Primena fazi logike u obradi slika

Jelena Mrdak, mi15021
Tijana Jevtić

5. januar 2019

Sažetak

U ovom radu je predstavljena primena fazi logike u obradi slika. Opisana su dve algoritma, Fuzzy C-means (FCM) za binarizaciju slike i Fuzzy Edge Detection (FED) za detekciju ivica. Takođe, ovi algoritmi su poređeni sa algoritmima koji ne koriste fazi logiku. Konkretno, poredili smo FCM sa k-means-om i FED sa Canny Edge Detection (CED) algoritmom. Slike dobijene FCM-om i k-means-om se skoro i ne razlikuju, dok to nije slučaj sa FED-om i CED-om, čiji su izlazi primetno drugačiji. U oba slučaja su algoritmi koji koriste fazi logiku bila brža.

Sadržaj

1 Uvod

U matematici smo do sad navikli da je nešto tačno ili netačno, da nešto pripada ili ne pripada skupu. Međutim, nekad je teško povući granicu i odrediti kad je nešto 0, a kad 1. Što smo bliži granici, to nesigurnost veća. Na primer, ako želimo da klasifikujemo ljude na niske i visoke i postavimo granicu na $170cm$, dobijamo da je osoba visoka $170cm$ visoka, dok je osoba visoka $169cm$ niska, što i nema mnogo smisla.

Fazi logiku je 1965. godine uveo Lotfi Zadeh i ona nam u velikoj meri može pomoći za rešavanje pomenutih problema. Fazi skupovi se razlikuju od klasičnih skupova kod kojih

je granica jasna (element ili prripada ili ne pripada skupu). Zadeh je uopštio klasične skupove tako što je proširio skup valuacije $\{0, 1\}$ na interval realnih brojeva $[0, 1]$. Stepen pripadnosti nekog elementa fazi skupu opisuje koliko taj element odgovara pojmu koji je reprezentovan fazi skupom. Konkretno, element x pripada skupu A sa stepenom $\mu_A(x)$, gde je $\mu_A : A \to [0, 1]$ karakteristična funkcija skupa A.

U ovom radu smo koristili upravo ove ideje kako bismo odredili da li je piksel crn ili beo, ili da li je piksel ivica ili nije.

Kodovi su pisani u jeziku *C++* pomoću biblioteke *OpenCV*. Napominjemo da su svi kodovi, kako oni navedeni u radu, tako i oni koji nisu, ali koji su korišćeni za poređenje, pisani ručno od strane autora, te da svi rezultati i zaključci zavise od njihove implementacije. Takođe prikazana vremena izvršavanja u velikoj meri zavise od mašine na kojoj se kodovi izvršavaju, te mogu varirati, ali su sva vremena dobijena testiranjem na istom računaru u sličniim uslovima.

2 FCM

Fuzzy C-means (FCM) je jedan od najpopularnijih algoritama za fazi klasterovanje. U ovom poglavlju ćemo ga najpre detaljno opisati, a zatim ćemo ga iskoristiti za binarizaciju slike.

Cilj ovog algoritma je da skup $X = \{x_1, x_2, ..., x_n\}$ particioniše na k delova (klastera) po nekom kriterijumu. Preciznije, kriterijum je minimizacija sledeće funkcije:

$$F(\bar{w}, \bar{c}) = \sum_{i=1}^{n} \sum_{j=1}^{k} w_{ij}^{m} \left\| x_i - c_j \right\|^2,$$

gde $w_{ij} \in [0, 1]$ predstavlja pripadnost tačke x_i j-tom klasteru i $\sum_{j=1}^{k} w_{ij} = 1$, dok je c_j centroid j-tog klastera. Realni parametar $m > 1$ predstavlja faktor fazifikacije i on se zadaje unapred. U nastavku ćemo preciznije odrediti ove koeficijente. Sada ćemo samo ukratko opisati korake algoritma.

FCM je veoma sličan algoritmu k-means i sastoji se iz sledećih koraka:

- Ako je slika u boji, konvertovati je u sivu.
- Izabrati broj klastera k.
- Svakoj tački x_i dodeliti koeficijente $w_{ij} \in [0, 1], j = 1, 2, ..., k$.
- Ponavljati sve dok ne dođe do konvergencije:
  - Izračunati centroide za svaki klaster.
  - Ažurirati koeficijente.

