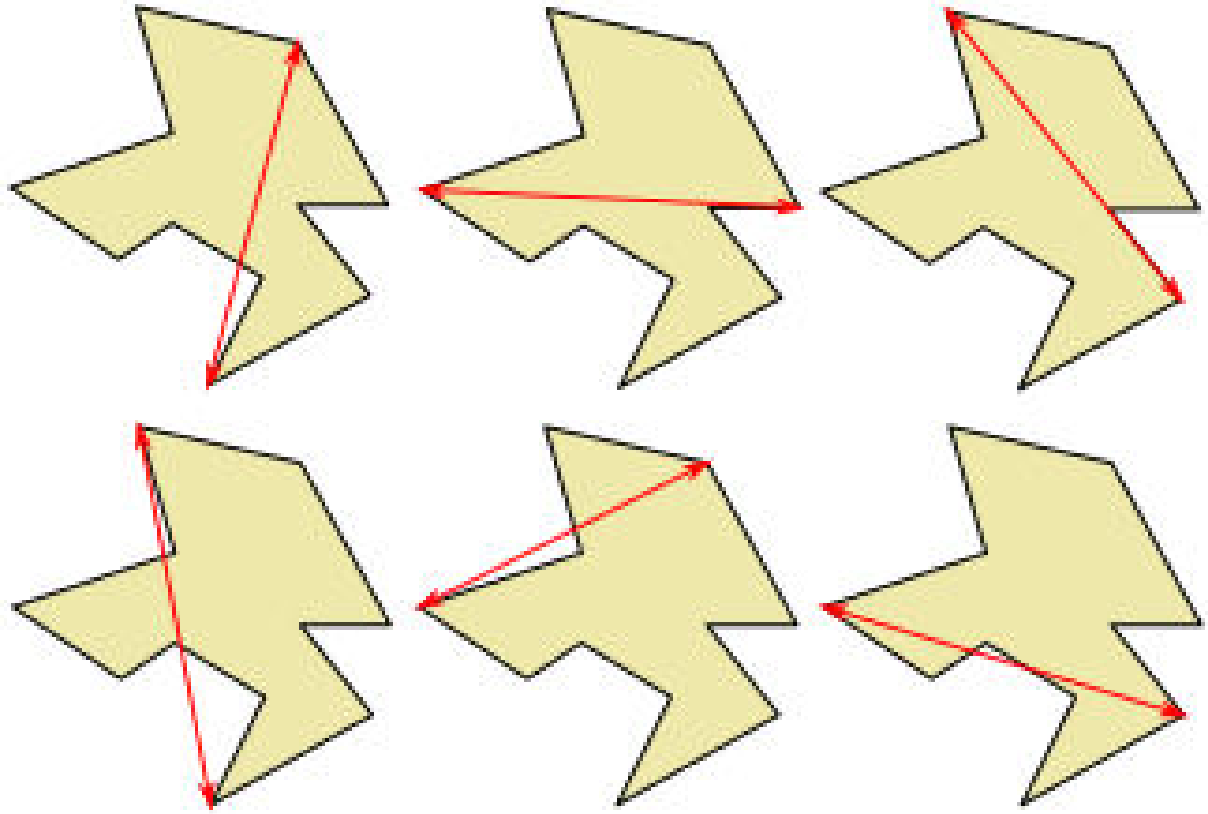
Geometrijski algoritmi @ MATF

Pronalaženje pravougaonika minimalne površine koji obuhvata ceo dati poligon metodom rotirajućih šestara

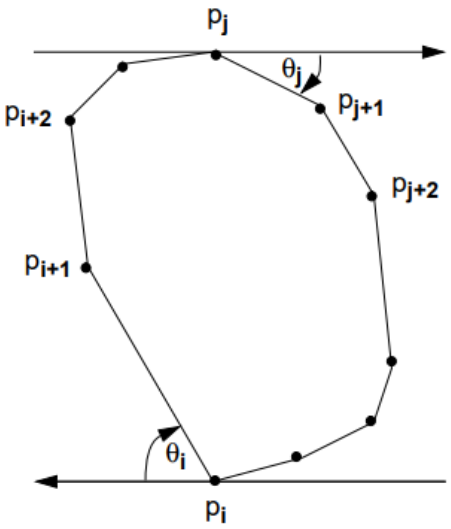
Kristina Stanojević



Opis problema

"Rotirajući šestari" (eng. Rotating calieprs) predstavljaju metodu kojom se mogu rešiti mnogi geometrijski problemi. Prvi put je ova ideja izneta 1978. godine u doktoratu Majkla Šejmosa (eng. Michael Shamos) u kome je on pokazao da se prečnik konveksnog poligona

