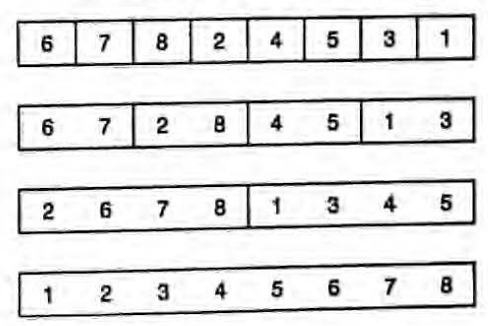
**Алгоритм «Разделяй и властвуй»**

При реализации подхода «разделяй и властвуй» задача делится на несколько подзадач, которые решаются рекурсивно и затем достигнутые результаты объединяются для получения решения исходной задачи.

**Сортировка путем слияния**

При сортировке путем слияния заданный массив элементов делится на левый и правый подмассивы, рекурсивно сортируется каждый из двух подмассивов и затем оба отсортированных подмассива сливаются в один отсортированный массив.

****

*Сортировка путем слияния*

void mSort(T a[], int l, int r, int (\*cmp)(T,T)){

if (1 < r) {

int m = (l + r) / 2;

mSort(a, l, m, cmp);

mSort(a, m+l, r, cmp);

merge(a, l, m, r, cmp);

}

}

Функция mSort выполняет сортировку путем слияния подмассивов a[l..m].

void mergeSort(T a[], int n, int (\*cmp)(T,T))

{

mSort(a, 0, n-i, cmp);

}

Функция mergeSort определяет ведущую функцию.

Функция merge осуществляет слияние отсортированных подмассивов x[l..m] и x[m+1..r] в отдельный массив с, который затем копируется обратно в x[l..r].

void merge(T x[], int l, int m, int r, int (\*cmp) (T,T))

{

T \*a = x+l;

T \*b = x+m+l;

T \*c = new T[r-l+1];

int aindx = 0, bindx = 0, cindx = 0;

int alim = m-l+1, blim = r-m;

while ((aindx < alim) && (bindx < blim)){

if ((\*cmp)(a[aindx], b[bindx]) < 0){

c[cindx++] = a[aindx++];

}

else{

c[cindx++] = b[bindx++];

}

}

while (aindx < alim){

c[cindx++] = a[aindx++];

}

while (bindx < blim){

c[cindx++] = b[bindx++];

}

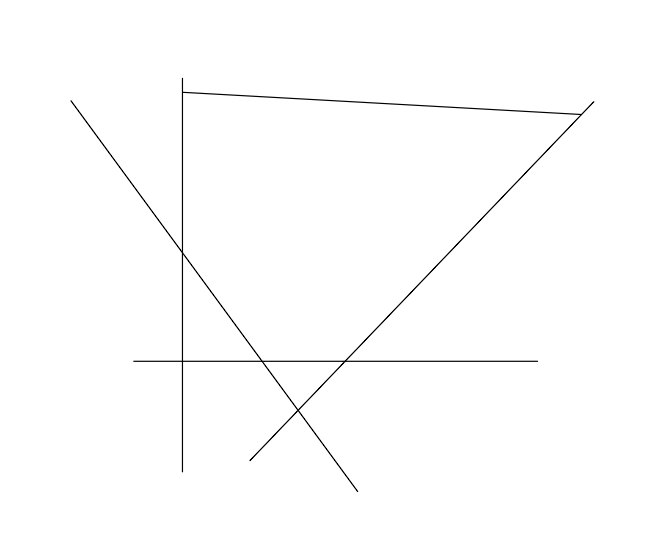
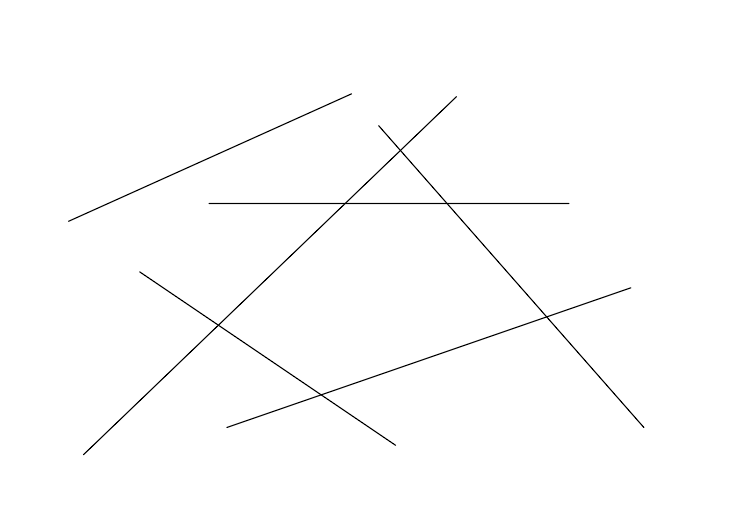
for (aindx=cindx=O; aindx <= r-1; a[aindx++] = c[cindx++]);

delete c;

}

**Вычисление области пересечения полуплоскостей**

Для вычисления области пересечения набора полуплоскостей, мы будем разбивать набор полуплоскостей на 2 непустых поднабора, затем рекурсивно вычислять области пересечения для них. После этого мы комбинируем полученные области пересечения.



*а) Выпуклый полигон. б) Неограниченный выпуклый политоп.*

void advance(Polygon &A, Polygon &R, int inside)

{

A.advance(Vertex::CLOCKWISE);

if (inside && (R.point() != A.point()))

R.insert(A.point());

}

bool aimsAt(Edge &a, Edge &b, int aclass, int crossType) {

Point va = a.dest - a.org;

Point vb = b.dest - b.org;

if (crossType != Edge::COLLINEAR) {

if ((va.x \* vb.y) >= (vb.x \* va.y))

return (aclass != Point::RIGHT);

else

return (aclass != Point::LEFT);

}

else {

return (aclass != Point::BEYOND);

}}

}

Функция pointInConvexPolygon обсчитывает точку s и полигон p и возвращает значение TRUE только в том случае, если точка лежит внутри полигона.

bool pointInConvexPolygon(Point &s, Polygon &p)

{

if (p.size() == 1)

return s == p.point();

if (p.size() == 2)

