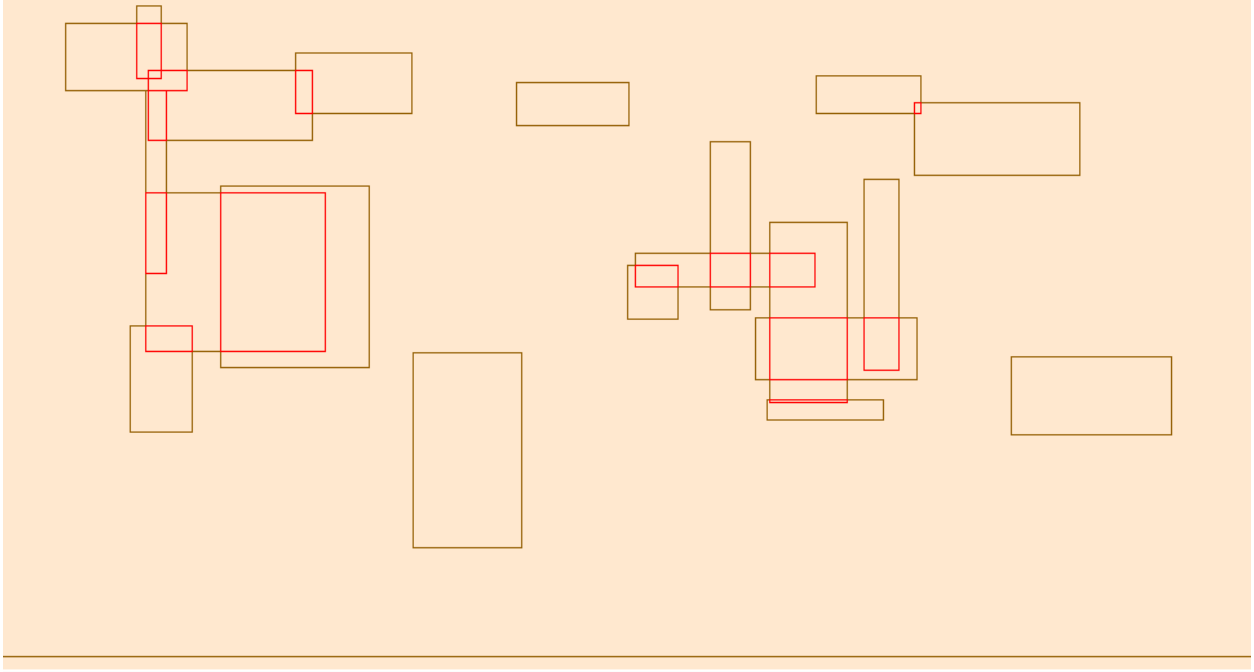
Geometrijski algoritmi @ MATF

Algoritam za određivanje preseka *pravougaonika*

Ana Stanković



Opis problema

Za dati skup pravougaonika sa stranicama paralelnim osama, potrebno je odredite sve preseke pravougaonika I obeležiti ih.

Ulaz: *skup od n pravougaonika*

Izlaz: *skup pravougaonika koje predstavljaju preseke pravougaonika*

Naivno rešenje problema

Naivni algoritam je odrađen metodom grube sile gde se za svaki pravougaonik proverava da li ima presek sa svim ostalim pravougaonicima. Veoma neefikasan način zbog same složenosti $O(n^2)$.

U nastavku je dat segment koda. Proveravanje za svaka dva pravougaonika je odrađeno tako što su slati kao argumenti u funkciji findIntersectionRec gde se proveravao presek dva pravougaonika.

```
for(int i = 0; i < _rectangles.size(); i++)
{
    for(int j = i+1; j < _rectangles.size(); j++)
    {
        if(_rectangles[i].topLeft().x() > _rectangles[j].bottomRight().x()
                || (_rectangles[i].bottomRight().x() < _rectangles[j].topLeft().x())
                || (_rectangles[i].topLeft().y() > _rectangles[j].bottomRight().y())
                || (_rectangles[i].bottomRight().y() < _rectangles[j].topLeft().y()))
        {

        }
        else
        {
            QRect newRect;
            findIntersectionRec(_rectangles[i],_rectangles[j],&newRect);
            _naiveIntersectionRectangles2.push_back(newRect);

        }
    }
}
```