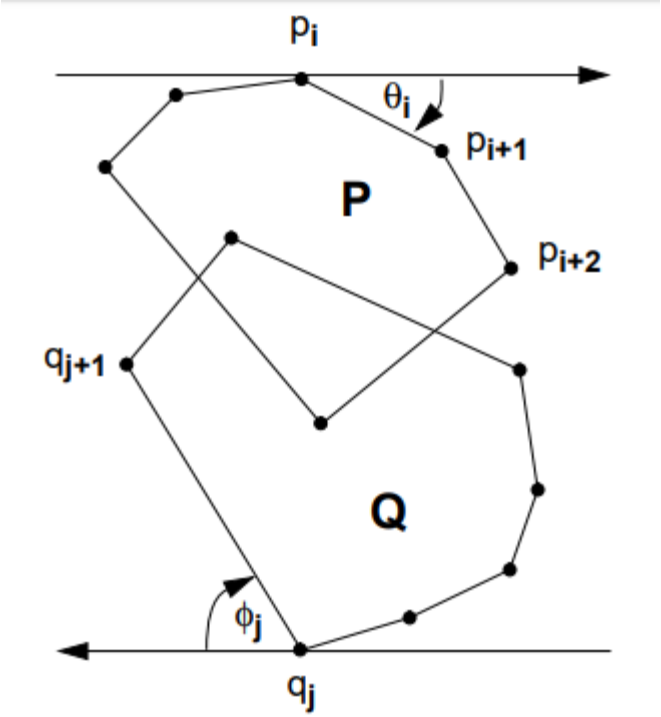
sa n stranica može naći tim algoritmom u složenosti O(n). Procedura pronalazi prečnik rotacijom šestara oko celog poligona. Ova ideja omogućava nam da algoritmom složenosti O(n) rešavamo različite geometrijske probleme kao što su pronalaženje pravougaonika minimalne površine koji obuhvata ceo dati poligon, računanje maksimalnog rastojanja između dva poligona, računanje sumirajućeg vektora dva poligona, spajanje poligona u jedan konveksni omotač i pronalaženje kritičnih linija podrške između dva poligona itd.



Neka nam je dat konačan skup tačaka p_1, p_2, ..., p_n. Neka je P = (p_1, p_2, ..., p_n) konveksni poligon sa n temena tako da nikoje tri tačke nisu kolinearne. Potrebno je naći prečnik poligona P, a to je najveće rastojanje između paralelnih pravih poligona P (vidi sliku). Te prave rotiraju oko poligona i zastanu svaki put kada dođu do dva temena poligona. Tako se pronalaze antipodalni parovi datog poligona nakon čega se utvrdi koji par je najudaljeniji. Pošto ova procedura liči na rotiranje šestara oko poligona, po tome je i dobila naziv. Vreme potrebno za pronalaženje antipodalnih parova je O(n) nakon čega je pretraga dobijenog niza za pronalazak najdaljeg para ne menja složnost. U ovom radu najpre će biti ukratko opisani već navedeni problemi koji se mogu rešiti ovom procedurom, a sam akcenat će biti na pronalasku najmanjeg pravougaonika koji sadrži neki dati poligon.