{

int c = s.classify(p.edge());

return ((c == Point::BETWEEN) || (c == Point::ORIGIN) | (c == Point::DESTINATION));

}

Vertex \*org = p.v();

for (int i = 0; i < p.size(); i++, p.advance(Vertex::CLOCKWISE))

if (s.classify(p.edge()) == Point::LEFT)

{

p.setV(org);

return false;

}

return true;

}

Функция crossingPoint показывает взаимное положение ребер e и f. Также возвращает точку пересечения ребер через ссылочный параметр p.

int crossingPoint(Edge &e, Edge &f, Point &p)

{

double s, t;

int classe = e.intersect(f, s);

if ((classe == Edge::COLLINEAR) || (classe == Edge::PARALLEL))

return classe;

double lene = (e.dest - e.org).length();

if ((s < -EPSILON2 \* lene) || (s > 1.0+ EPSILON2 \* lene))

return Edge::SCEW\_NO\_CROSS;

f.intersect(e, t);

double lenf = (f.org - f.dest).length();

if ((-EPSILON2 \* lenf <= t) && (t <= 1.0 + EPSILON2 \* lenf))

{

if (t <= EPSILON2 \* lenf)

p = f.org;

else if (t >= 1.0 - EPSILON2 \* lenf)

p = f.dest;

else if (s <= EPSILON2 \* lene)

p = e.org;

else if (s >= 1.0 - EPSILON2 \* lene)

p = e.dest;

else

p = f.point(t);

return Edge::SCEW\_CROSS;

}

else

return Edge::SCEW\_NO\_CROSS;

}

Функция clipPolygonToEdge применяется для выделения части ограничивающего прямоугольника, лежащего справа от ребра (для каждой полуплоскости)

bool clipPolygonToEdge(Polygon &s, Edge &e, Polygon \*&result)

{

Polygon \*p = new Polygon;

Point crossingPt;

for (int i = 0; i < s.size(); s.advance(Vertex::CLOCKWISE), ++i)

{

Point org = s.point();

Point dest = s.cw()->point();

int orgIsInside = (org.classify(e) != Point::LEFT);

int destIsInside = (dest.classify(e) != Point::LEFT);

if (orgIsInside != destIsInside)

{

double t;

e.intersect(s.edge(), t);

crossingPt = e.point(t);

}

else if (orgIsInside && !destIsInside)

{

if (org != crossingPt)

p->insert(crossingPt);

}

else if (orgIsInside && !destIsInside);

else

{

p->insert(crossingPt);

if (dest != crossingPt)

p->insert(dest);

}

}

result = p;

return (p->size() > 0);

}

Polygon\* convexPolygonIntersect(Polygon &P, Polygon &Q) {

Polygon \*R;

Point iPnt, startPnt;

int inflag = UNKNOWN;

int phase = 1;

int maxItns = 2 \* (P.size() + Q.size());

for (int i = 1; (i <= maxItns || phase == 2); ++i) {

Edge p = P.edge();

Edge q = Q.edge();

int pclass = p.dest.classify(q);

int qclass = q.dest.classify(p);

int crossType = crossingPoint(p, q, iPnt);

if (crossType == Edge::SCEW\_CROSS)

{

if (phase == 1)

{

phase = 2;

R = new Polygon;

R->insert(iPnt);

startPnt = iPnt;

}

else if (iPnt != R->point())

{

if (iPnt != startPnt)

R->insert(iPnt);

else

return R;

}

if (pclass == Point::RIGHT)

inflag = P\_IS\_INSIDE;

else if (qclass == Point::RIGHT)

inflag = Q\_IS\_INSIDE;

else

inflag = UNKNOWN;

}

Функция convexPolygonIntersect формирует область пересечения для двух выпуклых полигонов.

else if (crossType == Edge::COLLINEAR && pclass != Point::BEHIND && qclass != Point::BEHIND)

{

inflag = UNKNOWN;

}

bool pAIMSq = aimsAt(p, q, pclass, crossType);

bool qAIMSp = aimsAt(q, p, pclass, crossType);

if (pAIMSq && qAIMSp) {

if ((inflag == Q\_IS\_INSIDE) || ((inflag == UNKNOWN) && (pclass == Point::LEFT)))

advance(P, \*R, false);

else

advance(Q, \*R, false);

}

else if (pAIMSq) {

advance(P, \*R, inflag == P\_IS\_INSIDE);

}

else if (qAIMSp) {

advance(Q, \*R, inflag == Q\_IS\_INSIDE);

}

else {

if ((inflag == Q\_IS\_INSIDE) || (inflag == UNKNOWN && pclass == Point::LEFT))

advance(P, \*R, false);

else {

advance(Q, \*R, false);

}

}

}

Point pp = P.point();

Point qp = Q.point();

if (pointInConvexPolygon(pp, Q))

return new Polygon(P);

else if (pointInConvexPolygon(qp, P))

return new Polygon(Q);

return new Polygon;

}

Функции halfplaneIntersect задается массив H из n ребер и выпуклый ограничивающий прямоугольник box.

Polygon \*halfplaneIntersect(Edge H[], int n, Polygon &box)

{

Polygon \*c;

if (n == 1)

clipPolygonToEdge(box, H[0], c);

else

{

int m = n / 2;

Polygon \*a = halfplaneIntersect(H, m, box);

Polygon \*b = halfplaneIntersect(H + m, n - m, box);

c = convexPolygonIntersect(\*a, \*b);

delete a;

delete b;