- Tačku x_i dodeliti klasteru kom najviše pripada, tj. r-tom klasteru, gde je $w_{ir} = \max_j w_{ij}$.

Teorema 2.1. *Potrebni uslovi za minimizator* $(\bar{w}^*, \bar{c}^*)$ *funkcije* $F(\bar{w}, \bar{c})$ *su:*

$$c_j = \frac{\sum_{i=1}^{n} w_{ij}^m \cdot x_i}{\sum_{i=1}^{n} w_{ij}^m} \tag{1}$$

i

$$w_{ij} = \frac{1}{\sum_{u=1}^{k} \left(\frac{\|x_i - c_j\|}{\|x_i - c_u\|} \right)^{\frac{2}{m-1}}} \tag{2}$$

Dokaz. Pronaći ćemo potencijalne tačke lokalnih uslovnih ekstremuma. Koristićemo Lagranževe množioce. Posmatraćemo pomoćnu funkciju:

$$J(\bar{w}, \bar{c}, \bar{\lambda}) = \sum_{i=1}^{n} \sum_{j=1}^{k} w_{ij}^m \|x_i - c_j\|^2 - \sum_{i=1}^{n} \lambda_i \left(\sum_{j=1}^{k} w_{ij} - 1 \right).$$

Tačke koje tražimo moraju da zadovoljavaju uslov $\nabla J = \mathbf{0}$. Dakle,

$$\frac{\partial J}{\partial c_j} = 0, 1 \leq j \leq k \tag{3}$$

$$\frac{\partial J}{\partial w_{ij}} = 0, 1 \leq i \leq n, 1 \leq j \leq k \tag{4}$$

$$\frac{\partial J}{\partial \lambda_i} = 0, 1 \leq i \leq n \tag{5}$$

Rešavanjem (3) dobijamo (1). Iz (4) imamo

$$m w_{ij}^{m-1} \|x_i - c_j\|^2 - \lambda_i = 0,$$

odnosno

$$w_{ij} = \left(\frac{\lambda_i}{m \|x_i - c_j\|^2} \right)^{\frac{1}{m-1}}. \tag{6}$$

Iz (5) dobijamo:

$$\begin{aligned}
1 &= \sum_{u=1}^{k} w_{iu} \\
&= \sum_{u=1}^{k} \left(\frac{\lambda_i}{m \left\| x_i - c_u \right\|^2} \right)^{\frac{1}{m-1}} \\
&= \sum_{u=1}^{k} \left(\frac{m \left\| x_i - c_u \right\|^2}{\lambda_i} \right)^{\frac{1}{1-m}} \\
&= \sum_{u=1}^{k} \frac{(m \left\| x_i - c_u \right\|^2)^{\frac{1}{1-m}}}{\lambda_i^{\frac{1}{1-m}}} \\
&= \frac{1}{\lambda_i^{\frac{1}{1-m}}} \sum_{u=1}^{k} (m \left\| x_i - c_u \right\|^2)^{\frac{1}{1-m}},
\end{aligned}$$

pa zaključujemo da je

$$\lambda_i = \left(\sum_{u=1}^{k} (m \left\| x_i - c_u \right\|^2)^{\frac{1}{1-m}} \right)^{1-m}.$$

Konačno, zamenjujući poslednju jednakost u (6), dobijamo (2). □

FCM algoritam za određivanje minimizatora funkcije F je iteracija kroz potrebne uslove.

2.1 Binarizacija slike

Ispod je prikazan kod za binarizaciju slike koji koristi FCM algoritam. Napominjemo da se zbog čitljivosti koda u ovom delu nismo odlučili za efikasnu implementaciju. O tome će biti više reči u narednoj sekciji.

```
#include <iostream>
#include <opencv2/highgui/highgui.hpp>

int main(int argc, const char *argv[])
{
  if (argc != 2) {
    std::cerr << "Usage: ./binarization path_to_img" << std::endl;
    return 1;
  }

  // read image
  cv::Mat img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);
  cv::Mat img_binary = cv::Mat(img.rows, img.cols, CV_8UC1,
  ↪  cv::Scalar(255));
```