Napredni algoritam – zasnovan na metodi brišuće prave i korišćenja stabla pretrage duži

Neefikasnost naivnog algoritma predstavlja proveravanje svakog pravougaonika sa svakim. U naprednom algoritmu koristimo brišuću pravu koja se kreće odozgo nadole i kada naiđe na gornju duž pravougaonika, stavlja ga u status i proverava presek tog pravougaonika sa pravougaonicima iz statusa. S obzirom da je status implementiran kao stablo pretrage duži, neće se proveravati svi pravougaonici iz statusa već se proverava za trenutnu duž da li seče duž koja je u korenu stabla pretrage, a zatim pretražuje levo podstablo, ukoliko je maksimum u levom podstablu veći od niže vrednosti trenutne duži, inače se ide na desno podstablo. Osnovna ideja naprednog algoritma je da se proveravaju samo neki pravougaonici koji su u statusu i koji predstavljaju potencijalne kandidate za presek, a ne sve pravougaonike iz statusa. Jednom kad brišuća prava naiđe na donju duž pravougaonika, on se izbacuje iz statusa i ne predstavlja više potencijalnog kandidata za presek sa ostalim pravougaonicima koje treba proveriti.

Definisane su tačke događaja - gornja i donja prava pravougaonika u klasi IntervalRect koje se čuvaju u intervalnom stablu pretrage (u kodu označenom std::set<IntervalRect,EventNodeComp> eventQueue). Status je označen kao ITree statusQueue.

ITree predstavlja stablo pretrage duži čiji su čvorovi intervalne vrednosti, tj. gornje duži pravougaonika na koje je naišla brišuća prava, označene sa ITNode. U slučaju dodavanja novih čvorova tj, kad naiđemo na novi pravougaonik, poziva se funkcija koja je definisana u stablu insert koja dodaje novi čvor u stablo, ali ga ujedno balansira tako da nakon dodavanja čvora, stablo ostaje izbalansirano. Kada brišuća prava naiđe na donju duž pravougaonika, poziva se funkcija deleteIntervalRect koja briše taj pravougaonik I balansira stablo da bi I nakon brisanja ostalo balansirano. Funkcija overlapSearch proverava za trenutni

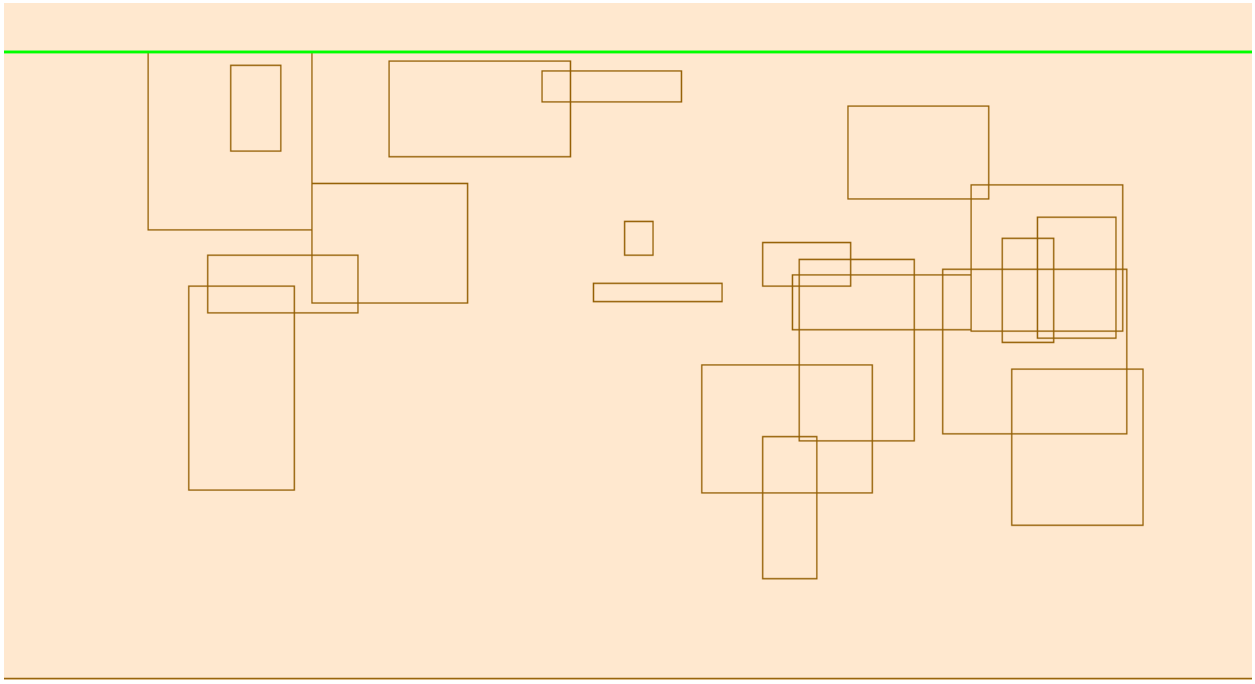
pravougaonik preseke sa pravougaonicima koji su u stablu, ali ne proverava za sve čvorove iz stabla. Za one čvorove sa kojima ne može da ima preseke, ne proverava.