}

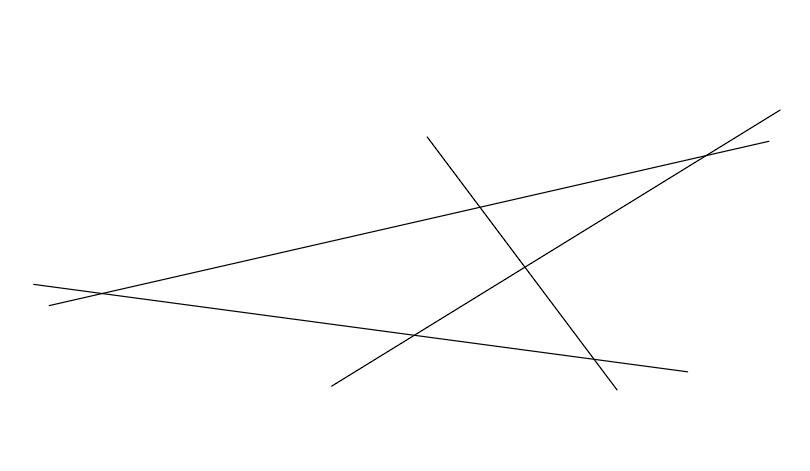
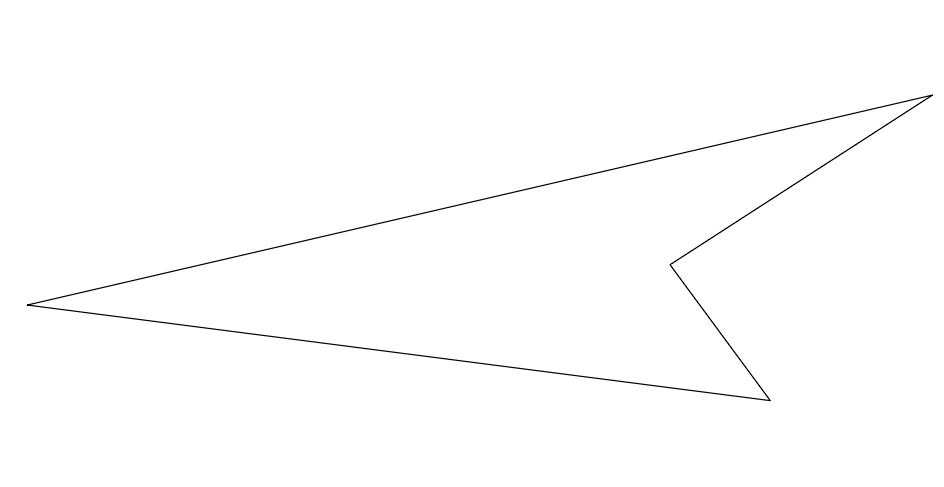
return c;

}

Она возвращает выпуклый полигон, равный пересечению прямоугольника box и n полуплоскостей в наборе H.

**Определение ядра полигона**

Ядро полигона – множество точек, из которых видны все точки внутри полигона. Если P является n-угольником и H – набор n полуплоскостей, определяемых ребрами P, то тогда ядро для P равно пересечению I(H) этих полуплоскостей.



*Ядро полигона P равно пересечению полуплоскостей, определяемых ребрами P.*

В функцию kernel передается полигон p. Она возвращает полигон, представляющий ядро полигона p, или пустой полигон, если p не является звездчатым полигоном.

Polygon \*kernel(Polygon &p)

{

Edge \*edges = new Edge[p.size()];

for (int i = 0; i < p.size(); i++, p.advance(Vertex::CLOCKWISE)){

edges[i] = p.edge();

Polygon box;

box.insert(Point(-DBL\_MAX, -DBL\_MAX));

box.insert(Point(-DBL\_MAX, DBL\_MAX));

box.insert(Point(DBL\_MAX, DBL\_MAX));

box.insert(Point(DBL\_MAX, -DBL\_MAX));

Polygon \*r = halfplaneIntersect(edges, p.size(), box);

delete edges;

return r;