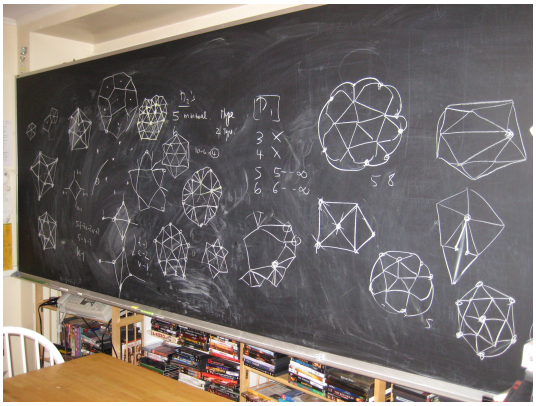
```
// weights
std::vector<std::vector<float>> w1(img.rows,
↪  std::vector<float>(img.cols, 0));
std::vector<std::vector<float>> w2(img.rows,
↪  std::vector<float>(img.cols, 0));
// fuzzification factor
double m = 2;
// centroids
std::pair<float, float> c;

// init weights
for (int i = 0; i < img.rows; i++) {
  for (int j = 0; j < img.cols; j++) {
    w1[i][j] = img.at<unsigned char>(i,j)/255.0;
    w2[i][j] = 1 - w1[i][j];
  }
}

// stopping criteria
float eps = 0;

do {
  // calculate centroids
  std::pair<float, float> c1_fraction{0,0};
  std::pair<float, float> c2_fraction{0,0};

  for (int i = 0; i < img.rows; i++) {
    for (int j = 0; j < img.cols; j++) {
      c1_fraction.first += std::pow(w1[i][j], m)*img.at<unsigned
      ↪  char>(i,j);
      c1_fraction.second += std::pow(w1[i][j], m);
      c2_fraction.first += std::pow(w2[i][j], m)*img.at<unsigned
      ↪  char>(i,j);
      c2_fraction.second += std::pow(w2[i][j], m);
    }
  }

  auto old_c = c;
  c = {c1_fraction.first/c1_fraction.second,
  ↪  c2_fraction.first/c2_fraction.second};
  eps = (old_c.first-c.first)*(old_c.first-c.first) +
  ↪  (old_c.second-c.second)*(old_c.second-c.second);

  // update weights
  for (int i = 0; i < img.rows; i++) {
```

```
      for (int j = 0; j < img.cols; j++) {
        float d1 = std::abs(img.at<unsigned char>(i,j)-c.first);
        float d2 = std::abs(img.at<unsigned char>(i,j)-c.second);
        w1[i][j] = 1/(std::pow(d1/d1, 2/(m-1)) + std::pow(d1/d2,
        ↪  2/(m-1)));
        w2[i][j] = 1/(std::pow(d2/d1, 2/(m-1)) + std::pow(d2/d2,
        ↪  2/(m-1)));
      }
    }

  } while(eps > 1);

  // cluster pixels based on weights
  for (int i = 0; i < img_binary.rows; i++) {
    for (int j = 0; j < img_binary.cols; j++) {
      img_binary.at<unsigned char>(i,j) = (w1[i][j] > w2[i][j]) ? 255 :
      ↪  0;
    }
  }

  // show and save binary image
  namedWindow("Display window", cv::WINDOW_AUTOSIZE);
  imshow("Display window", img_binary);
  cv::waitKey(0);
  imwrite("fcm.png", img_binary);

  return 0;
}
```

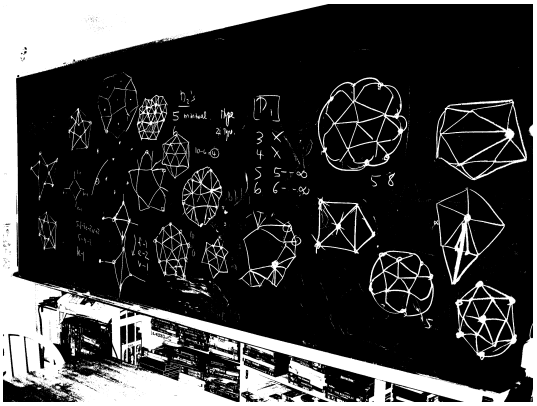
Rezultat izvršavanja algoritma je prikazan ispod.

