0 y -> 2.0},
        {x -> 5.0 y -> 1.0},
        {x -> 5.0 y -> 4.0},
        {x -> 3.0 y -> 5.0},
        {x -> 1.0 y -> 4.0} >;
  triangulate(L, D);
}
test(D);

krun ../tests/comp-geom3/triangulate.alk -cINIT=".Map"
<k> .K </k>
<state>
  D |-> { { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 2e+00 y -> 2e+00 } },
          { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 3e+00 y -> 1e+00 } },
          { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 4e+00 y -> 2e+00 } },
          { A -> {x -> 4e+00 y -> 2e+00 } B -> {x -> 5e+00 y -> 4e+00 } },
          { A -> {x -> 1e+00 y -> 4e+00 } B -> {x -> 5e+00 y -> 4e+00 } }
        }
</state>
```

Triangularizare: analiza 1/2

Corectitudine:

- invariant:
  - L este un poligon nediagonalizat cu vârfurile o submulțime a celui inițial
  - D este o traingularizare a porțiunii rămase prin eliminarea lui L din poligonul inițial

Triangularizare: analiza 2/2

Timp:

- dimensiune instanță: $n = L.size()$
- cazul cel mai nefavorbil:
  - `setEarStatus(L)`: $O(n^2)$
  - bucla `while`:
    - căutarea urechii: $O(n)$
    - `isDiag(...)`: $O(n)$
    - restul operațiilor: $O(1)$

  și se execută de $n - 3$ ori
- $\Longrightarrow T(n) = O(n^2)$

Plan

1. Triangularizarea unui poligon

2. Probleme de apropiere (Proximity problems)
   - Un mic catalog al problemelor apropiere
   - Cea mai îndepărtată pereche

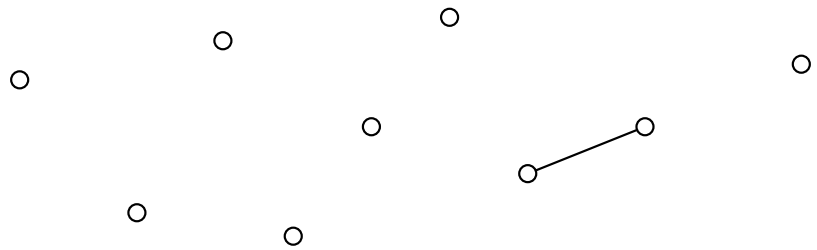
Plan

1 Triangularizarea unui poligon

2 Probleme de apropiere (Proximity problems)
- Un mic catalog al problemelor apropiere
- Cea mai îndepărtată pereche

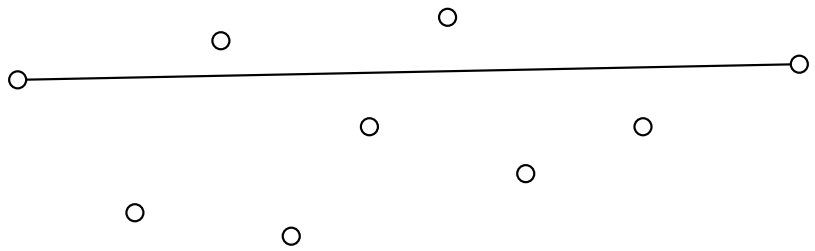
Un mic catalog al problemelor apropiere

CLOSEST PAIR
Intrare: O mulțime S cu n puncte în plan.
Ieșire: Două puncte din S aflate la cea mai mică distanță.



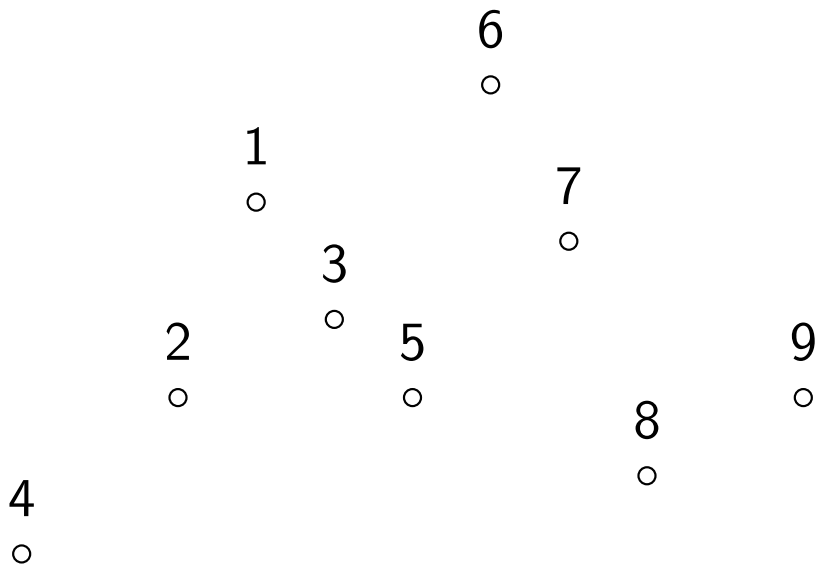
Un mic catalog al problemelor apropiere

DIAMETER SET
Intrare: O mulțime S cu n puncte în plan.