Ukoliko se trenutni pravouganik na koji je naišla prava seče sa nekim iz statusa, stavljamo njihov presek u std::vector<QRect> advancedIntersectionRectangle2.

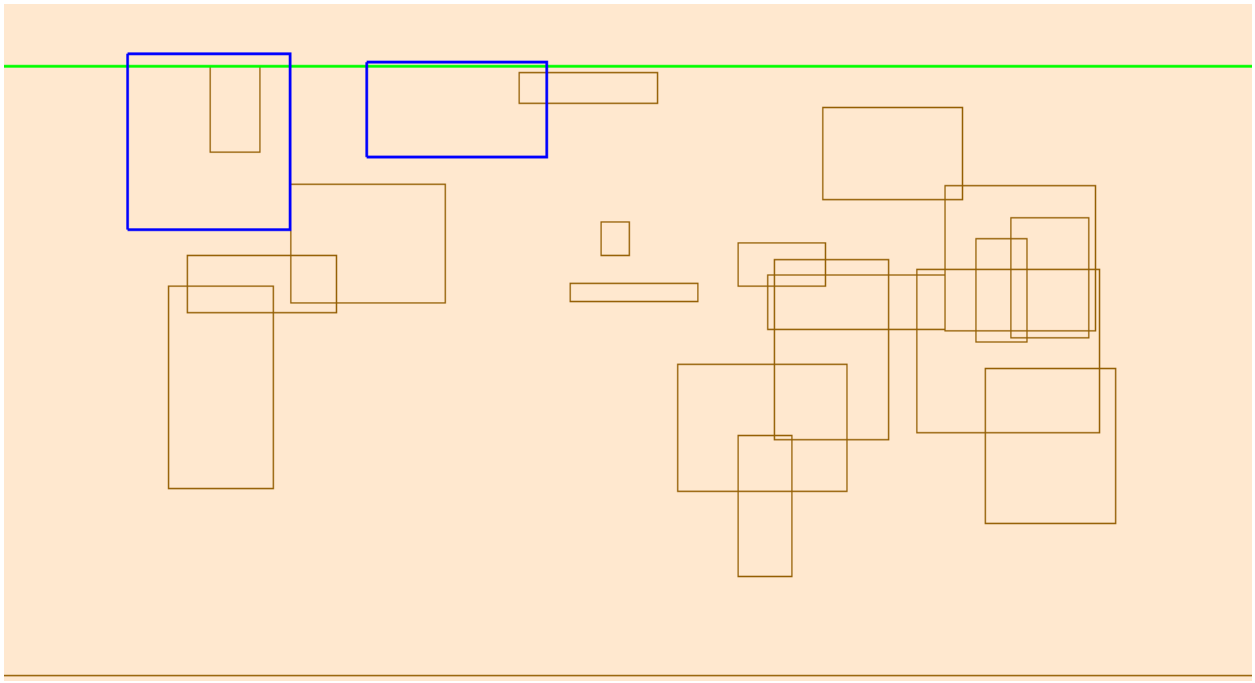
```
// Struktura koja predstavlja duz koja se stavlja u status i proverava
struct IntervalRect
{
public:
    int low, high; // manja i veca x koordinata
    int height; // y koordinata
    TypeOfEvent type;
    QRect rect; // pamti se i ceo pravougaonik jedino zbog crtanja preseka
    IntervalRect(int l, int h, int he,QRect r, TypeOfEvent t1)
        : low(l), high(h), height(he),rect(r), type(t1)
    {
    }

};

// cvor interval search tree
struct ITNode
{
public:
    IntervalRect i;
    int max, balance;
    ITNode *left,*right, *parent;

    ITNode(IntervalRect i1, ITNode *p): i(i1),max(i1.high), balance(0), left(NULL), right(NULL), parent(p)
    {
    }

    ~ITNode() {
        delete left;
        delete right;
    }
```

```
void IntersectionRectangle::runAlgorithm()
{
    _statusQueue.inorder(_statusQueue.root);

    while(!_eventQueue.empty())
    {

        IntervalRect eventNode = *_eventQueue.begin();
        _eventQueue.erase(_eventQueue.begin());

        if(eventNode.type == UPPER_P)
        {
            //pomeranje brisuce prave na bas to teme koje se obradjuje
            _sweepLine = eventNode.height;
            AlgorithmBase_updateCanvasAndBlock();
            _statusQueue.overlapSearch(_statusQueue.root,eventNode);
            _statusQueue.insert(eventNode);
        }
       //ukoliko je donje teme, samo se izbrise pravougaonik iz statusa
       //i on se  vise ne proverava sa drugim pravougaonicima
        else if (eventNode.type == LOWER_P)
        {
            _sweepLine = eventNode.height;
            AlgorithmBase_updateCanvasAndBlock();
           _statusQueue.deleteIntervalRect(eventNode);

        }
    }

    _sweepLine = 0;
    AlgorithmBase_updateCanvasAndBlock();
    emit animationFinished();
```

