Geometrijski algoritmi @ MATF

Određivanje rasporeda čuvara u poligonalnom dvodimenzionom prostoru.

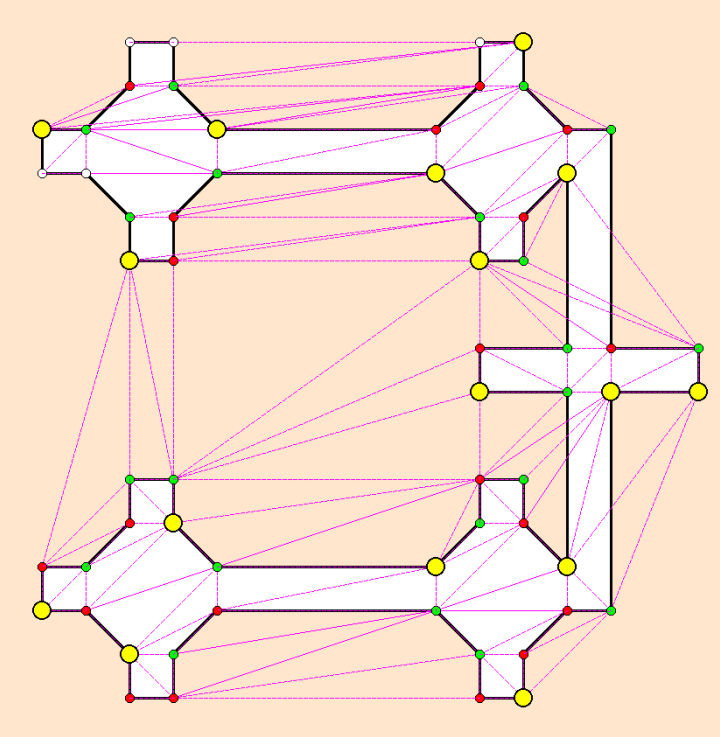
Božidar Radivojević

Opis problema

Prost poligon je definisan u ravni tako da predstavlja zidove poligonalnog prostora. Potrebno je odrediti raspored čuvara, tako da ceo prostor bude pod nadzorom. Čuvar ima vidno polje od 360 stepeni, i može biti pozicioniran bilo gde u unutrašnjosti prostora, ili na unutrašnjem zidu.

Ulaz: niz koordinata temena poligona, navedenih prema susedstvu, posmatrano u pozitivnom matematičkom smeru.

Izlaz: niz koordinata tačaka u kojima treba da budu pozicionirani čuvari, tako da ceo prostor bude pod nadzorom.



Slika 1 – Primer vizuelizacije krajnjeg rezultata algoritma.

Rešenje problema

Unutar svakog prostog poligona moguće je izvršiti triangulaciju (podeliti poligon na trouglove), tako da temena poligona budu ujedno i temena triangulacije. Da triangulacija uvek postoji, lako se dokazuje indukcijom po broju temena poligona. Time smo efektivno prostor podelili na skup konveksnih podprostora. Ukoliko se u temenu jednog trougla triangulacije postavi čuvar, on će imati pregled svih trouglova kojima je susedan.

Sledeće što možemo primetiti jeste da, ukoliko je triangulacija takva da ivice triangulacije nemaju presečnih tačaka sa ivicama prostora, tada je moguće izvršiti tri-bojenje triangulacije. Tri-bojenje podrazumeva da se svakom temenu triangulacije dodeli tačno jedna od tri prethodno definisane boje, uz uslov da u svakom trouglu, sva temena moraju biti obojena međusobno različitim bojama.

Ukoliko je poligon čija se triangulacija vrši prost, i ukoliko ivice triangulacije ne seku ivice prostora, tada je moguće konstruisati stablo, čija temena odgovaraju poljima triangulacije, a ivice povezuju

temena koja odgovaraju susednim poljima. Dokaz da tri-bojenje postoji je ujedno i opis procedure za njegovo izvršavanje: počevši od proizvoljnog lista drveta, bojenje vršimo tako što prelazimo u sledeće polje i bojimo neobojeno teme do tada nekorišćenom bojom. S obzirom na način konstrukcije stabla, u svakom momentu obilazimo trouglove kojima su dva temena već obojena, tako da na potpuno deterministički način određujemo kojom bojom treba obojiti treće teme.

Nakon izvršenog tri-bojenja, dovoljno je odabrati boju kojom je obojeno najmanje temena, a zatim ta temena proglasiti čuvarima. Kako se u svakom trouglu triangulacije prostora nalazi jedno teme obojeno tom bojom, to je i svaki deo prostora pod nadzorom, što rešava za dati problem.

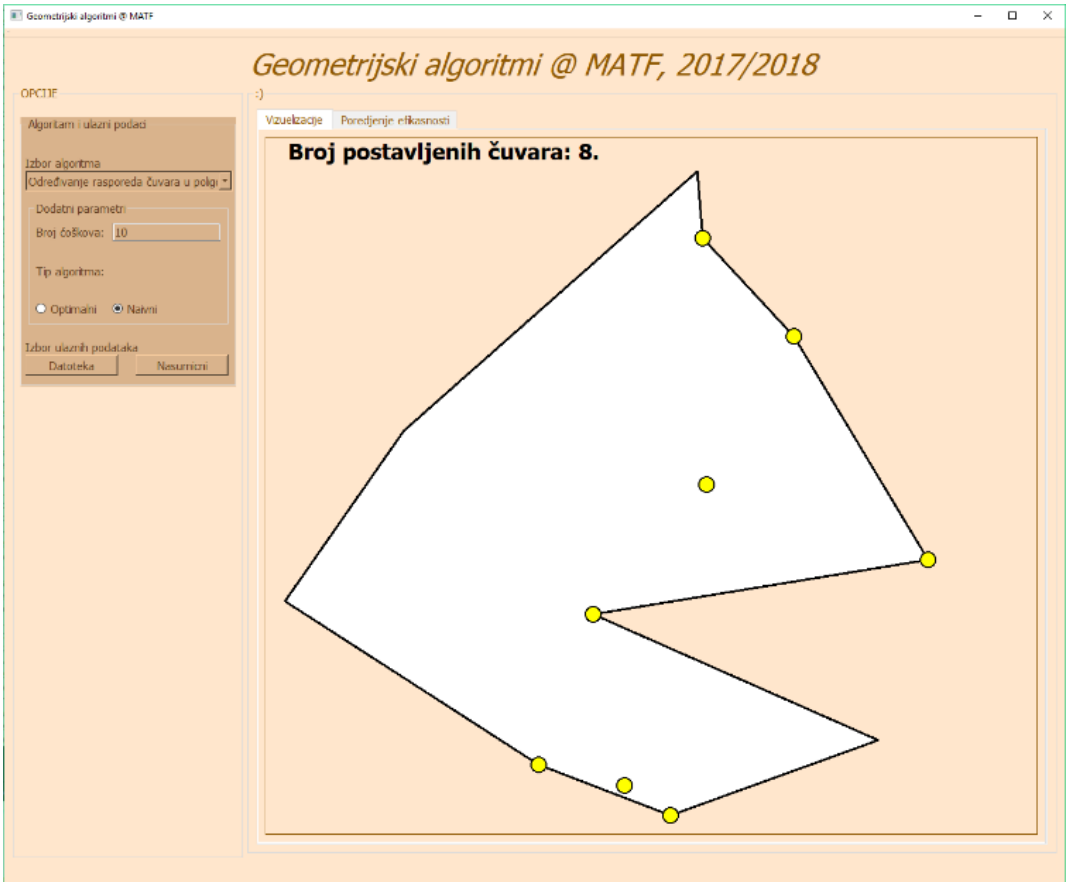
Moramo primetiti da broj čuvara određenih ovom metodom ne mora biti optimalan. Naime, lako je zamisliti centralno simetričnu prostoriju u kojoj bi čuvar smešten u centru simetrije imao uvid u svaki deo prostora, dok bi naš algoritam koji se oslanja na tribojenje, postavio u svakom slučaju više od jednog čuvara.