Drugi problemi koji se mogu rešiti metogom rotirajućih šestara

1. Najveće rastojanje između dva poligona:



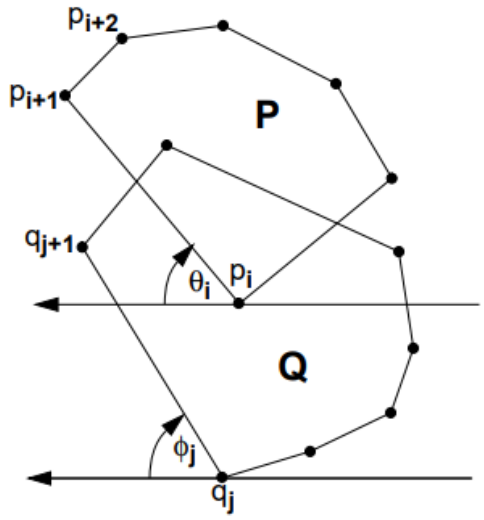
Neka su P = (p_1, … p_n) i Q = (q_1, … q_n) dva konveksna poligona. Najveće euklidsko rastojanje između njih definisano je formulom

$$d_{max}(P,Q) = \max_{i,j}\{d(p_i, q_j)\}$$

za i, j = 1, 2, … n. Korišćenjem para šestara kao na slici dobili bismo jednostavno rešenje. Parlelnim pravama na poligona P i Q tražili bismo antipodalne parove temena između ta dva poligona. Procedura bi išla poput osnovne ideje pronalaska prečnika jednog poligona, uz neke izmene zbog različitosti pojmova prečnika i najdaljeg rastojanja (ne može se koristiti osnovni algoritam za problem P U Q).

2. Sumirajući vektor dva poligona:

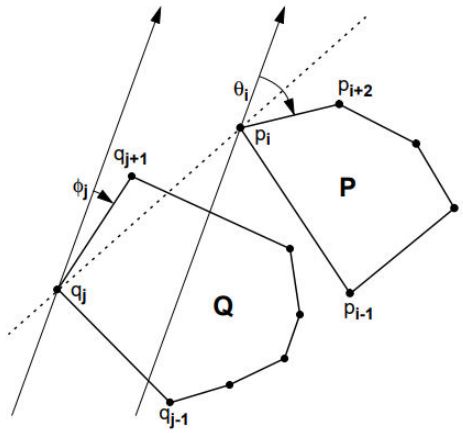
Pomoću ovih vektora rešavaju se problemi kod poligona kao što je izbegavanje kolizije. Neka su P i Q dva konveksna poligona. Označimo tačke na poligonima P i Q redom sa r = (x_r, y_r) i s = (x_s, y_s). Sumirajući vektor skupa tačaka ova dva poligona, u oznaci $P \oplus Q$, predstavlja skup koji dobijamo tako što dodajemo svaku tačku iz Q svakoj tački iz P. $P \oplus Q$ predstavljaće takođe konveksni poligon sa ne više od 2n temena.



Nekim algoritmima se $P \oplus Q$ može naći u O(n logn) vremenu, dok se metodom rotirajućih šestara računa sa O(n) složenošću. Dva temena p_i iz P, q_j iz Q biće kopodalni parovi tačaka u položaju prikazanom na slici. Oslanjajući se na malopređašnju teoremu koja govori da temena poligona $P \oplus Q$ predstavljaju sumirajuće vektore kopodalnih parova od P i Q, možemo koristiti upravo samo takve parove tačaka. Koristeći rotirajuće šestare možemo konstruisati $P \oplus Q$ dok tražimo kopodalne parove tačaka.

Svako teme poligona $P \oplus Q$ može se konstruisati u O(1) vremenu. Pošto imamo najviše 2n temena tog poligona, ukupla složenost kreiranja $P \oplus Q$ biće O(n).

3. Spajanje konveksnih poligona:



Spajanje dva konveksna poligona P i Q vrši se pronalaženjem dva para temena p_i, p_j i q_k, q_l takvih da nove ivice p_iq_k i q_lp_j zajedno sa preostalim ivicama q_k, q_{k+1}, ..., q_l i p_j, p_{j+1}, ..., p_i formira konveksni poligon P U Q. Ivica p_iq_k se u ovom slučaju naziva *most*, a temena p_i i q_k su *temena mosta*. Pronalazak mosta dva disjunktna poligona se može pronaći koristeći metodu rotirajućih šestara.