Ieșire: Două puncte din S aflate la cea mai mare distanță.



Un mic catalog al problemelor apropiere

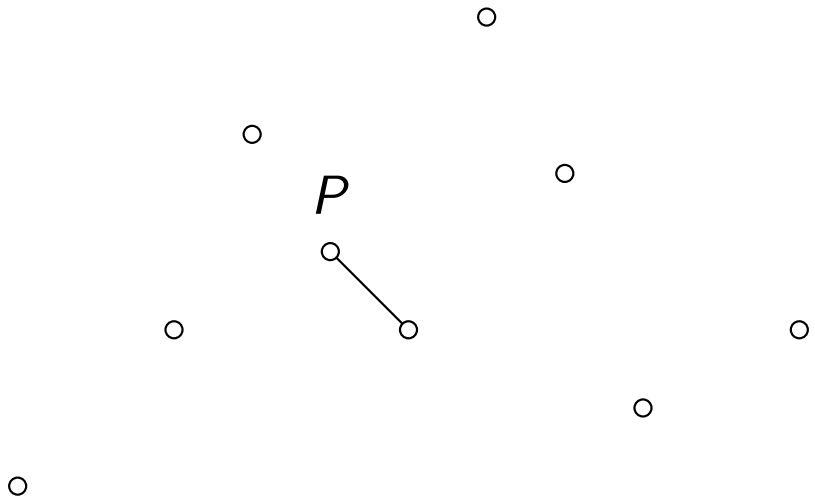
ALL NEAREST NEIGHBOR
Intrare: O mulțime S cu n puncte în plan.
Ieșire: Cel mai apropiat vecin din S pentru fiecare punct din S.



$(1, 3), (2, 3), (3, 5), (4, 2), \ldots$

Un mic catalog al problemelor apropiere

NEAREST NEIGHBOR
Intrare: O mulțime S cu n puncte, un punct P, toate în plan.
Ieșire: Cel mai apropiat punct de P din S.
(E posibilă o preprocesare a lui S.)

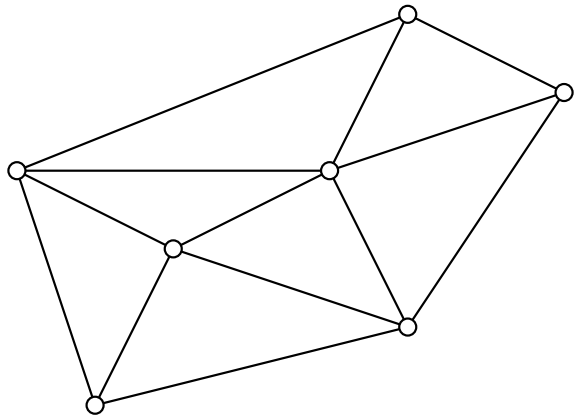


Un mic catalog al problemelor apropiere

TRIANGULATION

Intrare: O mulțime S cu n puncte în plan.

Ieșire: O mulțime de segmente care unesc puncte din S astfel încât orice regiune internă înfășurătorii convexe este triunghi.