Naivno rešenje problema

Glavno rešenje ovog problema zasnovano je na triangulaciji i tri-bojenju, i uvek daje rezultat, koji ne mora biti optimalan. Stoga „naivnost" nije imalo smisla postići kreiranjem lošijeg algoritma koji bi radio korišćenjem istih principa, već je kreiran „Las Vegas" algoritam. Naime, slučajno se odabira lokacija čuvara, zatim se proverava da li postoji deo prostora koji do sada nije pokriven. Ukoliko ne postoji, algoritam završava sa radom i vraća listu do sada odabranih čuvara. Vidljivost prostora se, imajući u vidu da se radi o prostim poligonalnim prostorima, odredjuje proverom vidljivosti „ćoškova" odnosno temena poligona. Jasno je da se radi o algoritmu koji potencijalno može da nikada ne nađe tačno rešenje, naročito ukoliko prostor ima „uske hodnike" koje je teško slučajno pogoditi.

```
Algoritam: runNaiveAlgorithm(P)
      Ulaz: skup koordinata ćoškova prostorije P
      Izlaz: skup odabranih čuvara S
      1.    inicijalizuj  S kao praznu kolekciju
      2.    inicijalizuj koleciju V temena koja su do sad vidljiva kao praznu
```

```
2.    while count(V) <> count(P)
3.          dodaj novog slučajno odabranog čuvara s u S
4.          forall p iz P/V do
5.                if Vidljiv(s, p) then
6.                      dodaj p u V
7.    return S
```



Slika 2 – Prikaz završetka rada naivnog algoritma

Napredno rešenje problema

Napredni pristup rešavanju problema je zasnovan na kombinaciji triangulacije sa algoritmom tri-bojenja, kao što je već navedeno u opštem opisu.

Algoritam: runAlgorithm(**P**)
Ulaz: skup koordinata ćoškova prostorije **P**
Izlaz: skup odabranih čuvara **S**

```
1.    initTriangulation(T)
2.    forall p iz P do
3.          insertVertexToTriangulation(p, T)
4.    doThreeColoring(T)
5.    selectFinalSentries(T, S)
6.    return S
```

Ovakav algoritam, predložen u literaturi, podrazumeva upotrebu triangulacije koja prvo deli poligon na parcijalno monontone delove, a potom triangulaciju vrši obilaskom temena na specifičan način. Ovakva triangulacija zadovoljava sve uslove postavljene problemom pretrage čuvara (poštuje ivice poligona, postoji obilazeći graf koji nema cikluse – postoji tri-bojenje). Međutim, kako je takva forma triangulacije već obrađivana na časovima vežbi, predložena je upotreba druge vrste triangulacije – Delone triangulacije.

Delone triangulacija skupa tačaka podrazumeva povezivanje tačaka linijama, tako da se dobije mreža trouglova koja je optimalna u smislu veličine uglova trouglova. Drugim rečima, ako definišemo sa *A(T)* funkciju koja slika triangulaciju u rastući niz unutrašnjih uglova trougla triangulacije, onda Delone triangulacija *T'* ima svojstvo da za svaku drugu triangulaciju *T* važi $A(T') \geq_{lex} A(T)$, gde je operacija $\geq_{lex}$ leksikografsko poređenje niza brojeva. Rečeno i na treći način: Delone triangulacija maksimizuje minimalni ugao. Rezultat Delone triangulacije je mreža trouglova koja sadrži najmanji mogući broj uskih i dugačkih trouglova, sa velikim tupim uglom sa jedne strane. Algoritam za Delone triangulaciju, može se ugrubo opisati sledećim pseudo-kodom:

```
Algoritam: initTriangulation(T, P)
Ulaz: neinicijalizovana struktura T, skup tačaka P temena poligona
Izlaz: struktura T sa kojom je moguće započeti triangulaciju
1. Definišemo ograničavajući trougao
2.    Određujemo ekstremnu tačku r iz skupa P kao leksikografski najveću po
Y, X osama
3.    Kreiramo dve simboličke tačke p1 i p2
4. Inicijalizujemo DCELL strukturu D triangulacije T trouglom p2rp1
5. Inicijalizujemo stablo pretrage tačaka L korenom koji je uzajamno vezan sa
```

ograničenim poljem triangulacije **T**

6. Uklanjamo tačku **r** iz **P**, budući da je već obrađena u inicijalizaciji

7. Vršimo slučajno mešanje niza tačaka P, pošto želimo da tačke u triangulaciju dodajemo slučajnim redosledom

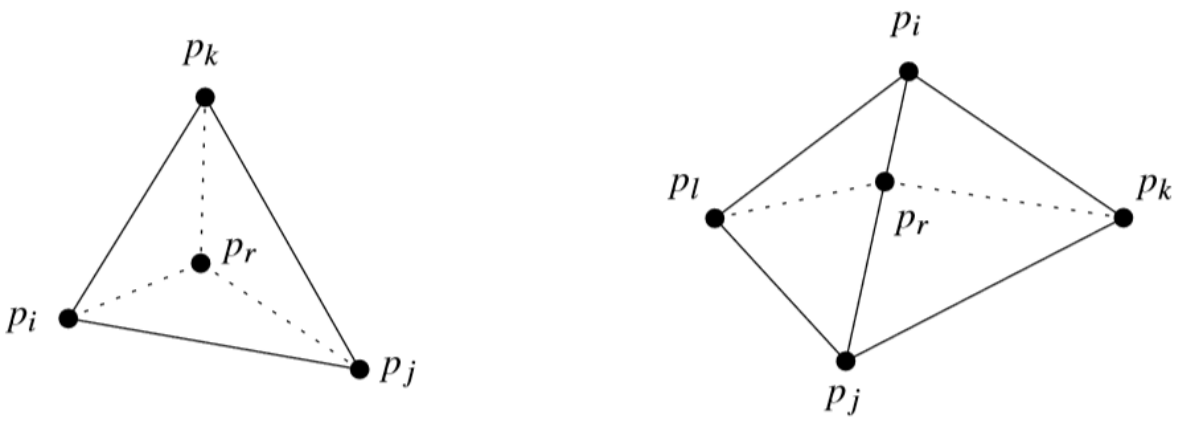
8. **return T**

Algoritam: insertVertexToTriangulation(**T**, **p**)

Ulaz: inicijalizovana struktura **T**, nova tačka koju treba dodati u triangulaciju **p**