Koristeći teoremu koja govori o tome da su dva temena p_i i q_j (sa različitih poligona) temena mosta ako i samo ako predstavljaju kopodalni par tačaka i ako njima susedna temena leže na istoj strani prave koja prolazi kroz temena p_i i q_j, možemo konstruisati odgovarajući algoritam spajanja konveksnih poligona. Na slici imamo tačke p_i i q_j koje jesu kopodalne (takve parove nalazimo tokom ''rotiranja šestara'') ali njima susedna temena nisu sa iste strane prave koja prolazi kroz temena koja razmatramo. U tom slučaju znamo da p_iq_j nije most. Algoritam se zaustavlja kada je most pronađen. Složenost ovakvog algoritma je dakle O(n).

4. Pronalaženje kritičnih linija podrške poligona:

Neka su dati konveksni poligoni P i Q. Prava L(p_i, q_j) biće njihova kritična linija podrške ako predstavlja liniju podrške i jednom i drugom poligonu (u temenima p_i i q_j koji su antipodalni) i ako se poligoni P i Q nalaze na suprotnim stranama prave L. Algoritmi za rešavanje ovog problema koriste se u različitim praktičnim problemima vidljivosti objekata ili izbegavanja kolizije... Antipodalne parove pronalazimo u O(1), tako da je ukupna složenost O(n).

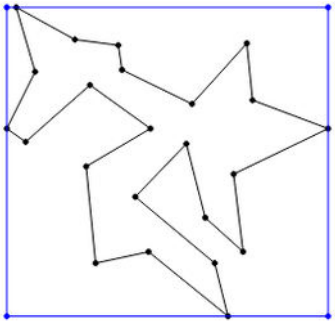
Ulaz: *skup od n tačaka u ravni*

Izlaz: *skup tačaka koje predstavljaju temena najmanjeg pravougaonika*

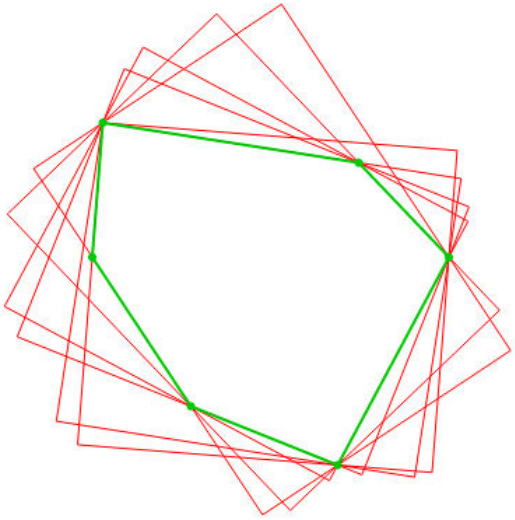
Naivno rešenje problema

Algoritmi za rešavanje problema najmanjeg pravougaonika koji sadrži dati poligon se koriste, na primer, za procesiranje slika. Ideja korišćenja rotirajućih šestara za rešavanje ovog problema zasnovana je na teoremi koja kaže sledeće: jedna strana traženog pravougaonika najmanje površine (koji sadrži dati konveksni poligon) je kolinearna jednoj ivici tog ulaznog poligona.

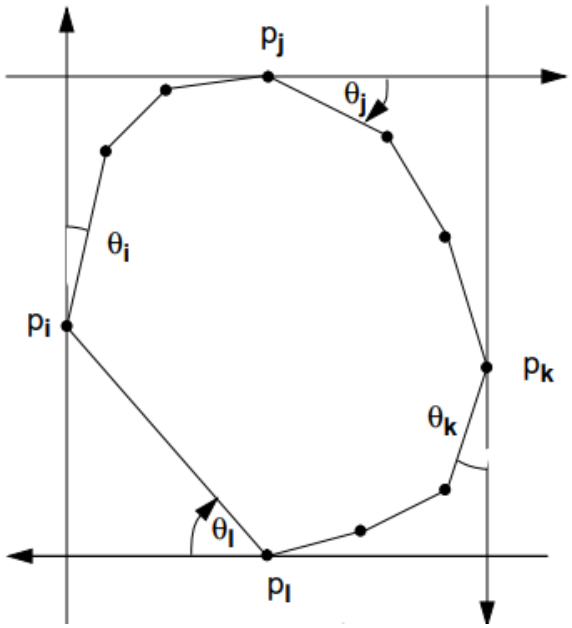
Nakon ove teoreme zaključujemo da je za bilo koji poligon potrebno razmatrati samo njegov konveksni omotač, čime se smanjuje broj pravougaonika (mogućih kandidata) od kojih je jedan traženo rešenje. Što se složenosti tiče, mogući pravougaonici se, dakle, mogu naći u O(n) vremenu nakon čega se bira najmanji, tako da je ukupna složenost O(n^2).