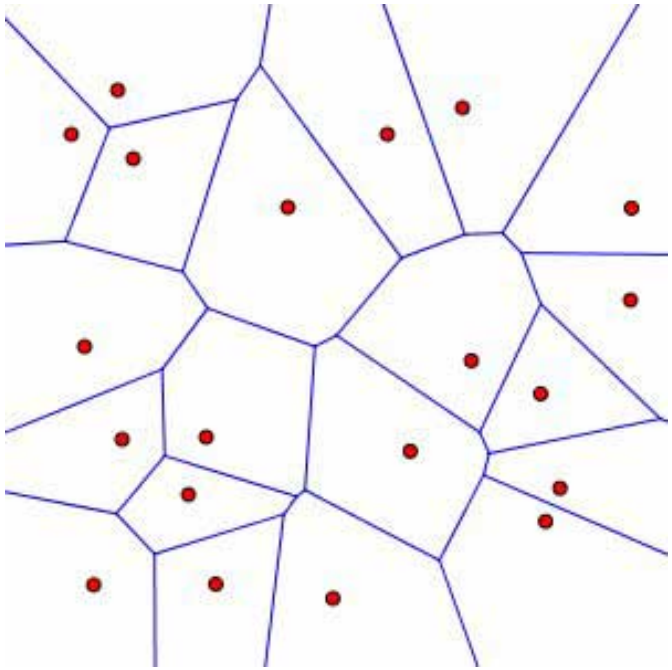


Un mic catalog al problemelor apropiere

VORONOI DIAGRAM

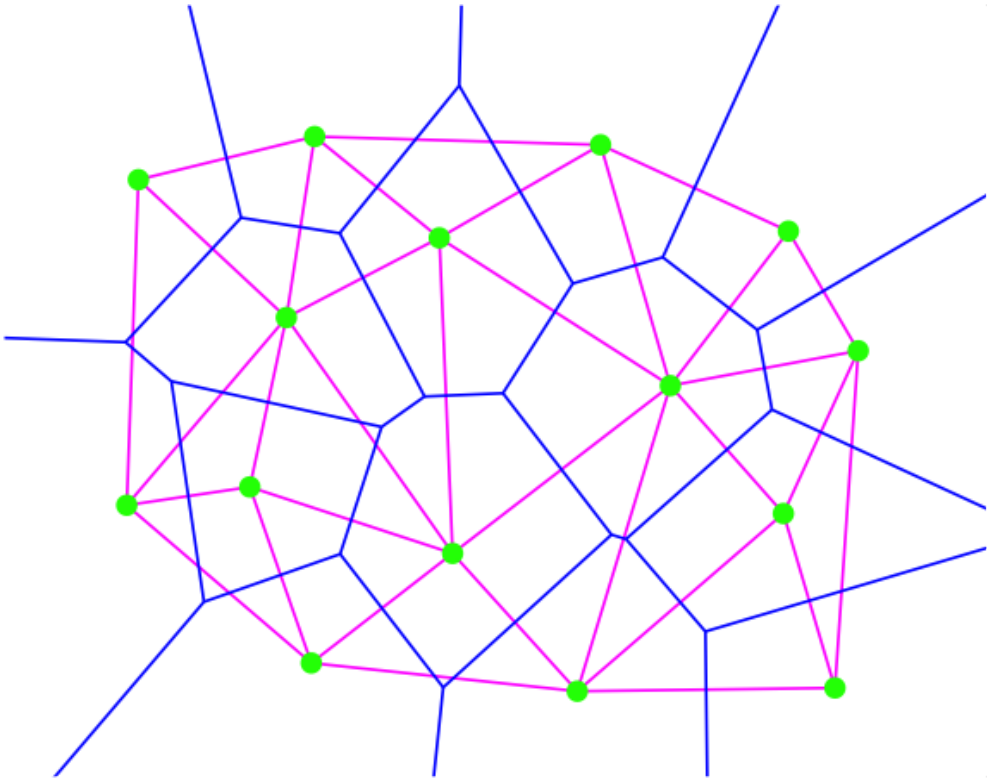
Intrare: O mulțime S cu n puncte în plan.

Ieșire: Pentru fiecare punct P din S, locul geometric la celor mai appropiate puncte, adică mulțimea punctelor mai apropiate de P decât de orice alt punct din S.



Sursa: http://www.ams.org/samplings/feature-column/fcarc-voronoi

Triangularizare Delaunay: exemplu clasic



Reduceri

Theorem

ALL NEAREST NEIGHBOR $\propto_n$ VORONOI DIAGRAM.

- se determină diagrama Voronoi
- pentru fiecare punct P, cel mai apropiat vecin al său se găsește pe o muchie Voronoi (nedenerată)
- cel mai apropiat vecin se poate determina parcurgând poligonul Voronoi care îl include punctul P
- numărul total de muchii parcurse este $2\times$ numărul de muchii din diagrama Voronoi (o muchie este frontiera a exact două regiuni
- numărul de muchii ale digramei Voronoi este $O(n)$ deoarece graful este planar (dacă oricare patru puncte nu sunt coliniare, atunci există cel mult $3n - 5$ muchii)

Corollary

ALL NEAREST NEIGHBOR poate fi rezolvată în timpul $O(n \log n)$.