Izlaz: modifikovana struktura **T**

1. Pronadji trougao $\mathbf{p_i p_j p_k}$ trenutne triangulacije koji sadrži tačku **p**
2. **if** p leži u strogoj unutrašnjosti trougla pipjpk
3. podeli trougao $\mathbf{p_i p_j p_k}$ na tri nova trougla, dodavanjem ivica $\mathbf{pp_i}$ $\mathbf{pp_j}$ $\mathbf{pp_k}$
4. legalizeEdge(**p**, $\mathbf{p_i p_j}$, **T**)
5. legalizeEdge(**p**, $\mathbf{p_j p_k}$, **T**)
6. legalizeEdge(**p**, $\mathbf{p_k p_i}$, **T**)
7. **else** ako **p** leži na nekoj od ivica, na primer na ivici $\mathbf{p_i p_j}$
8. dodaj nove ivice od tačke **p** do suprotnih tačaka $\mathbf{p_k}$ i $\mathbf{p_l}$ dva trougla koji dele ivicu $\mathbf{p_i p_j}$
9. podeli svaki od dva susedna trougla na po dva nova
10. legalizeEdge(**p**, $\mathbf{p_i p_l}$, **T**)
11. legalizeEdge(**p**, $\mathbf{p_l p_j}$, **T**)
12. legalizeEdge(**p**, $\mathbf{p_j p_k}$, **T**)
13. legalizeEdge(**p**, $\mathbf{p_k p_i}$, **T**)
14. Ukloni sve imaginarne ivice, tj. ivice koje imaju dodira sa imaginarnim tačkama.
15. **return T**



Slika 3 – prikaz dva moguća slučaja pri dodavanju temena p_r u triangulaciju

Kao što se može videti, u svakom koraku algoritma se pronalazi odgovarajuće mesto na koje treba umetnuti novu tačku, dodaju se potrebni elementi u DCELL, kao i deca čvora koji pokazuje na polje koje sadrži novu tačku. Ono što ostaje da se objasni je koncept legalizacije ivica. Naime, dodavanjem tačke u triangulaciju koja je Delone triangulacija, dodajemo nove ivice čije prisustvo može učiniti trenutno stanje ilegalnim. Stoga se susedne ivice obilaze redom, i „popravljaju" po potrebi, tako da se ponovo dobije Delone triangulacija. Legalnost ivice u četvorouglu koji nema simboličkih tačaka može se proveriti korišćenjem elementarne geometrije, dok se za rad sa simboličkim tačkama više detalja može videti u knjizi Computational geometry, na strani 204.

Algoritam: legalizeEdge(**p**, $\mathbf{p_i p_j}$, **T**)
Ulaz: tačka **p** koja je poslednja dodata u triangulaciju, ivica $\mathbf{p_i p_j}$ čiju legalnost treba proveriti, trenutna triangulacija **T**
Izlaz: modifikovana triangulacija **T**
1. if ivica $\mathbf{p_i p_j}$ nije legalna
2. Neka je $\mathbf{p_i p_j p_k}$ trougao koji je susedan ivici $\mathbf{p_i p_j}$, a ne sadrži tačku **p**
3. Zameni ivicu $\mathbf{p_i p_j}$ sa ivicom $\mathbf{p p_k}$ u **DCELL**
4. Dodaj u **L** novu decu listovima koji pokazuju na trouglove $\mathbf{p p_i p_j}$ $\mathbf{p_i p_j p_k}$
5. legalizeEdge(**p**, $\mathbf{p_i p_k}$, **T**)
6. legalizeEdge(**p**, $\mathbf{p_k p_j}$, **T**)
7. **return T**

Nakon završene triangulacije, na raspolaganju nam je DCELL struktura D, koja sadrži mrežu kreiranih trouglova. Problem kod korišćenje Delone triangulacije za ostatak ovog algoritma je taj što je Delone triangulacija **potpuno nesvesna ivica poligona**, što dovodi do situacije da pojedini trouglovi seku ivice postojećeg poligona. Modifikacija Delone algoritma tako da se u obzir uzima i sečenje ivica poligona, bilo bi suviše mukotrpno i potencijalno velike računske složenosti. Drugi problem nastaje iz prethodnog, zato što, budući da se ivice poligona ne poštuju, lako je doći do situacije u kojoj ni strogo **tri-bojenje ne mora da postoji**. Naime, postojanje razapinjućeg grafa bez ciklova, kakvo je opisano na početku ovog algoritma, obezbeđeno je tipom triangulacije. Delone triangulacija ne garantuje postojanje tri-bojenja trouglova, čak je lako konstruisati i trivijalan primer u kojem je tri-bojenje nemoguće. Stoga je za potrebe ovog projekta implementirano „best-effort" tri-bojenje, koje se trudi da raspored boja temena bude što raznolikiji moguć.

```
Algoritam: doThreeColoring(T)
Ulaz: završena triangulacija T
Izlaz: modifikovana triangulacija, tako da se su temena triangulacije obojena
1. pronadjimo prvo neobojeno polje F koje nema imaginarnih temena
2. inicijalizujmo stek pretrage S poljem F
3. while S nije prazan do
4.     uzmimo jedno polje sa steka S, neka je to polje cF
5.     obojimo temena polja cF, tako da se nikad ne koristi boja koja je već
korišćena
6.     if polje cF je imalo barem jedno neobojeno polje
7.           dodaj na stek S sve tri konačna suseda polja cF
8. return T
```

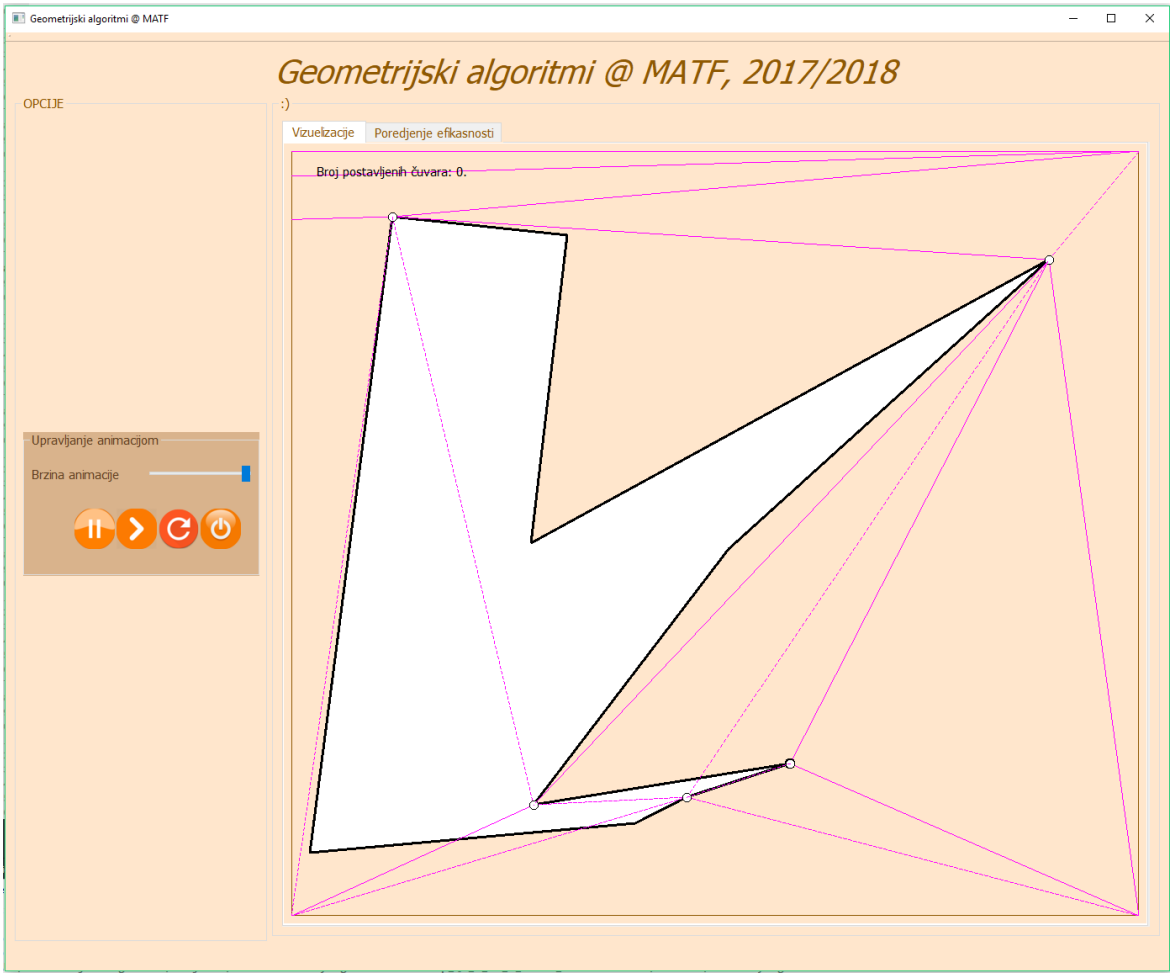
Ostaje još samo da se odaberu čuvari. Nakon što je bojenje (bilo strogo, bilo u varijanti „best-effort") izvršeno, prebrojava se koliko ima temena koje boje. Bira se ona boja koja je najmanje korišćena, i u kranji spisak čuvara se dodaju sva temena koja imaju tu boju.