Rotating Calipers algoritam



Koristeći rotirajuće šestare to se može nad datim konveksnim poligonom rešiti u složenosti O(n). Neka je L_s(p_i) linija podrške poligona P u temenu p_i koju usmeravamo tako da se poligon P nalazi sa njene desne strane. Neka prava L(p_i, p_j) prolazi kroz temena p_i i p_j. Na početku tražimo temena sa najmanjom i najvećom x i y koordinatom. Neka su takva tražena temena kao na slici, redom, p_i, p_k, p_l, p_j.



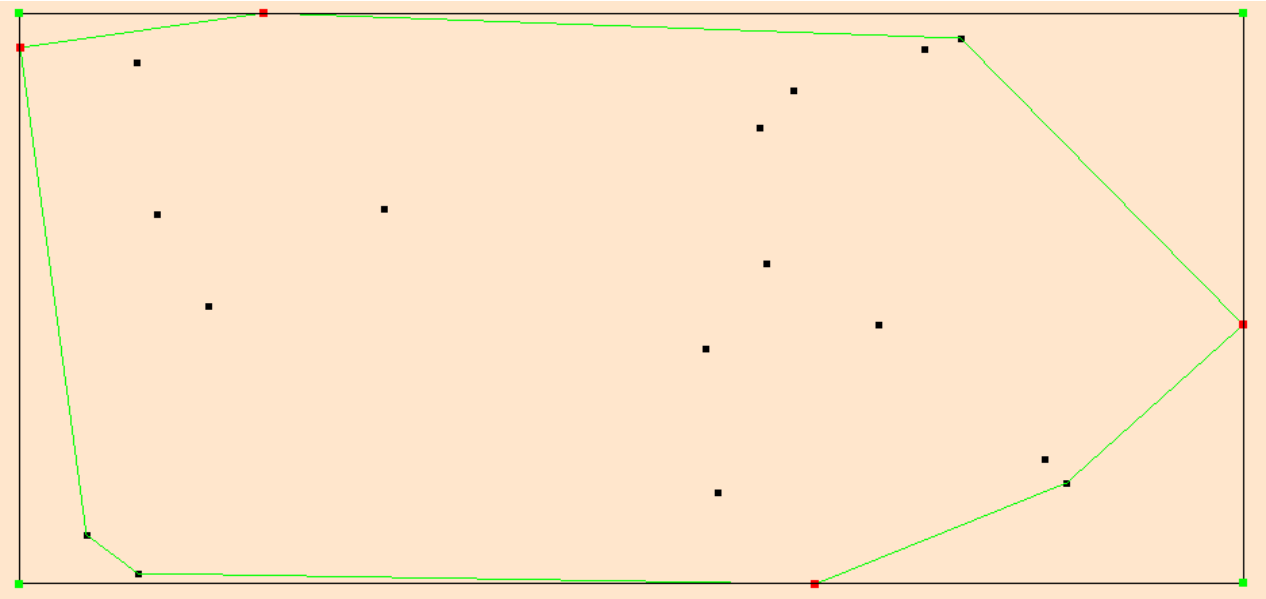
Zatim konstruišemo L_s(p_j) i L_s(p_l) kao prvi par linija šestara usmerenih poput x-ose, i L_s(p_i) i L_s(p_k) kao drugi par šestara usmerenih ortogonalno u odnosu na prvi par. Posmatrajući četiri odgovarajuća ugla, tražimo θ_i = min{ θ _i, θ _j, θ _k, θ _l}. Zatim rotiramo polazne prave za ugao θ _i. Nakon rotacije jedna linija šestara L(p_i, p_{i+1}) poklapaće se sa ivicom konveksnog poligona p_ip_{i+1} i predstavljaće jednu ivicu traženog pravougaonika. Ponavljajući postupak rotacije oko poligona, sačuvamo sve dobijene pravougaonike. Pretragu završavamo kada se svaka ivica konveksnog poligona jednom poklopila sa nekom od četiri linija šestara. Na kraju biramo onaj pravougaonik čija je površina najmanja. Površina pravougaonika se može naći u konstantnom vremenu, tako da ceo algoritam ima složenost O(n).