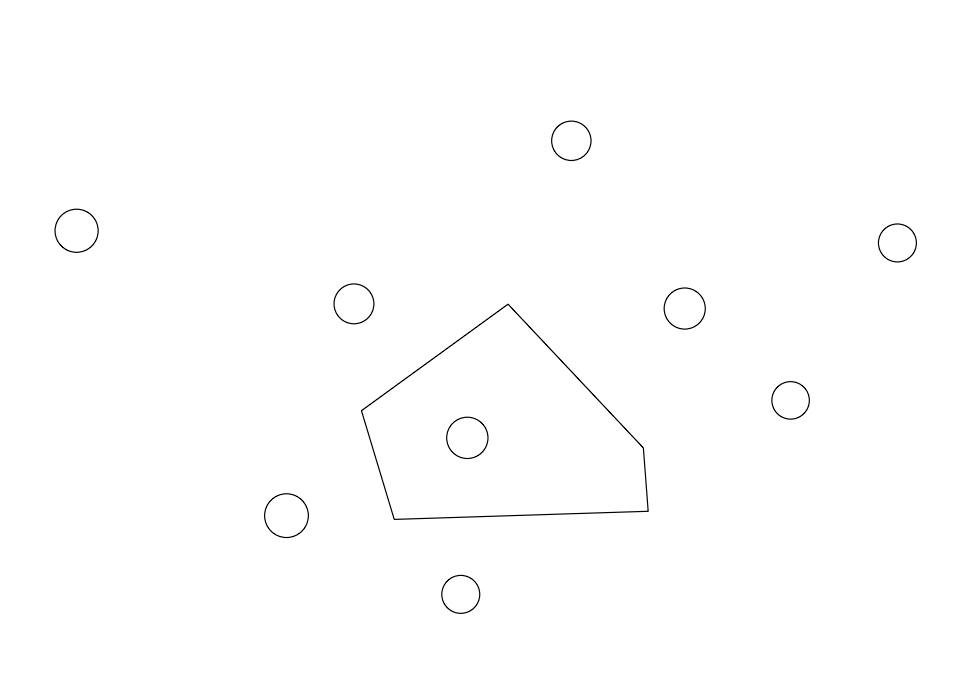
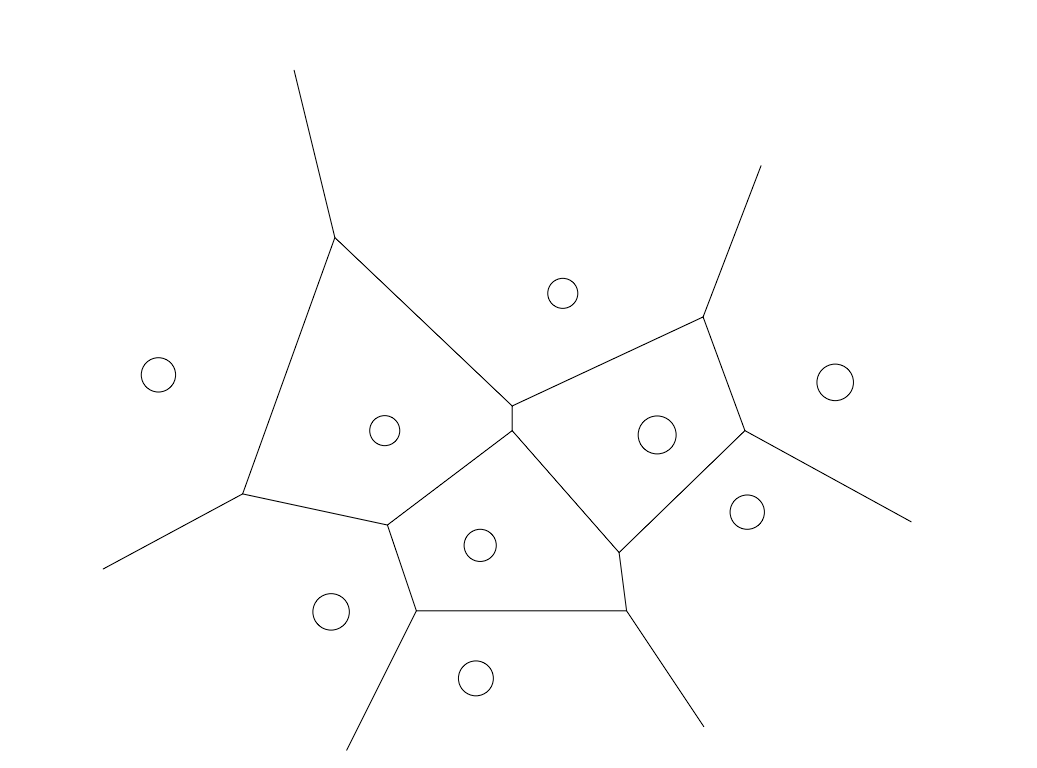
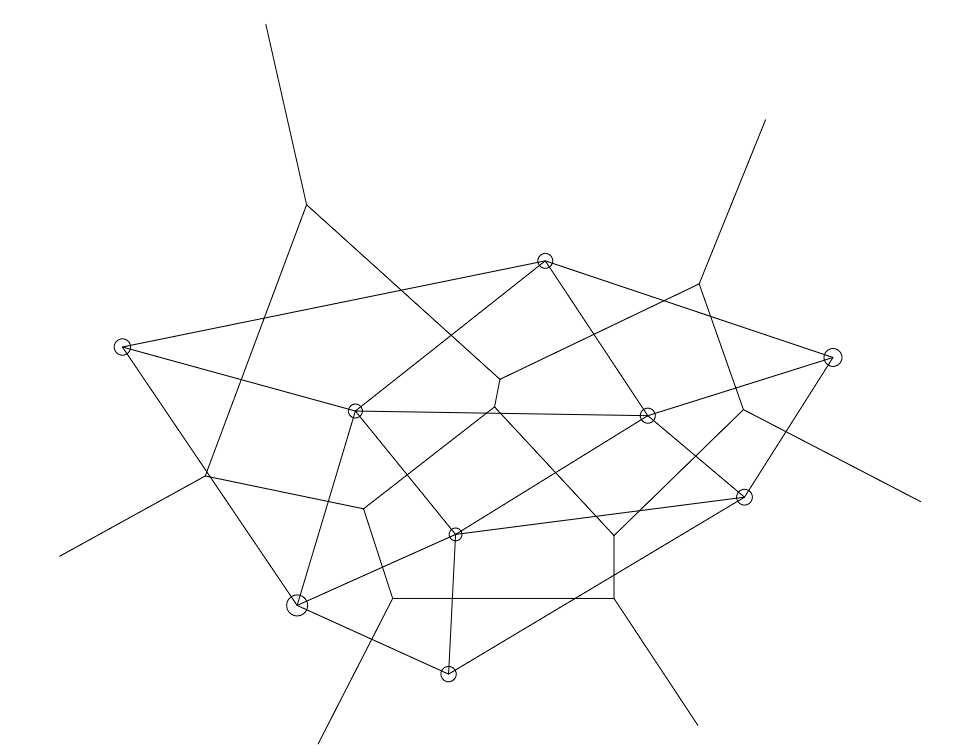
}

}

**Определение многоугольника Вороного**

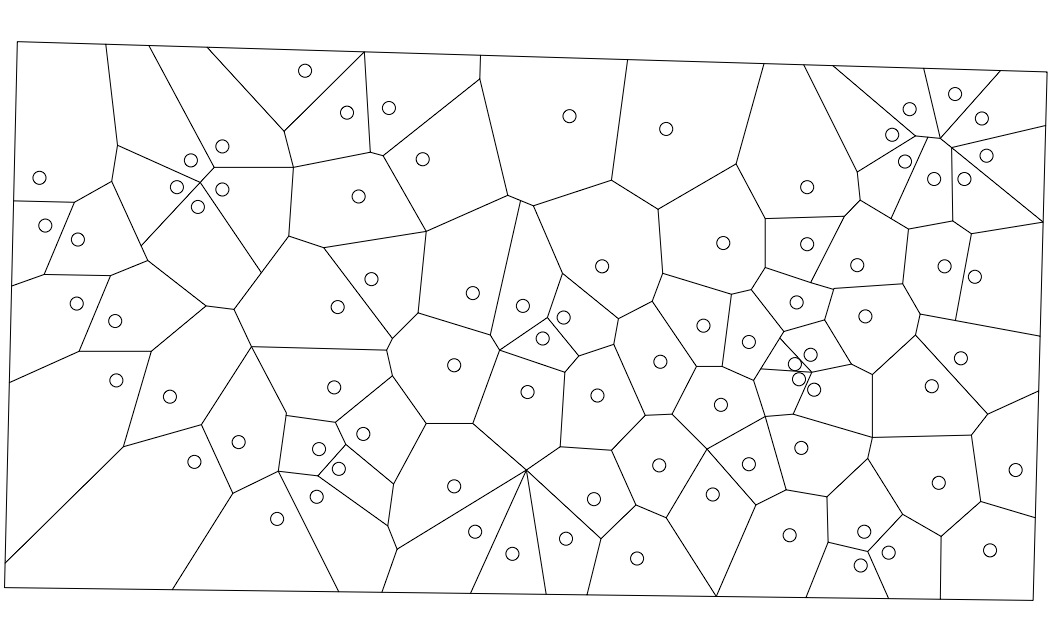
Для двух заданных различных точек p и q многоугольник Вороного от q равен полуплоскости, ограниченной перпендикуляром к середине отрезка pq. Многоугольник Вороного для множества S, образованного точками q, будет равен пересечению таких полуплоскостей.