Slika 1: input

Slika 2: output



Slika 3: input



Slika 4: output

2.2 FCM i k-means

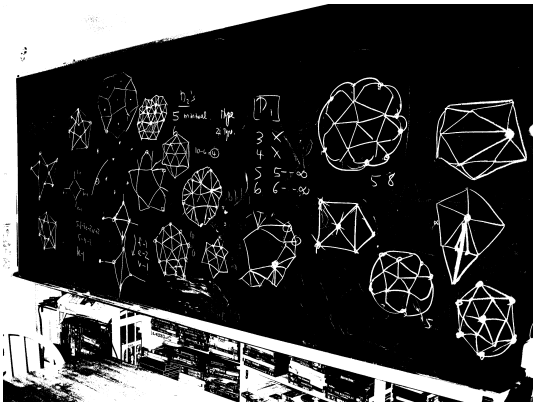
U ovom odeljku ćemo uporediti rezultate algoritama FCM i k-means, kao i vremena njihovih izvršavanja.

Napomena 2.1. Koristićemo efikasniju implementaciju FCM algoritma od one date u sekciji 2.1.

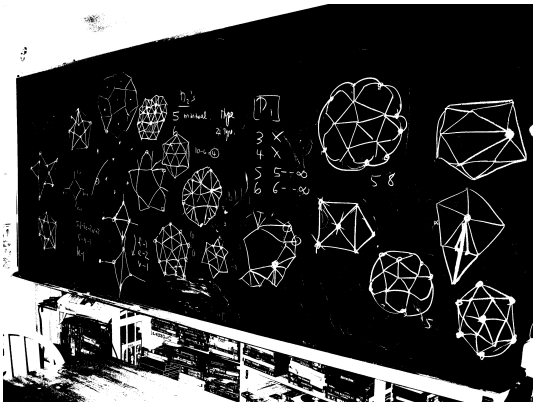
Na sledećim slikama su prikazani rezultati.

Slika 5: FCM output
Broj iteracija: 7

Slika 6: k-means output
Broj iteracija: 6



Slika 7: FCM output
Broj iteracija: 7



Slika 8: k-means output
Broj iteracija: 7

Možemo primetiti da su slike 5 i 6 identične, dok se slike 7 i 8 neznatno razlikuju. Međutim, vremena izvršavanja se primetno razlikuju. U Tabeli 1 su prikazana vremena (u sekundama) potrebna da algoritmi obrade Sliku 1 500, 1000 i 1500 puta. Slično, u Tabeli 2 su prikazana vremena potrebna da se obradi Slika 3 50, 100 i 150 puta.

broj izvršavanja	FCM	k-means
500	4	21
1000	8	42
1500	12	63

Tabela 1: Input Slika 1

broj izvršavanja	FCM	k-means
50	8	44
100	17	89
150	25	133

Tabela 2: Input Slika 3

Prikazaćemo još dva testa urađena na dve nove slike.

broj izvršavanja	FCM	k-means
100	15	119
150	22	179
200	31	238

Tabela 3:

broj izvršavanja	FCM	k-means
1000	3	36
1500	5	53
2000	6	71

Tabela 4:

Na osnovu podataka iz tabela, zaključujemo:

test	k-means/FCM
1	5
2	5
3	8
4	12

Tabela 5: Koliko puta je FCM brži od k-means

Treba napomenuti da su ovi rezultati okvirni, jer u velikoj meri zavise od implementacije samih algoritama. Naime, za centoride u k-means algoritmu je korišćen celobrojni tip

(int), dok je centroide u FCM algoritmu korišćen realni tip (float). U oba slučaja su se algoritmi zaustavljali kad je promena u centroidima bila manja od jedan. Dakle, već tu može doći do razlike u broju iteracija. Međutim, i dalje očekujemo da će FCM biti brži od k-means.
Takođe, ističemo da FCM koristi više memorije nego k-means.

3 Detekcija ivica

Detekcija ivica ima veliki značaj u obradi slika. Koristi se u raznim algoritmima kao što su segmentacija slike, detekcija i izdvajanje karakteristika, pa čak nekad i u kompresiji slika.

Ivice možemo definisati kao mesta na slici gde se intenzitet naglo menja, tj. gde je razlika vrednosti susednih piksela velika. Postoje mnogi algoritmi koji se bave ovim problemom, međutim, nijedan nije savršen. Primerom ćemo najbolje ilustrovati šta mislimo kad to kažemo. Naime, Sobelov algoritam je dobar kada treba detektovati oblike, ali ne radi dobro u realnom vremenu gde je brzina ključna (direktan prenos nekog događaja). Za takve situacije je prikladniji Canny algoritam.