Vizuelizacija algoritma (opciono)

```
//cuvamo trenutni pravougaonik, vektor tacaka, i skup svih pravougaonika, vektor vektora tacaka
std::vector<QPoint> rectangle;
std::vector <std::vector<QPoint>> rectangles;

//inicijalizujemo uglove pocetnih calipera tako da su paralelni sa x i y osom
Caliper caliper_left = Caliper(_convexHull, _index_left, 270);
Caliper caliper_right = Caliper(_convexHull, _index_right, 90);
Caliper caliper_up = Caliper(_convexHull, _index_up, 0);
Caliper caliper_down = Caliper(_convexHull, _index_down, 180);

while(caliper_up._angle < 90.0)
{
    A = caliper_left.findIntersectionPoint(caliper_up);
    B = caliper_right.findIntersectionPoint(caliper_up);
    C = caliper_right.findIntersectionPoint(caliper_down);
    D = caliper_left.findIntersectionPoint(caliper_down);

    rectangle.push_back(A);
    rectangle.push_back(B);
    rectangle.push_back(C);
    rectangle.push_back(D);

    _AB = QLine(A, B);
    _BC = QLine(B, C);
    _CD = QLine(C, D);
    _DA = QLine(D, A);
    AlgorithmBase_updateCanvasAndBlock();

    rectangles.push_back(rectangle);

    //nadji najmanji ugao, rotiraj
    double smallestAngle = findSmallestAngle(caliper_down, caliper_up, caliper_left, caliper_right);
    caliper_down.rotate(smallestAngle);
    caliper_up.rotate(smallestAngle);
    caliper_left.rotate(smallestAngle);
    caliper_right.rotate(smallestAngle);
}
```

U kodu je definisana klasa Caliper koja sadrži dobijeni konveksni omotač, indeks tačke iz konveksnog omotača kroz koju dati šestar prolazi i ugao koji pravi sa x-osom. U metodi za pokretanje algoritma najpre su pronadjene polazne tačke i inicijalizovani uglovi četiri šestara na 0, 90, 180 u 270 stepeni. Pomoću funkcije QPoint findIntersectionPoint(Caliper caliper) traže se presečne tačke tih šestara koje će predstavljati temena pravougaonika koji oni kreiraju. Dalje funkcija double RotatingCalipers::findSmallestAngle(Caliper c1, Caliper c2, Caliper c3, Caliper c4) pronalazi najmanji ugao θ za koji će se rotirati svi šestari. Ona u sebi ima poziv funkcije double calculateAngle(), koja se poziva za svaki šestar pojedinačno. Sama rotacija se izvršava pozivom funkcije void rotate(double angle) unutar klase Caliper. Ta funkcija svakom šestaru poveća ugao za nađeni najmanji ugao θ, a samo onom šestaru kod koga je pronađen taj najmanji ugao θ promeni indeks tačke na sledeći indeks iz niza tačaka

konveksnog omotača, tako da samo taj šestar prelazi na drugu tačku. Na slici je prikazan inicijalizovan početni položaj.



Za računanje presečnih tačaka tj. temena pravougaonika (zelenom značeno) koristi se sledeća ideja: za svaku pravu važi da tgθ = (y2-y1)/(x2-x1) gde je taj tangens zapravo koeficijent pravca prave. Pošto mi imamo za svaki šestar dat ugao, znamo i njegov tangens. Takođe imamo jednu tačku na pravoj, a to je teme konveksnog omotača, dakle (x2, y2) nam je iz formule poznato. Tačka (x1, y1) biće teme pravougaonika. S druge strane istom formulom predstavićemo tangens jos jednog, susednog, šestara. Dakle rešavanje sistema za gornji i desni šestar dobijamo gornje desno teme pravougaonika.