Диаграмма Вороного состоит из коллекции многоугольников Вороного для каждой из точек набора S.

**** ****

*(а) (б) (в)*

*а) многоугольник Вороного. б) диаграмма Вороного. в) диаграмма Вороного и двойственная ей триангуляция Делоне.*

****

*Диаграмма Вороного для набора из 80 точек.*

Функции voronoiRegion задается точка p, массив s из n точек и ограничивающий прямоугольник box. Она возвращает многоугольник Вороного для точки p относительно набора точек s.

List<Polygon\*> \*voronoiDiagram(Point s[], int n, Polygon &box)

{

List<Polygon\*> \*regions = new List<Polygon\*>;

for (int i = 0; i < n; i++) {

Point p = s[i];

s[i] = s[n-1];

regions->append(voronoiRegion(p, s, n-1, box));

s[i] = p;

}

return regions;

}

Polygon \*voronoiRegion(Point &p, Point s[], int n, Polygon &box)

{

Edge \*edges = new Edge[n];

for (int i = 0; i < n; i++) {

edges[i] = Edge(p, s[i]);

edges[i].rot();

}

Polygon \*r = halfplaneIntersect(edges, n, box);

delete edges;

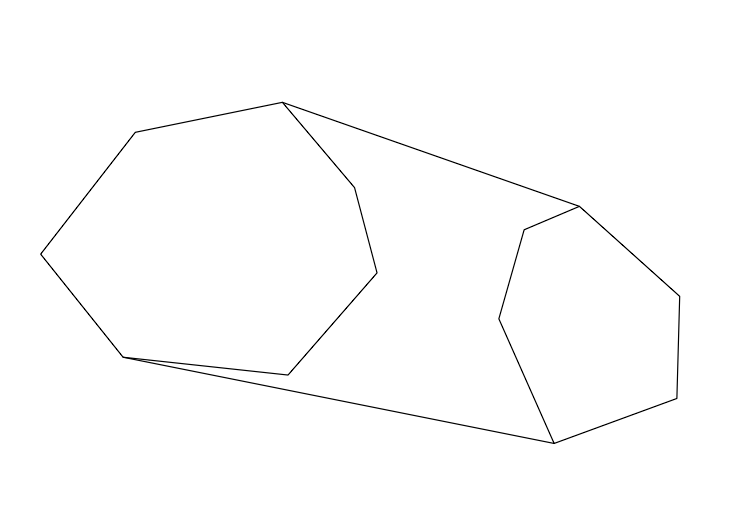
return r;

}

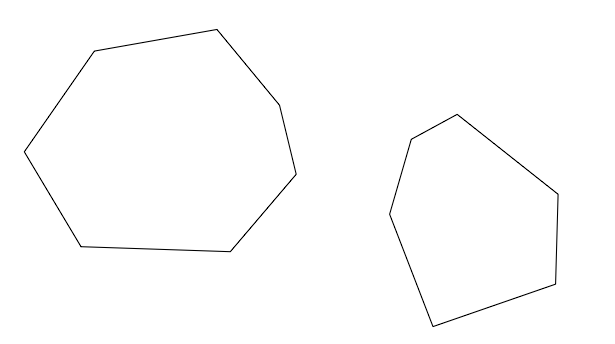
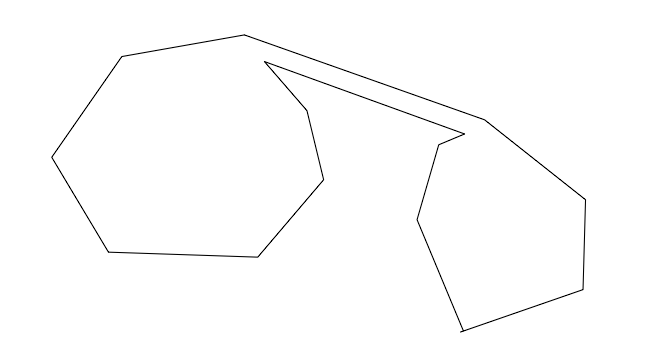
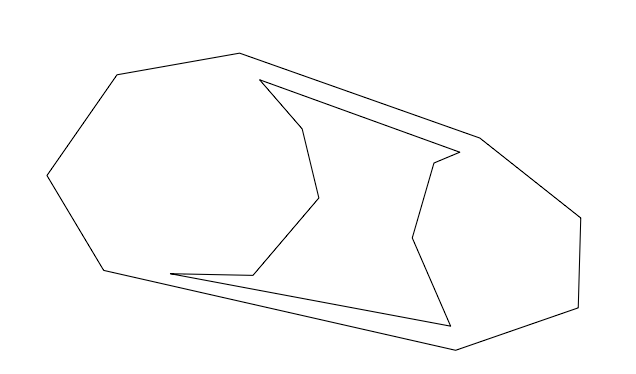
Функция voronoiDiagram формирует диаграмму Вороного для массива s из n точек. Она возвращает список многоугольников Вороного, из которых состоит диаграмма.