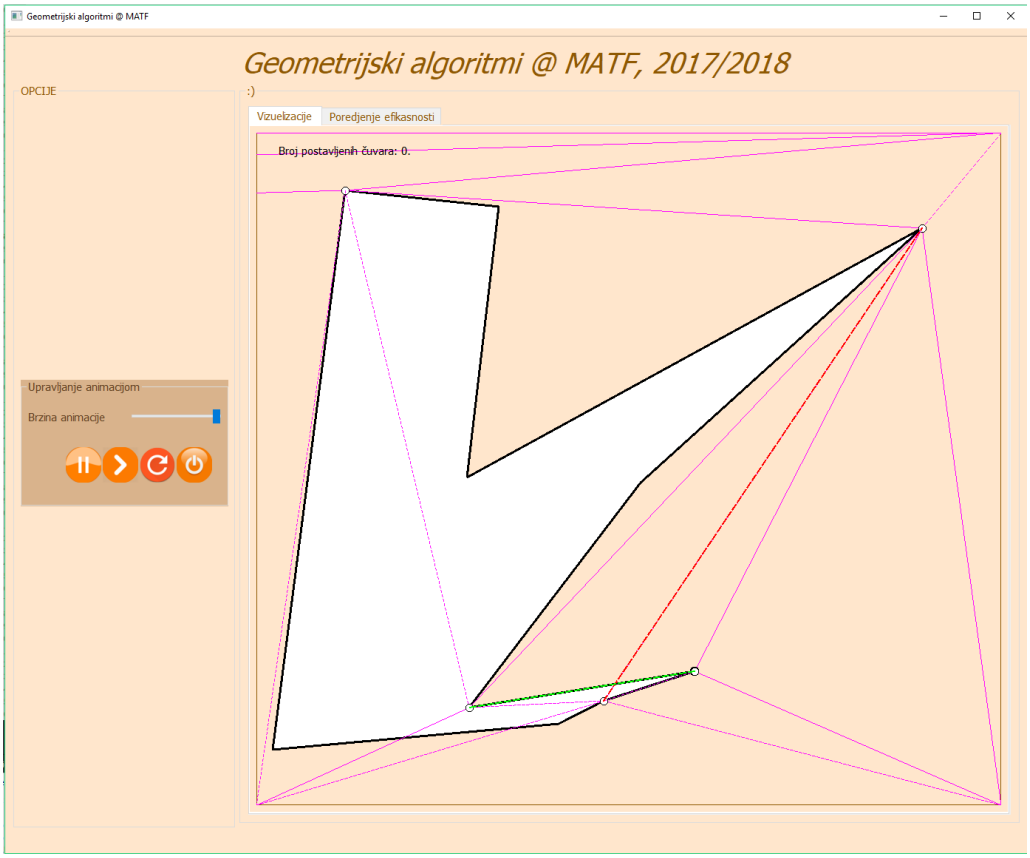
Vizuelizacija algoritma

Iako je rađena vizuelizacija i naivnog i naprednog algoritma (pošto su suštinski različiti, pa je tako nešto imalo smisla), u ovom dokumentu biće detaljnije prikazana vizuelizacija samo glavnog dela algoritma.

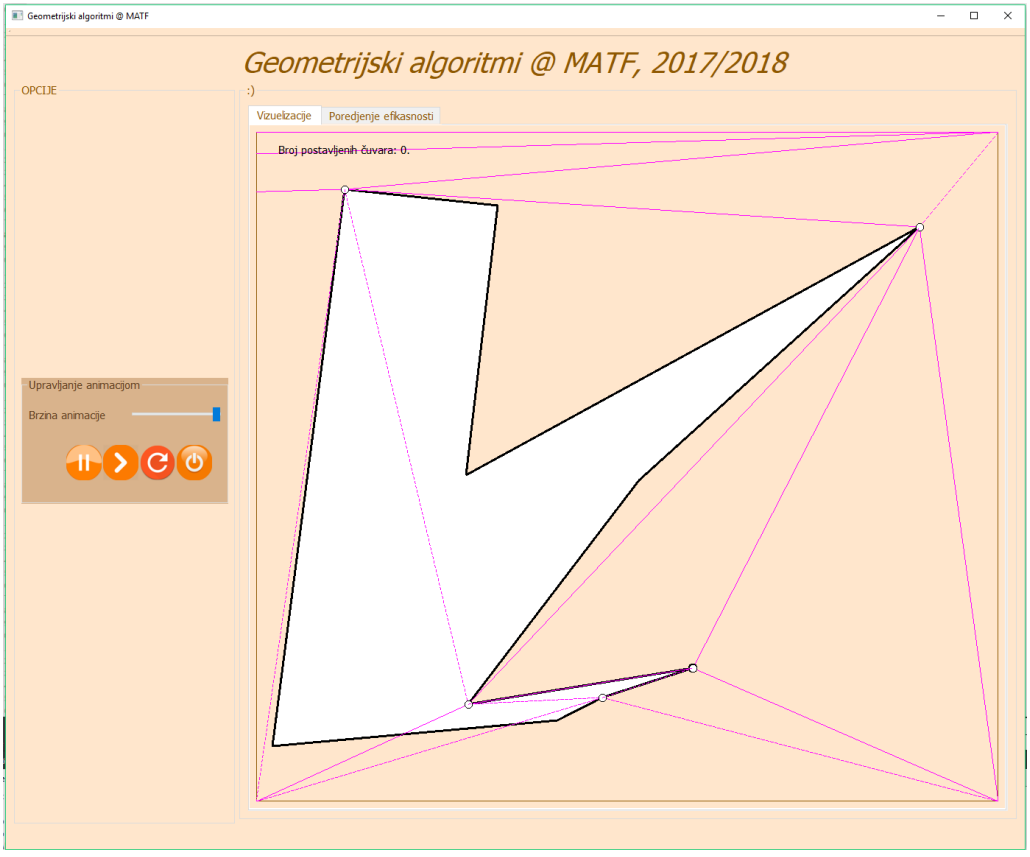
Elementi algoritma koji su vidljivi tokom animacije su sledeći:

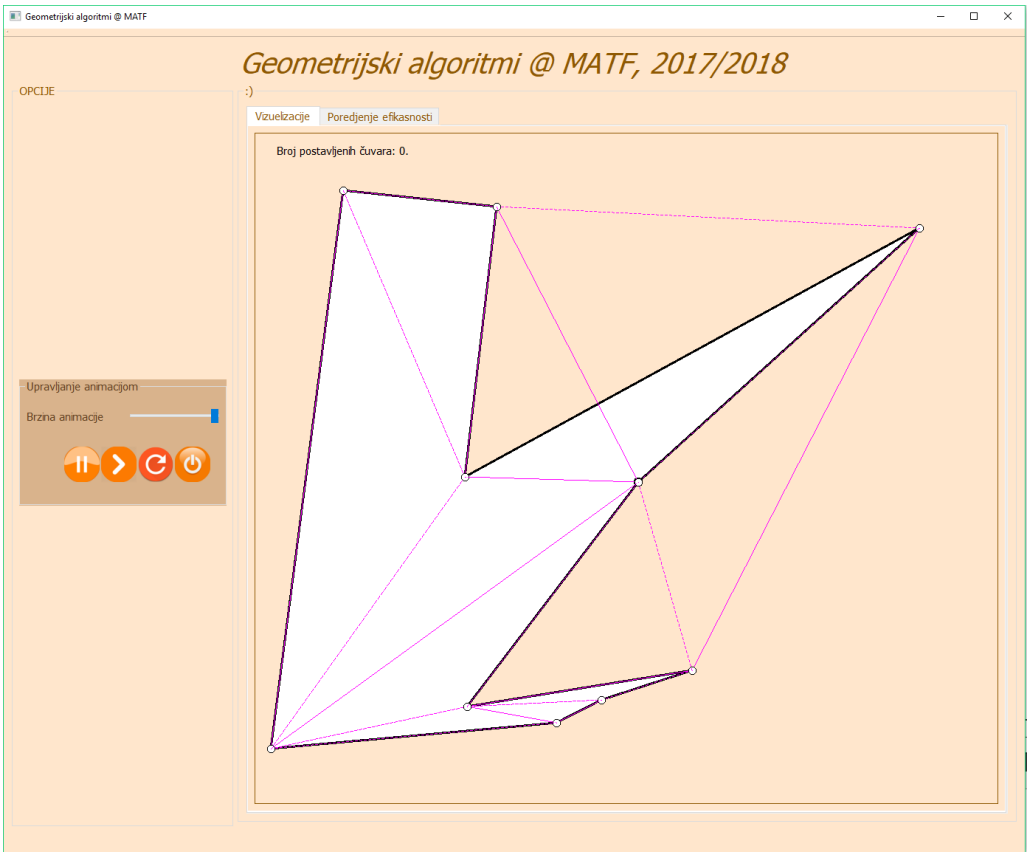
- Pretraga lokacije nove tačke koja se dodaje u triangulaciju
- Dodavanje novih ivica u triangulaciju
- Korak-po-korak legalizacija ivica
- Završetak triangulacije, uklanjanje imaginarnih ivica
- Bojenje jednog po jednog temena
- Selekcija čuvara