U nastavku ćemo videti još jedan pristup ovom problemu. Koristićemo fazi logiku i fazi skupove.

3.1 FED

Kao što smo već napomenuli, koristićemo fazi logiku i fazi skupove da bismo detektovali ivice na slici. Preciznije, napravićemo fazi skup koji sadrži uređene parove (piksel, vrednost karakteristične funkcije). Taj skup će predstavljati ivice, dok će nam vrednosti karakteristične funkcije govoriti u kojoj meri piksel pripada tom skupu.
Postavlja se pitanje šta izabrati za karakterističnu funkciju. Podsetimo se, ivica je mesto gde se intenzitet naglo menja. Shodno tome, treba uzeti u obzir razliku intenziteta piksela koji trenutno posmatramo i njegovih suseda. Kada je ta razlika velika, vrednost naše funkcije treba da teži jedinici, a kada je razlika mala, treba da teži nuli. Jedna takva funkcija je:

$$\mu_{edge}(p) = 1 - \frac{1}{1 + \frac{\sum_{n \in N(p)} \|p-n\|}{L-1}}, \tag{7}$$

gde je p piksel, $N(p)$ skup piksela iz njegove okoline, L broj sivih nijansi (najčešće 256) i $\|\cdot\|$ norma koju ćemo definisati kao apsolutnu vrednost razlike intenziteta piksela.

Pre nego što damo kod, ukratko ćemo opisati korake algoritma:

- **Pretprocesiranje** - ako je slika u boji, konvertovati je u sivu sliku.
- **Pretprocesiranje** - primeniti Gausov filter na sliku kako bismo je malo zamutili.

- **Fazifikacija** - računanje karakteristične funkcije μ_{edge} za svaki piksel sa slike. Takodje, čuvanje najveće vrednosti funkcije (promenljiva MAX).
- **Normiranje vrednosti** - vrednosti karakteristične funkcije podeliti sa MAX:

$$\mu_{edge}(p) = \frac{\mu_{edge}(p)}{MAX}$$

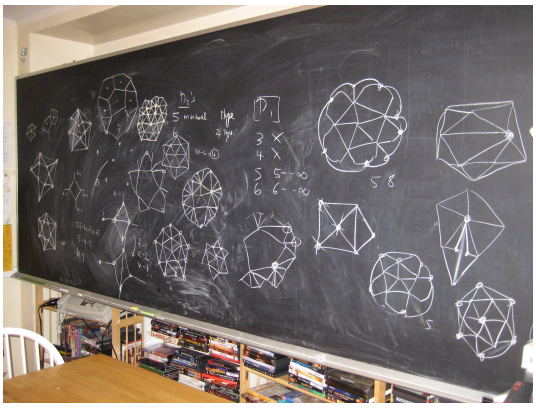
- **Defazifikacija** - na osnovu vrednosti $\mu_{edge}(p)$ i nekog unapred datog praga (threshold), pikselu p dodeliti crnu ili belu boju.

Pošto smo videli kratak opis algoritma, u nastavku dajemo kod radi boljeg razumevanja istog.

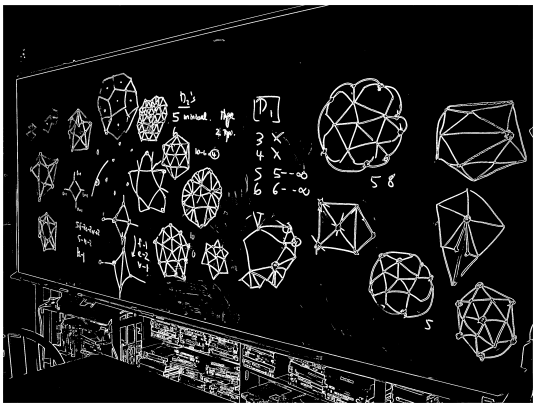
```
#include <opencv2/highgui/highgui.hpp>
#include <iostream>

int main(int argc, const char *argv[])
{
  if (argc != 3) {
    std::cerr << "Usage: ./binarization path_to_img threshold" <<
    ↪  std::endl;
    return 1;
  }

  // read image
  cv::Mat img = cv::imread(argv[1], cv::IMREAD_GRAYSCALE);

  float threshold = std::atof(argv[2]);
  const int L = 256;
  std::vector<std::vector<float>> mi(img.rows,
  ↪  std::vector<float>(img.cols, 0));
  auto output = gaussian_blur(img);

  float maxm = 0;
  for (int i = 1; i < output.rows-1; i++) {
    for (int j = 1; j < output.cols-1; j++) {
      unsigned s = 0;
      for (int x = -1; x <= 1; x++) {
        for (int y = -1; y <= 1; y++) {
          s += std::abs(output.at<unsigned char>(i,j)-output.at<unsigned
          ↪  char>(i+x,j+y));
        }
      }
      mi[i][j] = (1.0*s)/(s+(L-1));
      maxm = std::max(maxm, mi[i][j]);
```

```
    }
  }

  for (int i = 1; i < output.rows-1; ++i) {
    for (int j = 1; j < output.cols-1; j++) {
      output.at<unsigned char>(i,j) = (mi[i][j]/maxm <  threshold) ? 0 :
      ↪  255;
    }
  }

  namedWindow("Display window", cv::WINDOW_AUTOSIZE);
  imshow("Display window", output);
  cv::waitKey(0);
  imwrite("edge_detection.png", output);

  return 0;
}
```