**Слияние оболочек**

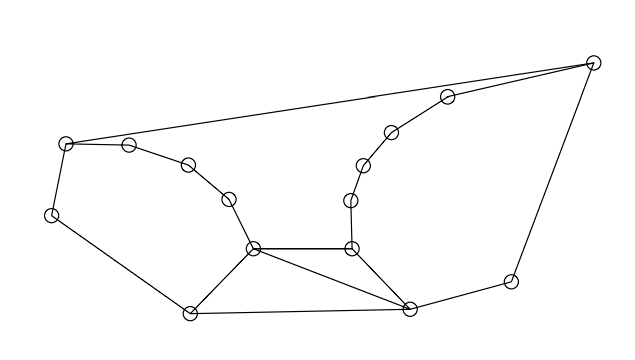
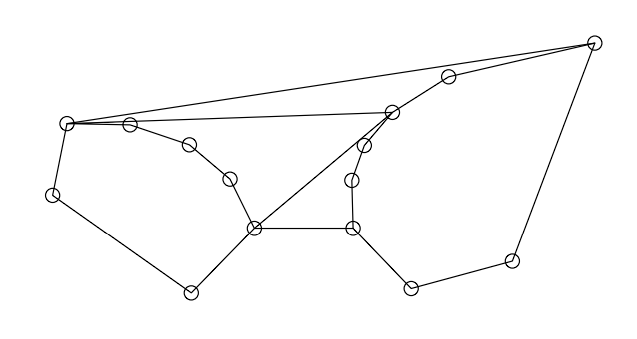
Для вычисления выпуклой оболочки для набора точек S, массив S воображаемой вертикальной линией делится на 2 равных по размеру поднабора, затем из них рекурсивно сформировываются выпуклые оболочки, которые в конечном счете объединяются при помощи мостиков.

****

*Слияние двух раздельных выпуклых оболочек.*

**** **** 

*Слияние выпуклых оболочек с помощью двух операций split.*

**

*Последовательность поиска верхнего и нижнего мостика для двух выпуклых полигонов.*

Функции mergeHull передается массив pts из n точек. Она возвращает полигон, представляющий выпуклую оболочку для этих точек.

Polygon \*mergeHull(Point pts[], int n)

{

Point \*\*p = new (Point\*[n]);

for (int i = 0; i < n; i++){

p[i] = &pts[i];

}

mergeSort(p, n, leftToRightCmp);

return mHull(p, n);

}

Функция mHull реализует рекурсивную часть алгоритма.

Polygon \*mHull(Point \*p[], int n)

{

if (n == 1) {

Polygon \*q = new Polygon;

q->insert(\*p[0]);

return q;

} else {

int m = n / 2;

Polygon \*L = mHull(p, m);

Polygon \*R = mHull(p+m, n-m);

return merge(L, R);

}

}

Функция leastVertex перемещает окно полигона p на наименьшую вершину и возвращает указатель на эту вершину

int leftToRightCmp(Point \*a, Point \*b)

{

if (\*a < \*b) return -1;

if (\*a > \*b) return 1;

return 0;

}

int rightToLeftCmp(Point \*a, Point \*b){

return leftToRightCmp(b, a);

}

Vertex \*leastVertex(Polygon &p, int (\*cmp)(Point\*,Point\*)){

Vertex \*bestV = p.v();

p.advance(Vertex::CLOCKWISE);

for (int i = 1; i < p.size(); p.advance(Vertex::CLOCKWISE), i++){

if ((\*cmp)(p.v(), bestV) < 0){

bestV = p.v();

}

}

p.setV(bestV);

return bestV;

}

Функции leftToRightCmp и rightToLeftCmp нужны для обнаружения самой правой и самой левой вершины.

Функция supportingLine перемещает окно полигона на вершину, которую находит.

void bridge(Polygon \*L, Polygon \*R, Vertex\* &vl, Vertex\* &vr, int type){

int sides[2] = { Point::LEFT, Point::RIGHT };

int indx = (type == UPPER) ? 0 : 1;

do {

vl = L->v();

vr = R->v();

supportingLine(L->point(), R, sides[indx]);

supportingLine(R->point(), L, sides[1-indx]);

} while ((vl != L->v()) || (vr != R->v()));

}

void supportingLine(Point &s, Polygon \*p, int side) {

int rotation = (side == Point::LEFT) ? Vertex::CLOCKWISE : Vertex::COUNTER\_CLOCKWISE;

Vertex \*a = p->v();

Vertex \*b = p->neighbour(rotation);

int c = b->classify(s, \*a);

while ((c == side) || (c == Point::BEYOND) || (c == Point::BETWEEN)) {

p->advance(rotation);

a = p->v();

b = p->neighbour(rotation);

c = b->classify(s, \*a);

}

}

Функция bridge находит мостик между выпуклыми полигонами L и R.

Функция merge определяет общую выпуклую оболочку для выпуклых полигонов L и R, и возвращает ее.

Polygon \*merge(Polygon \*L, Polygon \*R)

{

Vertex \*l1, \*r1, \*l2, \*r2;

Vertex \*vl = leastVertex(\*L, rightToLeftCmp);

Vertex \*vr = leastVertex(\*R, leftToRightCmp);

bridge(L, R, l1, r1, UPPER);

L->setV(vl);

R->setV(vr);

bridge(L, R, l2, r2, LOWER);

L->setV(l1);

L->split(r1);

R->setV(r2);