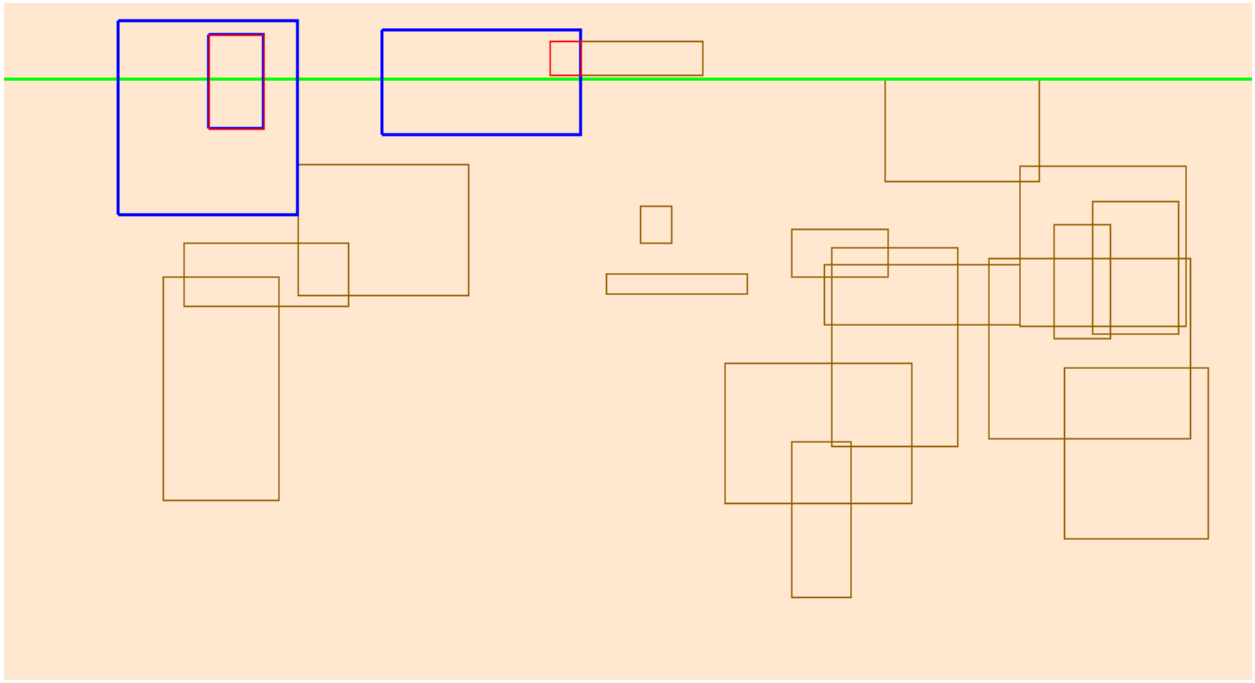
Vizuelizacija algoritma



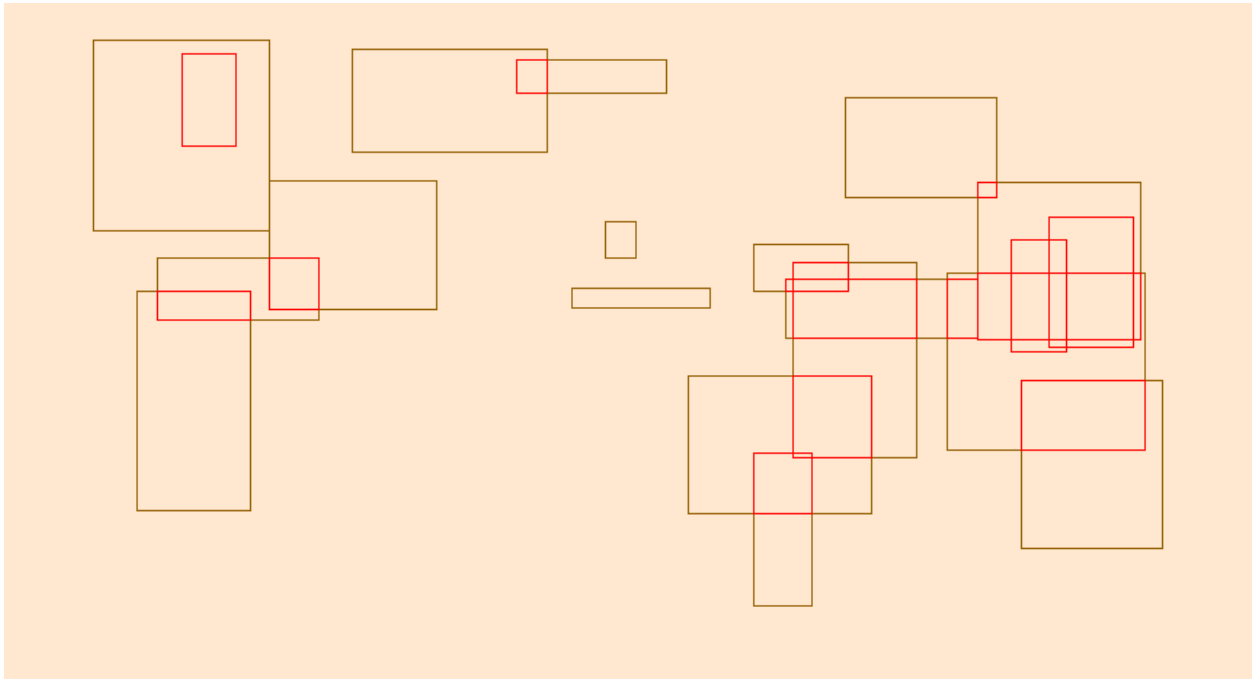
Prva slika pokazuje početno stanje, kada nije ispitan nijedan pravougaonik.



Druga slika pokazuje stanje kada je brišuća naišla na drugi pravougaonik. Jedan je u statusu već, drugi je stavljen i za njega se proverava da li ima preseka sa prvim pravougaonikom.