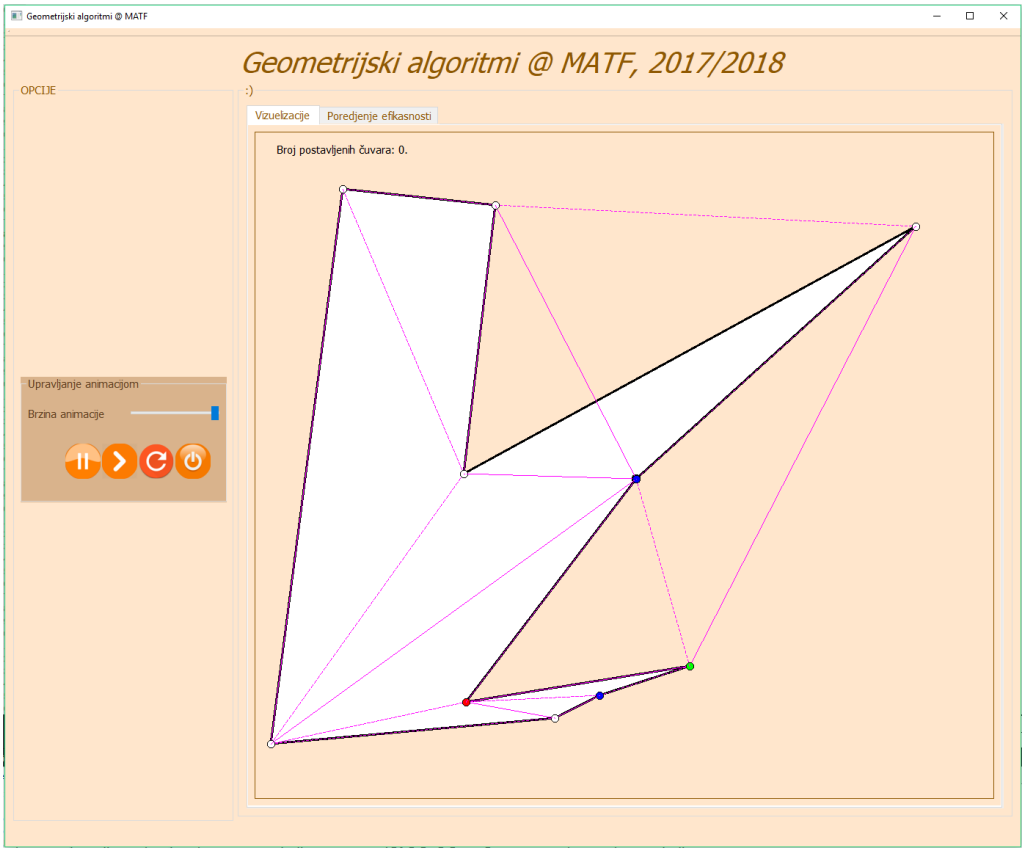


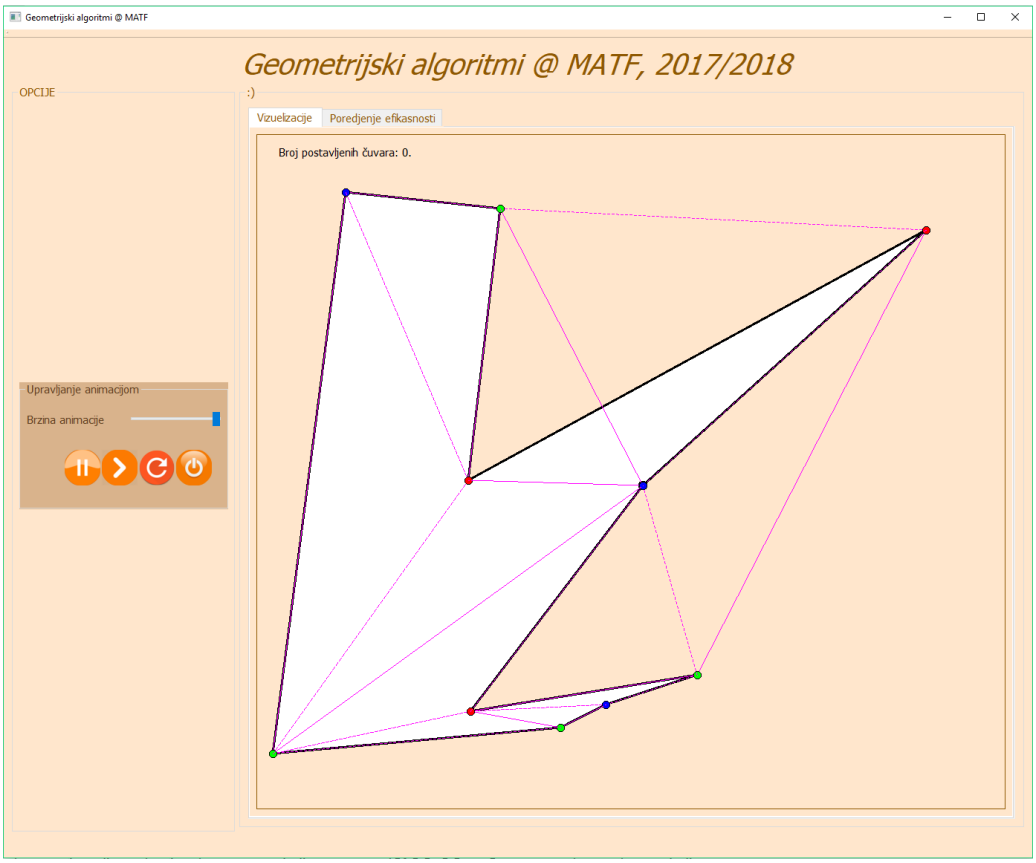

Geometrijski algoritmi @ MATF
Geometrijski algoritmi @ MATF, 2017/2018
OPCIJE
:)
Vizuelizacije
Poredjenje efikasnosti
Broj postavljenih čuvara: 0.
Upravljanje animacijom
Brzina animacije

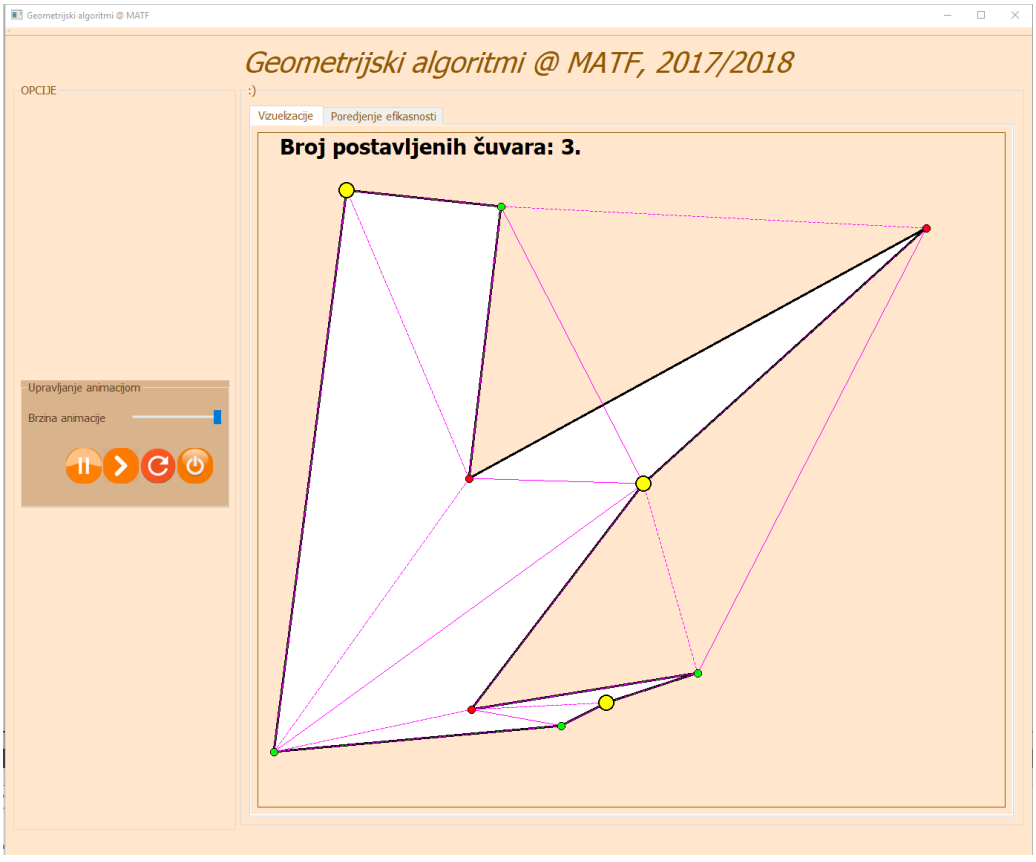
-


Geometrijski algoritmi @ MATF
Geometrijski algoritmi @ MATF, 2017/2018
OPCIJE
:)
Vizuelizacije
Poredjenje efikasnosti
Broj postavljenih čuvara: 0.
Upravljanje animacijom
Brzina animacije


Geometrijski algoritmi @ MATF
Geometrijski algoritmi @ MATF, 2017/2018
OPCIJE
:)
Vizuelzacije
Poredjenje efikasnosti
Broj postavljenih čuvara: 0.
Upravljanje animacijom
Brzina animacije


Geometrijski algoritmi @ MATF
Geometrijski algoritmi @ MATF, 2017/2018
OPCIJE
:)
Vizuelzacije
Poredjenje efikasnosti
Broj postavljenih čuvara: 0.
Upravljanje animacijom
Brzina animacije