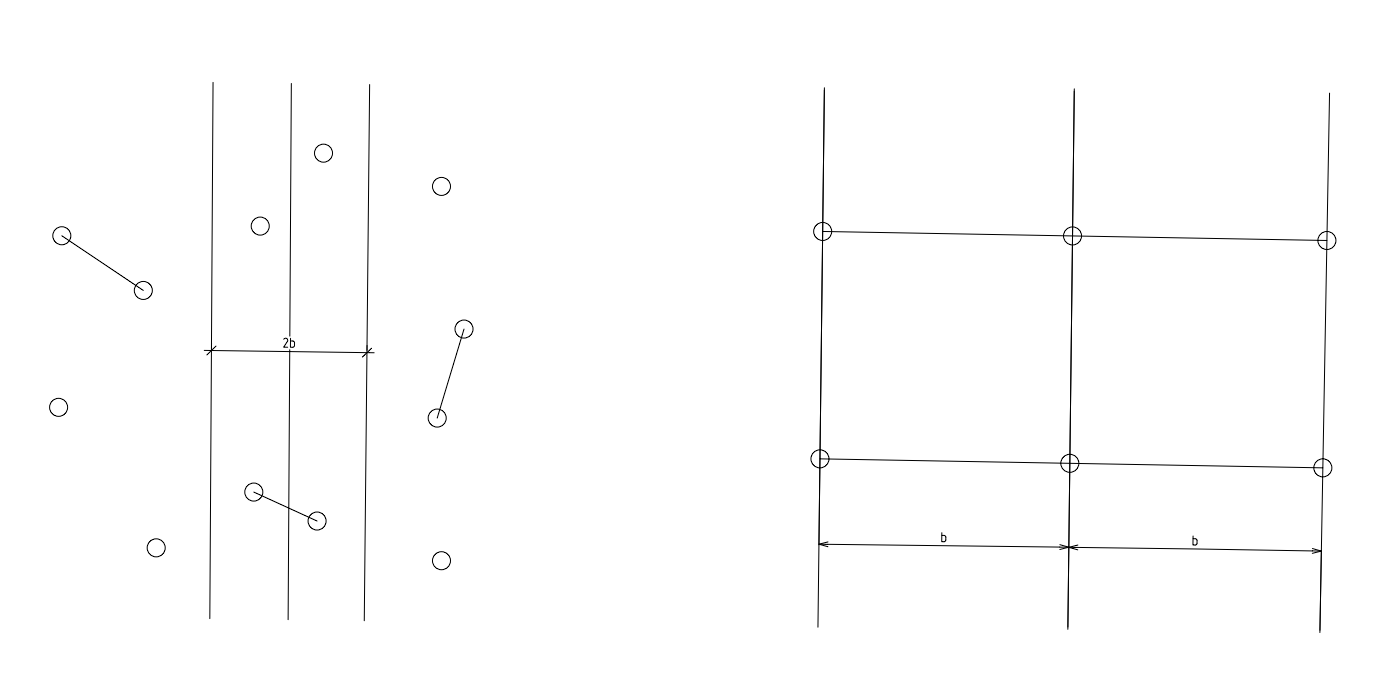
delete R->split(l2);

return R;

}

**Ближайшие точки**

Разделим множество точек на две подгруппы, используя вертикальную линию, которая делит их пополам. Рекурсивно найдем ближайшие пары точек в каждой из подгрупп. После этого найдем ближайшие пары точек, одна из которых находится в одной подгруппе, а другая - в другой. Это можно сделать путем проверки ближайших точек вдоль границы разделения. Наконец, найдем ближайшую пару точек, которая может находиться в разных подгруппах, и сравним ее расстояние с минимальным найденным расстоянием.

****

*Полоса шириной 2b. Полоса 2b x b.*

double closestPoints(Point s[], int n, Edge &c){

Point \*\*x = new (Point\*[n]);

Point \*\*y = new (Point\*[n]);

for (int i = 0; i < n; i++){

x[i] = y[i] =&s[i];

}

mergeSort(x, n, leftToRightCmp);

mergeSort(y, n, bottomToTopCmp);

return cPoints(x, y, n, c);

}

Функции closestPoints передается массив s из n точек. Она возвращает расстояние до ближайшей пары точек из массива s, а также некоторую ближайшую пару через ссылочный параметр с.

Функция bottomToTopCmp выполняет сортировку по увеличению координаты y.

double cPoints(Point \*x[], Point \*y[], int n, Edge &c) {

if (n == 1) {

return DBL\_MAX;

}

else {

int m = n / 2;

Point \*\*yL = new (Point\*[m]);

Point \*\*yR = new (Point\*[n-m]);

splitY(y, n, x[m], yL, yR);

Edge a, b;

double deltaL = cPoints(x, yL, m, a);

double deltaR = cPoints(x+m, yR, n-m, b);

delete yL;

delete yR;

double delta;

if (deltaL < deltaR) {

delta = deltaL;

c = a;

}

else {

delta = deltaR;

c = b;

}

return checkStrip(y, n, x[m], delta, c);

}

}

int bottomToTopCmp(Point \*a, Point \*b) {

if ((a->y<b->y) || ((a->y==b->y) && (a->x<b->x))) {

return -1;

}

else if ((a->y>b->y) || ((a->y==b->y) && (a->x>b->x))) {

return 1;

}

return 0;

}

В функцию cPoints передается массив x из n точек, отсортированных по координате x, и массив y тех же точек, отсортированных по координате y. Она возвращает расстояние до ближайшей пары из n точек, а также некоторую ближайшую пару через ссылочный параметр с.

void splitY(Point \*y[], int n, Point \*p, Point \*yL[], Point \*yR[]) {

int i, lindx, rindx;

i = lindx = rindx = 0;

while (i < n){

if (\*y[i] < \*p){

yL[lindx++] = y[i++];

}

else{

yR[rindx++] = y[i++];

}

}

}

Функция splitY делит массив y на два массива yL и yR.

Функция checkStrip осуществляет проверку тех точек массива y, которые лежат внутри полосы шириной 2\*delta, симметричной относительно вертикальной прямой линии, проходящей через точку p.

double checkStrip(Point \*y[], int n, Point \*p, double delta, Edge &c) {

int i, striplen;

Point \*s = new Point[n];

for (i = striplen = 0; i < n; i++) {

if ((p->x - delta < y[i]->x) && (y[i]->x < p->x + delta)) {

s[striplen++] = \*y[i];

}

}

for (i = 0; i < striplen; i++) {

for (int j = i+1; j < striplen; j++) {

if (s[j].y - s[i].y > delta){

break;

}

if ((s[i] - s[j]).length() < delta) {

delta = (s[i] - s[j]).length();

c = Edge(s[i], s[j]);