Reduceri

Theorem

NEAREST NEIGHBOR $\propto_n$ ALL NEAREST NEIGHBOR.

1. se determină lista celor mai apropiați vecini
2. întoarce din listă cel mai apropiat vecin al punctului dat

Reduceri

Theorem

SORT $\propto_n$ TRIANGULATION.

- fie $x_0, \ldots, x_{n-1}$ n numere reale de sortat
- construiește $(x_0, x_0^2), \ldots, (x_{n-1}, x_{n-1}^2)$ o instanță pentru triangularizare
- triangularizarea Delaunay dă ordinea crescătoare a numerelor

Corollary

TRIANGULATION are complexitatea timp în cazul cel mai nefavorabil $\Omega(n \log n)$.

Reduceri

Theorem

TRIANGULATION $\propto_n$ VORONOI DIAGRAM.

- se determină diagrama Voronoi
- graful dual al diagramei este o triangularizare (Delaunay)
- construire grafului dual se poate face în $O(n)$ parcurgând muchiile diagramei

Corollary

TRIANGULATION poate fi rezolvată în timpul $O(n \log n)$.

Corollary

VORONOI DIAGRAM are complexitatea timp în cazul cel mai nefavorabil $\Omega(n \log n)$.

Plan

1. Triangularizarea unui poligon

2. Probleme de apropiere (Proximity problems)
   - Un mic catalog al problemelor apropiere
   - Cea mai îndepărtată pereche

Proprietăți 1/2

Reamintim problema:

DIAMETER SET
Input O mulțime S cu n puncte în plan.
Output Două puncte din S aflate la cea mai mare distanță.

Definition

Un segment care unește două cele mai îndepărtate puncte se numește diametru.

Proprietăți 2/2

Theorem

Fie PQ un diametru al lui S, L_P și L_Q două perpendiculare pe PQ care trec prin P respectiv Q. Atunci toate punctele din S se află între L_P și L_Q.

Demonstrație.

Pe tablă.

Corollary

Dacă PQ este un diametru al lui S, atunci $P, Q \in CH(S)$.

ANTIPODAL PAIRS

Definition

Două puncte $P, Q \in CH(S)$ se numesc antipodale (un fel de diametral opuse) dacă există două drepte paralele L_P ș dLQ, care trec prin P respectiv Q, astfel încât toate punctele din S se află între L_P și L_Q.

Corollary

Dacă PQ este un diametru al lui S, atunci P, Q sunt antipodale.

Reciproca nu este adevărată.

Deci cele mai îndepărtate puncte se află printre cele antipodale, de aceea considerăm următoarea problemă:

ANTIPODAL PAIRS
Intrare: Un poligon convex L cu n puncte în plan.
Ieșire: Mulțimea perechilor de vârfuri antipodale.

ANTIPODAL PAIRS: proprietăți

Lemma

Fie AB o latură a poligonului convex L și P primul cel mai îndepărtat punct față de AB întâlnit prin parcurgerea în sens CCW a lui L începând din B. Atunci niciunul dintre vârfurile între B și P (în sensul CCW) nu este antipodal lui B.

Demonstrație

Pe tablă.

ANTIPODAL PAIRS: proprietăți

Lemma

Fie AB o latură a poligonului convex L și Q ultimul cel mai îndepărtat punct față de AB întâlnit prin parcurgerea în sens CCW a lui L începând din B. Atunci niciunul dintre vârfurile între Q și A (în sensul CCW) nu este antipodal lui A.

Demonstrație

Similar lemei precedente.

Lemma

Cele mai îndepărtate puncte față de un vârf B sunt consecutive.

Demonstrație

Pe tablă.

ANTIPODAL PAIRS: descriere algoritm

Presupunem ca oricare trei vârfuri nu sunt coliniare, rezultă că pot fi cel mult două puncte cel mai îndepărtate față de o latură AB.
Cele două leme conduc la următorul algoritm pentru determinarea punctelor antipodale:

- Mulțimea $APPSet$, care va memora vârfurile antipodale, este inițializată cu mulțimea vidă.