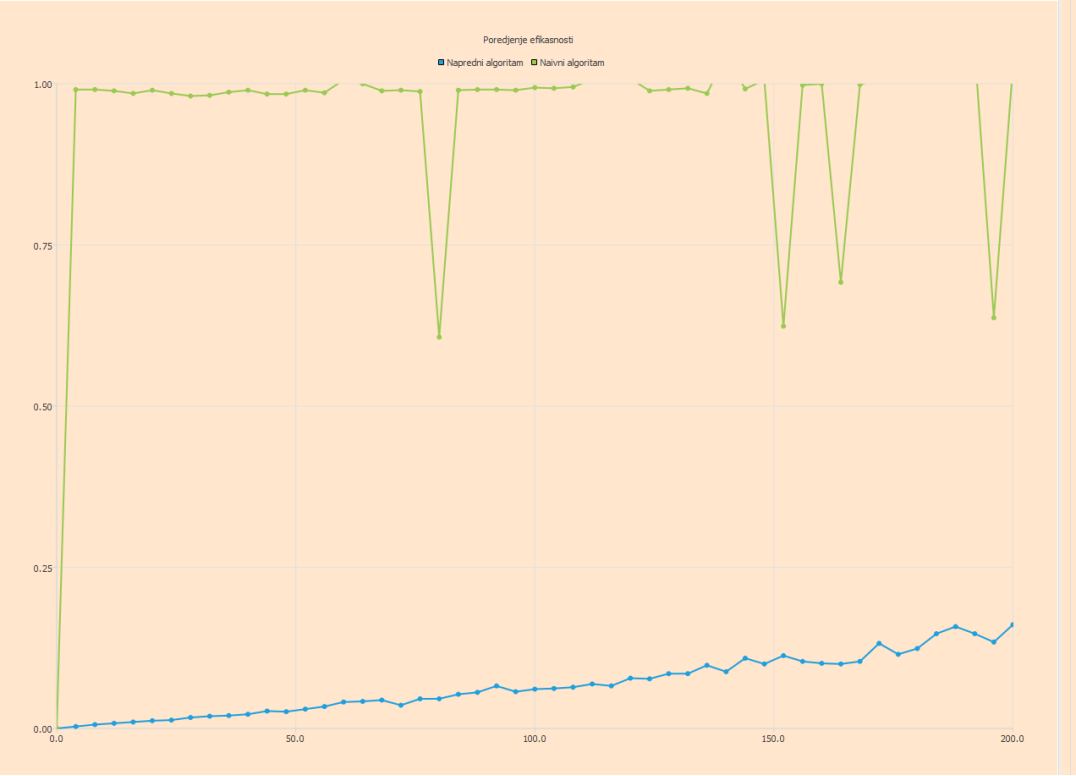

Geometrijski algoritmi @ MATF
Geometrijski algoritmi @ MATF, 2017/2018
OPCIJE
:)
Vizuelzacije
Poredjenje efikasnosti
Broj postavljenih čuvara: 0.
Upravljanje animacijom
Brzina animacije

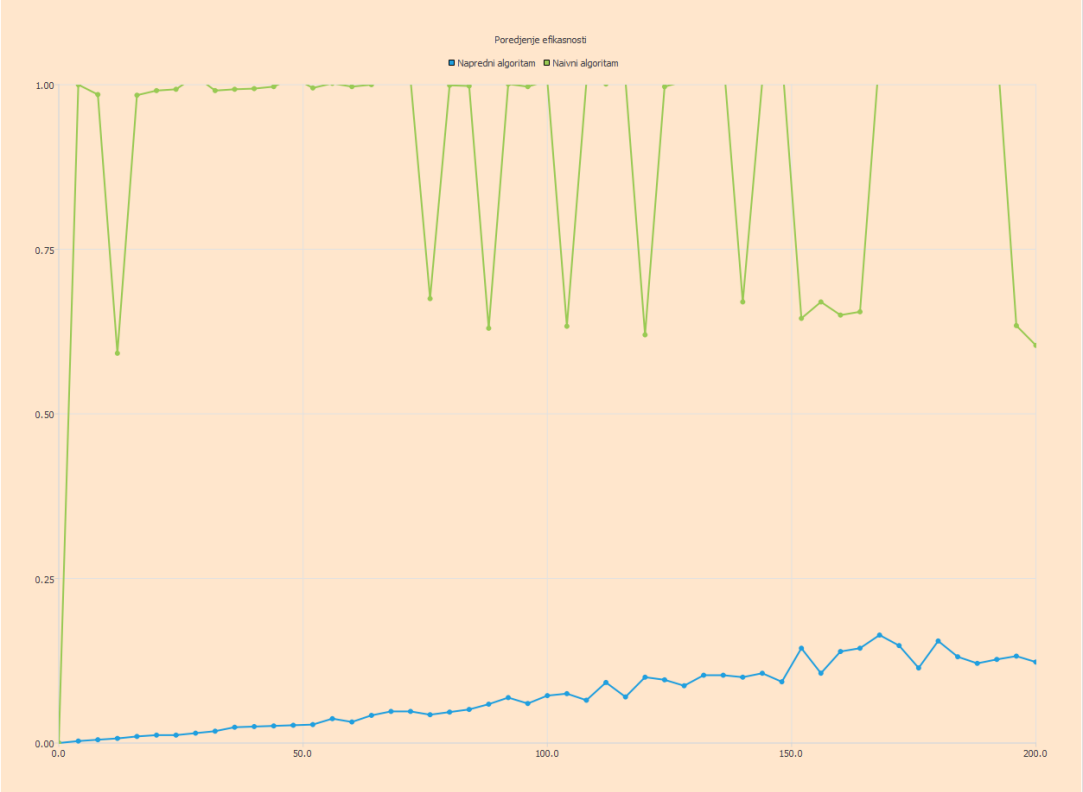

Geometrijski algoritmi @ MATF
Geometrijski algoritmi @ MATF, 2017/2018
OPCIJE
:)
Vizuelzacije
Poredjenje efikasnosti
Broj postavljenih čuvara: 3.
Upravljanje animacijom
Brzina animacije

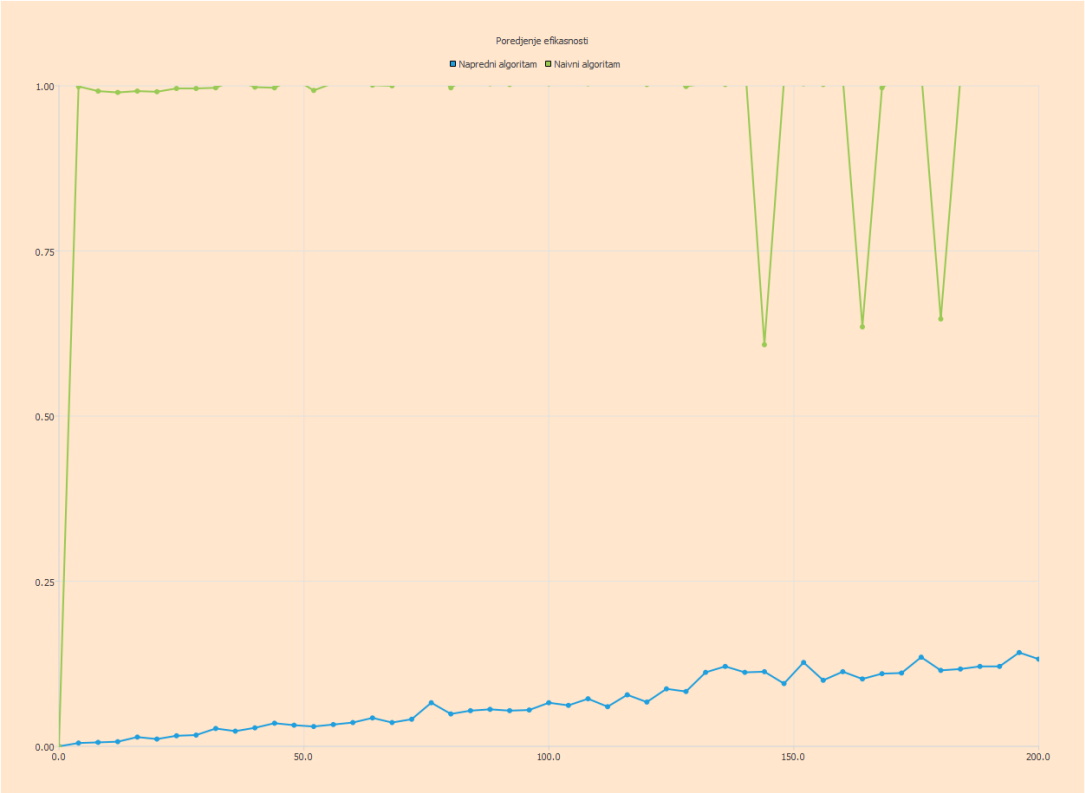
Poređenje efikasnosti naivnog i naprednog algoritma

Poređenje efikasnosti algoritama je u ovom slučaju samo formalna procedura, budući da se radi o dva algoritma potpuno različitih klasa. Naivni algoritam ima slučajno vreme izvršavanja, koje potencijalno može biti beskonačno, te je u konkretnoj implementaciji algoritma uvedeno granično vreme, nakon kojeg algoritam u svakom slučaju prestaje sa radom.