}

}

}

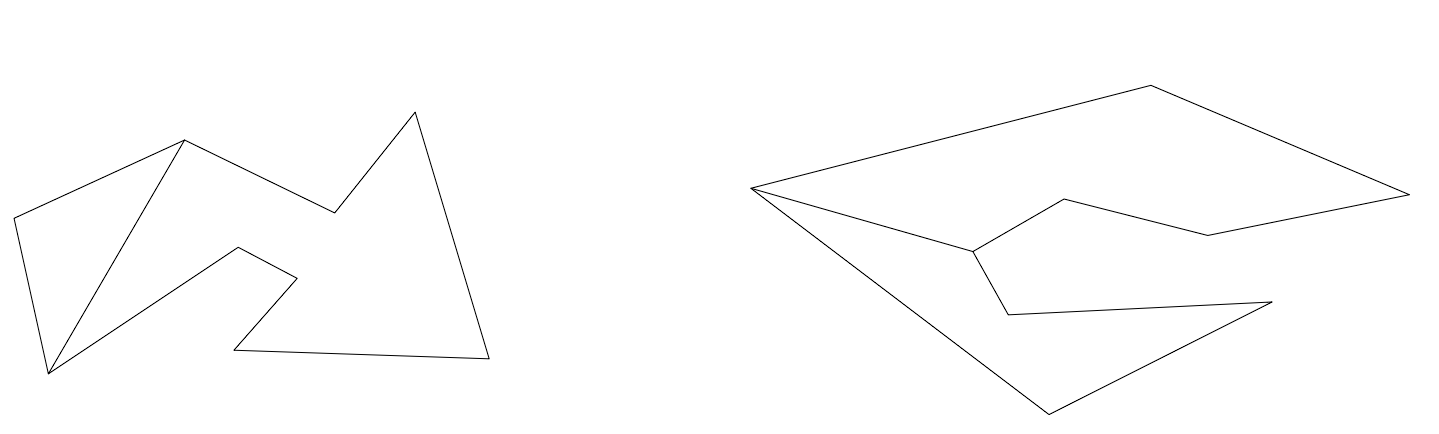
delete s;

return delta;

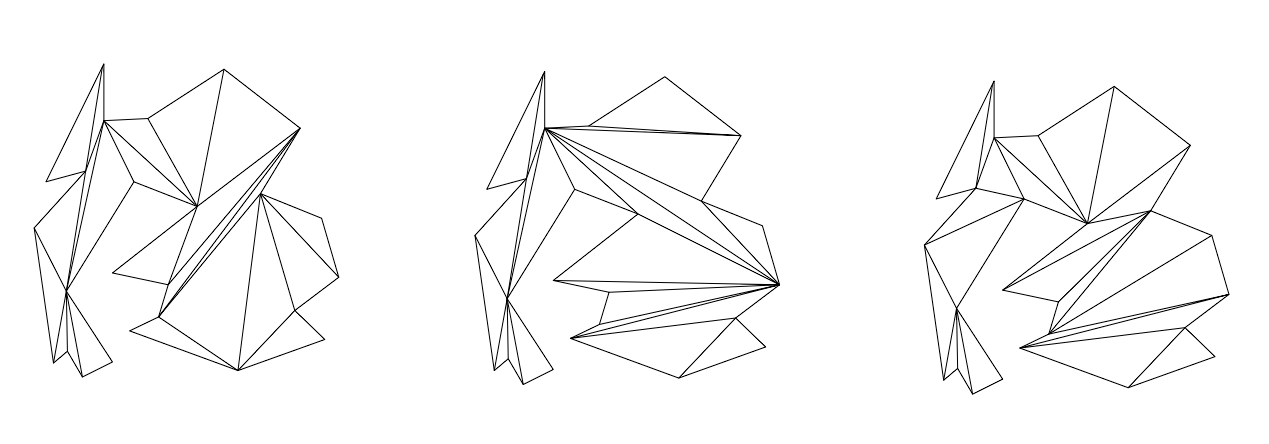
}

**Триангуляция полигона**

Идея заключается в расщеплении полигона P вдоль некоторой хорды и затем в рекурсивной триангуляции получившихся двух подполигонов. В конце возникает ситуация, когда подлежащий триангуляции полигон является треугольником.

****

*Две ситуации, рассматриваемые при доказательстве теоремы о триангуляции.*

**

*Три различных результата триангуляции, формируемых программой triangulate.*

Функции triangulate передается полигон p. Она возвращает список треугольников, представляющих его триангуляцию.

void findConvexVertex(Polygon &p){

Vertex \*a = p.neighbour(Vertex::COUNTER\_CLOCKWISE);

Vertex \*b = p.v();

Vertex \*c = p.neighbour(Vertex::CLOCKWISE);

while (c->classify(\*a, \*b) != Point::RIGHT) {

a = b;

b = p.advance(Vertex::CLOCKWISE);

c = p.neighbour(Vertex::CLOCKWISE);

}

}

List<Polygon\*> \*triangulate(Polygon &p){

List<Polygon\*> \*triangles = new List<Polygon\*>;

if (p.size() == 3){

triangles->append(&p);

}

else {

findConvexVertex(p);

Vertex \*d = findIntrudingvertex(p);

if (d == nullptr) {

Vertex \*c = p.neighbour(Vertex::CLOCKWISE);

p.advance(Vertex::COUNTER\_CLOCKWISE);

Polygon \*q = p.split(c);

triangles->append(triangulate(p));

triangles->append(q);

} else {

Polygon \*q = p.split(d);

triangles->append(triangulate(\*q));

triangles->append(triangulate(p));

}

}

return triangles;

}

Функция findConvexVertex перемещает окно в полигоне p на некоторую выпуклую вершину.

В функцию findIntrudingvertex передается полигон p, для которого предполагается, что его текущая вершина является выпуклой. Функция возвращает указатель на вторгающуюся вершину, если она существует или NULL в противном случае.

bool pointInTriangle(Point p, Point a, Point b, Point c){

return ((p.classify(a, b) != Point::LEFT) && (p.classify(b, c) != Point::LEFT) && (p.classify(c, a) != Point::LEFT));

}

Vertex \*findIntrudingvertex(Polygon &p)

{

Vertex \*a = p.neighbour(Vertex::COUNTER\_CLOCKWISE);

Vertex \*b = p.v();

Vertex \*c = p.advance(Vertex::CLOCKWISE);

Vertex \*d = nullptr;

double bestD = -1.0; // distance to best candidate

Edge ca(c->point(), a->point());

Vertex \*v = p.advance(Vertex::CLOCKWISE);

while (v != a) {

if (pointInTriangle(\*v, \*a, \*b, \*c)) {

double dist = v->distance(ca);

if (dist > bestD) {

d = v;

bestD = dist;

}

}

v = p.advance(Vertex::CLOCKWISE);

}

p.setV (b);

return d;

}

Функция pointInTriangle проверяет точки на попадание внутрь треугольника.