- Inițial AB este $L[n-1]L[0]$ și $P = *i$ primul cel mai îndepărtat de AB
- Pentru fiecare latura curentă AB:
  - determină Q cel mai îndepărtat de latura următoare (sensul CCW) BC
  - toate vârfurile între P și Q sunt antipodale lui B

Compararea distanțelor de la două puncte la o dreaptă

Distanța este proporțională cu aria, așa că se poate utiliza `sign2xTriArea()`.

```
/*
  compare distance from P to AB with the distance from Q to AB
  returns:
   -1 if dist(P, AB) < dist(Q, AB)
   0  if dist(P, AB) = dist(Q, AB)
   +1 if dist(P, AB) > dist(Q, AB)
*/
cmpDist(A, B, P, Q) {
  dpab = sign2xTriArea(A, B, P);
  if (dpab < 0) dpab = 0 - dpab;
  dqab = sign2xTriArea(A, B, Q);
  if (dqab < 0) dqab = 0 - dqab;
  if (dpab < dqab) return -1;
  if (dpab > dqab) return +1;
  return 0;
}
```

Testul pentru cel mai îndepărtat

Niciunul dintre vecini nu este mai îndepărtat:

```
/*
  tests if I is the farthest from AB, where
  I = *i, A = *a, B = *b
*/
isFarthest(a, b, i) {
  return cmpDist(*a, *b, *i, *(i -% 1)) >= 0 &&
         cmpDist(*a, *b, *i, *(i +% 1)) >= 0;
}
```

De la descriere în limbajul problemei la descriere în limbaj algoritmic

- Mulțimea *APPSet*, care va memora vârfurile antipodale, este inițializată cu mulțimea vidă.
- Inițial AB este $L[n-1]L[0]$ și $P = *i$ primul cel mai îndepărtat de AB

```
APPSet = emptySet;
k = L.first();  // A = *(k -% 1), B = *k
i = k + 1;
while ( ! isFarthest(k -% 1, k, i))
  i = i + 1;
// P = *i is the first farthest from AB
```

De la descriere în limbajul problemei la descriere în limbaj algoritmic

- determină Q cel mai îndepărtat de latura următoare (sensul CCW) BC
- toate vârfurile între P și Q sunt antipodale lui B

```
while (! isFarthest(k, k +% 1, i)) {
  // (B, *i) is antipodal pair
  addPair(APPSet, k, i);
  i = i +% 1;
}
// P' = *i is the first farthest from the next edge BC,
// where C = *(i+1)
if (isFarthest(k, k +% 1, i +% 1)) {
  // (B, *i) is antipodal pair
  addPair(APPSet, k, i);
  i = i +% 1;
}
// Q = *i is the last farthest from AB and the last antipodal to B
addPair(APPSet, k, i);
```

De la descriere în limbajul problemei la descriere în limbaj algoritmic

- Pentru fiecare latura curentă AB:

```
do {
 // A = *(k -% 1), B = *k
 // P = *i is the first farthest from AB
...
 k = k + 1;
 // next edge BC became AB, A = *(k -% 1), B = *k
} while (k != L.end());
```

ANTIPODAL PAIRS: algoritm

```
antipodalPairs(L) {
  APPSet = emptySet;
  k = L.first();  // A = *(k -% 1), B = *k
  i = k + 1;
  while ( ! isFarthest(k -% 1, k, i))
    i = i + 1;
  p = i;
  do {
    while (! isFarthest(k, k +% 1, i)) {
      addPair(APPSet, k, i);
      i = i +% 1;
    }
    if (isFarthest(k, k +% 1, i +% 1)) {
      addPair(APPSet, k, i);
      i = i +% 1;
    }
    addPair(APPSet, k, i);
    k = k + 1;
  } while (k <= p);
  return APPSet;
}
```

ANTIPODAL PAIRS: analiza algoritmului

Corectitudine:
invarianții sunt descriși de comentariile din algoritm

Timp:

- dimensiune instanță: $n = L.size()$;
- cazul cel mai nefavorabil: algoritmul parcurge L în toate cazurile;
- există $O(n)$ perechi candidat BP' cu $B = *k$, $P' = *i$;
- $\Longrightarrow$ $T(n) = O(n)$

DIAMETER SET: descriere algoritm

1. determină $CH(S)$
2. calculează vârfurile antipodale ale lui $CH(S)$
3. determină perechea de vârfuri antipodale aflate la distanța maximă

Complexitate timp: $O(n \log n)$.

Exreciţiu. Să se descrie algoritmul în Alk.