Algoritam je testiran na slučajno kreiranim instancama veličina između 4 i 200 temena.
Ono što je potrebno naglasiti jeste to da je algoritam za kreiranje slučajnog prostog poligona opšteg oblika netrivijalan, pa je korišćena varijanta algoritma koji slučajno odabrane tačke povezuje tako što ih sortira u odnosu na ugao koji grade sa ekstremnom tačkom i osom. Ovakvi poligoni umeju da budu prilično „ružni“, sa uskim zracima koje je teško pogoditi iz naivnog algoritma. Inače, kada bi se radilo o normalnim prostorijama, moguće je da bi slučajan algoritam iz nekoliko iteracija imao mnogo bolje rezultate. Bitno je napomenuti da su testiranja radjena na release verziji koda, pokrenutoj iz Qt Creator studija, Lenovo Thinkpad T460s laptopu (16GB RAM, 4 logička jezgra na 3.3GhZ sa 4MB L3 keša).




Poredjenje efikasnosti
Napredni algoritam
Naivni algoritam
1.00
0.75
0.50
0.25
0.00
0.0
50.0
100.0
150.0
200.0


Poredjenje efikasnosti
Napredni algoritam
Naivni algoritam
1.00
0.75
0.50
0.25
0.00
0.0
50.0
100.0
150.0
200.0

Testiranje ispravnosti algoritama

Za potrebe testiranja algoritma napravljena su tri skupa testova:

- Pravi unit testovi, koji su pokrivali funkcionalnost nekih pomoćnih metoda
- Test naivnog algoritma
- Testovi glavnog algoritma

Pravi unit testovi su korišćeni da bi se verifikovala tačnost procedura čija implementacija je osetljiva na greške. Takođe, dodavani su i „regression" testovi, koji osiguravaju da se pronađena i popravljena greška ne mogu javiti ponovo. Neki od testova koji su ostavljeni u krajnjem izboru, prikazani su u tabeli ispod.

ga04_SentryPlacementTests:

Naziv testa	Opis testa
generateRandomSimplePoly_invalidInputTests	Verifikuje se da će funkcija baciti grešku ako je broj zahtevanih temena manji od 3.
generateRandomSimplePoly_positiveTests	Verifikuje se da za dati broj tačaka (5, 10, 50, 100) funkcija proizvodi prost zatvoren poligon.
fastPolyContains_simpleSquareTests	Definisan je mali kvadrat i dva seta tačaka za koje se pretpostavlja da su unutar odnosno van kvadrata u fajlovima squareRoom.def, squareRoom.in, squareRoom.out, respektivno. Funkciji se prosleđuju redom tačke iz skupova in i out i verifikuju rezultati.
pointOutsideOfCircumcircleTests	Verifikuje se da funkcija koja je potrebna u legalizaciji ivica radi korektno.
isBetweenTests	Verifikuje se validnost metoda za proveru raporeda tačaka na duži.
CmpQPointOperatorTest	Verifikuje se validnost operatora poređenja koordinata tačaka.

containsPointTest	Verifikuje se validnost zajedničkog metoda za ispitivanje pripadnosti tačke trouglu.
triangleEdgeContainsPointTest	Verifikuje se da prethodni metod vraća valjan rezultat i kada se ispituje pripadnost tačke koja se nalazi tačno na ivici trougla.
lexGreaterTest	Verifikuje ispravnost funkcije za leksikografsko poređenje tačaka.
onTheLeftSideOfAB	Verifikuje osnovnu funkcionalnu ispravnost metoda za ispitivanje orijentacije trojke tačaka u prostoru.
triangleContainsPointRegressionTest	Malo komplikovaniji test, koji prvo inicijalizuje delimičnu DCELL strukturu, a potom proverava da li trougao date triangulacije sadrži datu tačku.

Za testiranje naivnog algoritma, samo je jedan test kreiran, i to sa trivijalnim brojem temena, koji prosto verifikuje funkcionalnu ispravnost algoritma. Neko opsežnije testiranje naivnog algoritma, s obzirom na slučajnost kvaliteta dobijenih rezultata nije realno.

ga04_SentryPlacementTests_endToEnd_naive:

Naziv testa	Opis testa
flagRoom	Verifikuje se da naivni algoritam radi na nekonveksnoj sobi sa 5 ćoškova.

Poslednji skup testova odnosi se na testiranje rada glavnog algoritma. Imajući u vidu faktor slučajnosti koji igra ulogu u Delone triangulaciji, verifikacija krajnjeg rezultat, osim do domena funkcionalnog testiranja, nema preteranog smisla. Ono što bi bilo moguće napraviti, jesu testovi koji bi vršili punu verifikaciju, da je svako teme vidljivo imajući u vidu odabir čuvara, ali to ne bi bilo ispravno, budući da algoritam po trenutnoj konstrukciji i ne daje uvek tačno rešenje.

ga04_SentryPlacementTests_endToEnd_main

Naziv testa	Opis testa
flagRoom	Verifikuje se da glavni algoritam radi na nekonveksnoj sobi sa 5 ćoškova.
mediumRoom	Verifikuje da glavni algoritam radi korektno na sobi sa 17 ćoškova, uz mogućnost pogađanja podslučaja, da se tačka koja se dodaje u triangulaciju nalazi tačno na već postojećoj ivici.
threeGuardsRoom	Verifikuje da glavni algoritam radi na sobi sa 9 ćoškova, u kojoj se očekuje da broj čuvara bude približno jednak 3.
artGallery	Verifikuje da glavni algoritam radi na sobi koja podseća na umetničku galeriju, sa 62 temena (slika na početku dokumenta).

Zaključak

Predloženi napredni algoritam ima značajnu grešku u samom startu, što ga onemogućava da bude smatran validnim rešenjem problema. Uopšte uvođenje same Delone triangulacije ima smisla, ako bismo zapravo razmatrali raspored ljudi-čuvara. Delone triangulacija obezbeđuje da „reoni" za koje su zaduženi čuvari budu što prirodnijeg oblika, time čineći posao ljudskih čuvara značajno lakšim. Sa druge strane, kompleksnost rešenja koje bi uzimalo najbolje od dva sveta (Delone triangulaciju i pouzdanost dobijanja rešenja za čuvare koje nudi klasična tri angulacija) bi za veće i kompleksnije prostore bila preterano velika.

Implementacija Delone triangulacije, kao i njena vizuelizacija, su same po sebi predstavljale dovoljan izazov, s obzirom na kompleksnost debagovanja i praćenja izvršavanja algoritma koji operiše sa kompleksnim grafovskim strukturama. Najbolji način praćenja izvršavanja, osim gledanja same vizuelizacije, bilo je ispisivanje qDebug poruka, koje je moguće (nakon modifikacije

u .pro fajlovima) isključiti u „Release“ verziji programa. Isključivanje debug poruka je neophodno, radi dobijanja validnih rezultata pri merenju performansi koda.

U trenutno napisanom kodu ima dosta mesta za poboljšanje, kako i performansi (postoji nepotrebno kopiranje podataka na nekoliko mesta, ne koriste se move reference), tako i čitljivosti (moguće je koristiti neku od smart-pointer klasa, umesto rada sa fizičkim pokazivačima i kompleksnim procedurama za potonje oslobađanje memorije).