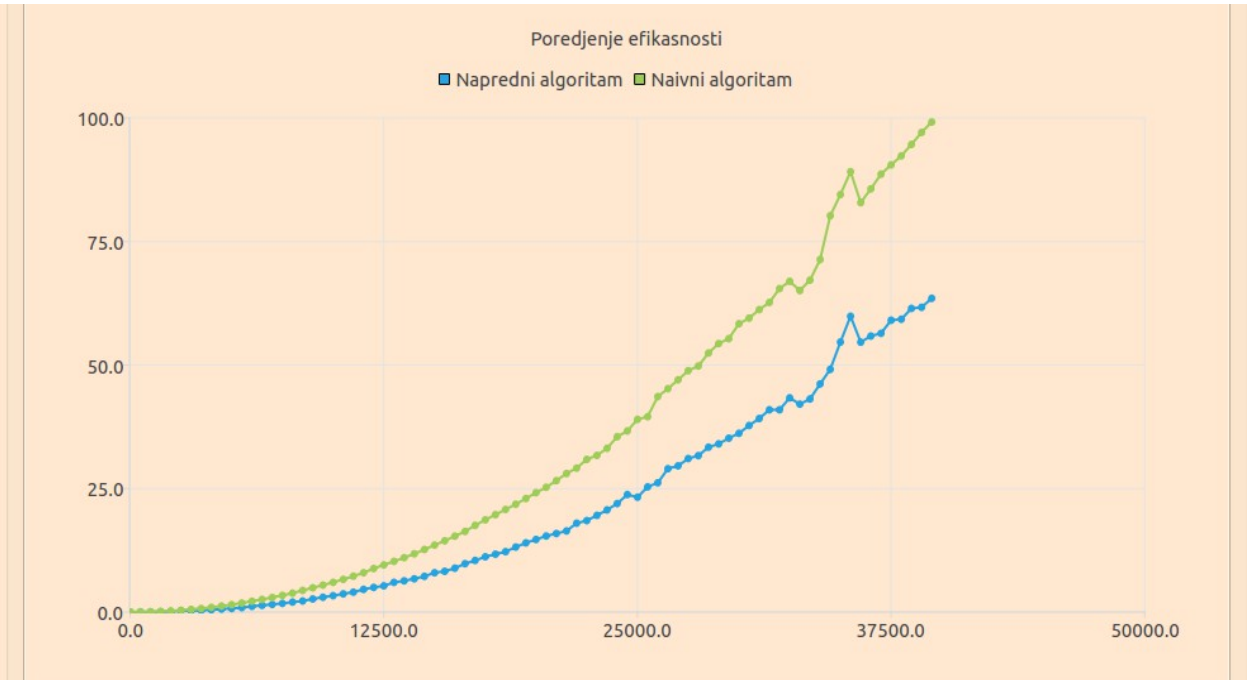


Treća slika pokazuje koji su pravougaonici u statusu (označeni plavom bojom) i nađene preseke (crvenom bojom).



Četvrta slika pokazuje nađene preseke nakon završetka algoritma.

Poredjenje efikasnosti naivnog i naprednog algoritma



alg. / dim. ulaza	10	100	500	10000	50000
naivni	2.512e-05	0.001136	0.008952	2.005572	19.0098
optimalni	5.362e-05	0.000911	0.006284	1.06656	11.8246

Vidimo iz tabele da sa povećanjem broja pravougaonika, brzina izvrsavanja naprednog algoritam ce biti dosta manja od naivnog algoritma.

Testiranje ispravnosti algoritma

Naziv testa	Opis testa	Ulaz	Očekivani izlaz
withoutRectangle	Zadavanje nula pravougaonika	[]	Nema preseka
oneRectangle	Zadavanje jednog pravougaonika	Niz dimenzije 1	Nema preseka

tenRandomRectangle	Zadavanje deset slučajnih pravougaonika	Niz dimenzije 10	Rezultati naprednog i naivnog algoritma su se poklopili
sortedX	Zadavanje sortiranih pravougaonika po X	Niz dimenzije 7	Rezultati naprednog i naivnog algoritma su se poklopili
sortedY	Zadavanje sortiranih pravougaonika po Y	Niz dimenzije 7	Rezultati naprednog i naivnog algoritma su se poklopili
50RandomRectangle	Zadavanje pedet slučajnih pravougaonika	Niz dimenzije 50	Rezultati naprednog i naivnog algoritma su se poklopili