Računanje najmanjeg ugla za koji ćemo rotirati sve šestare se vrši tako što od ugla trenutnog šestara oduzmemo ugao sledećeg mogućeg šestara. Ugao sledećeg mogućeg šestara nalazimo pomoću funkcije atan2(y, x) koja računa pod kojim uglom je nagnuta prava sa temenom (x, y) što je u našem slučaju neko od temena šestara.

```
/*odvojeno dodatno racunanje za racunanje presecne tacke - temena*/
double additionalCalculation()
{
    QPoint point = _convexHull.at(_pointIndex);
    return slope() * point.x() - point.y();
}


/*trazi presecnu tacku dva kalipera tj. teme pravougaonika*/
QPoint findIntersectionPoint(Caliper caliper)
{
    double x, y;

    /*koriste se izvedene formule iz sistema jednacina tgA = (y1 - y) / (x1 - x) i tbB = (y2 - y) / (x2 - x)
    nakon sredjivanja:
    x = (tgA*x1 - y1 + tgB*x2 - y2) / (tgB + tgA)
    y = (tgA*(tgB*x2 - y2) - tgB*(tgA*x1 - y1)) / (tgB - tgA) */

    x = (caliper.additionalCalculation() + this->additionalCalculation()) / (caliper.slope() + this->slope());
    y = (caliper.slope() * this->additionalCalculation() - this->slope() * caliper.additionalCalculation()) / (this->slope() - caliper.slope());

    return QPoint(x, y);
}
```

```
/*racuna ugao koji pravi trenutni caliper sa mogucim sledecim caliperom*/
double calculateAngle()
{
    //ugao izmedju jednog calipera i sledeceg calipera (koji bi nastao
    //nakon rotacije tog pocetnog) se moze naci pomocu funkcije
    //atan2(y, x) koja vraca ugao izmedju x-ose i vektora (y, x),
    //a to je nama tacka kroz koju prolazi caliper.
    //Onda je ugao izmedju dva calipera zapravo angle = alfa2 - alfa1
    //odnosno angle = atan2(y2, x2) - ugaoPoznatogCalipera

    int thisIndex = _pointIndex;
    int nextIndex = (_pointIndex + 1) % _convexHull.size();

    QPoint caliperPoint = _convexHull.at(thisIndex);
    QPoint nextPoint = _convexHull.at(nextIndex);

    double angleNextCaliper;

    double deltaX = nextPoint.x() - caliperPoint.x();
    double deltaY = nextPoint.y() - caliperPoint.y();

    //trazimo ugao koji bi zaklapao sledeci susedni caliper sa x-osom
    angleNextCaliper = atan2(deltaY, deltaX) * 180 / M_PI;
    //normalizacija
    double rotatingAngle = angleNextCaliper < 0? 360 + angleNextCaliper : angleNextCaliper;

    //kao sto je vec receno,
    //ugao izmedju trenutnog i sledeceg calipera se dobija kada se od
    //ugla koji sledeci caliper zaklapa sa x-osom oduzme ugao
    //trenutnog calipera sa x-osom
    rotatingAngle = rotatingAngle < 0? 360 + rotatingAngle - _angle : rotatingAngle - _angle;

    return rotatingAngle < 0 ? 360 : rotatingAngle;
}
```

Poredjenje efikasnosti naivnog i naprednog algoritma

Ovde tasdreba da afbude dat tabelarni i/ili grafički prikaz brzine izvršavanja oba algoritma u zavisnosti od veličine ulaza

Testiranje ispravnosti algoritma

Ovde treba da bude opisano na koji način je izvršeno testiranje algoritma.

Naziv testa	Opis testa	Ulaz	Očekivani izlaz
WrongInput1	Zadavanje ulaza koji nije ispravan. Program ce ispisati poruku o gresci, a rezultujuci niz treba da bude prazan.	[{1,2}, {2,3}]	[]
ThreePoints	....	[neki ulaz]	[ocekivani izlaz]
RandomInput1	...	Niz dimenzije 30	Poklapanje rezultata naivnog i naprednog algoritma
RandomInput2	...	Niz dimenzije 1000	Poklapanje rezultata naivnog i naprednog algoritma
...	...	...	...