Na Slici 10 je prikazan rezultat rada algoritma.

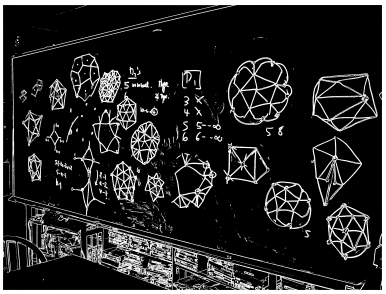


Slika 9: input

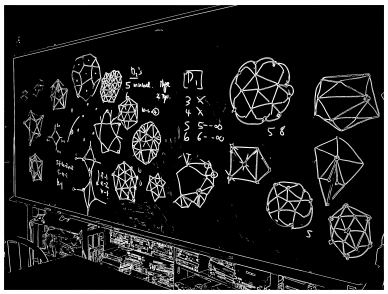


Slika 10: output

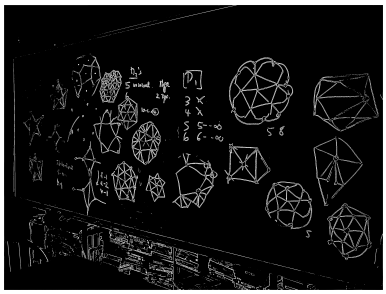
Na sledećim slikama možemo videti kako se rezultat menja u zavisnosti od praga koji se zadaje.



Slika 11: threshold = 0.25



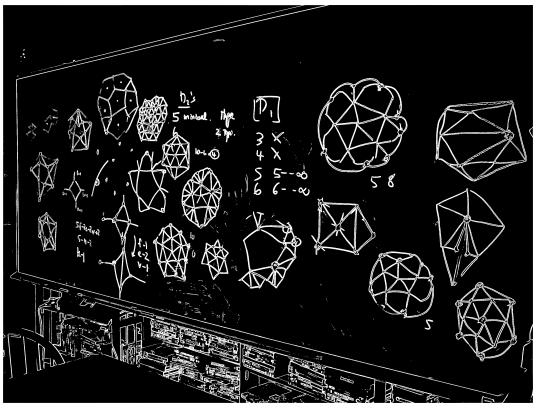
Slika 12: threshold = 0.35



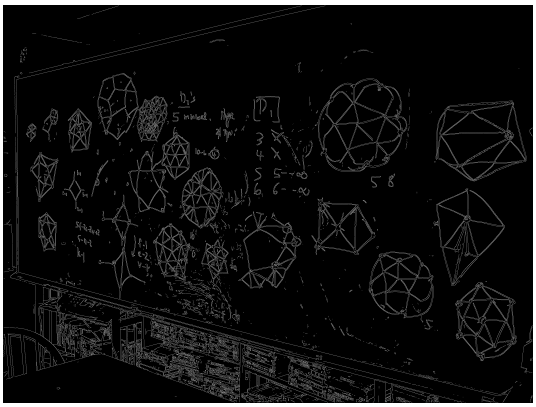
Slika 13: threshold = 0.5

3.2 FED i CED

Na Slikama 14 i 15 možemo videti izlaze ovih algoritama.

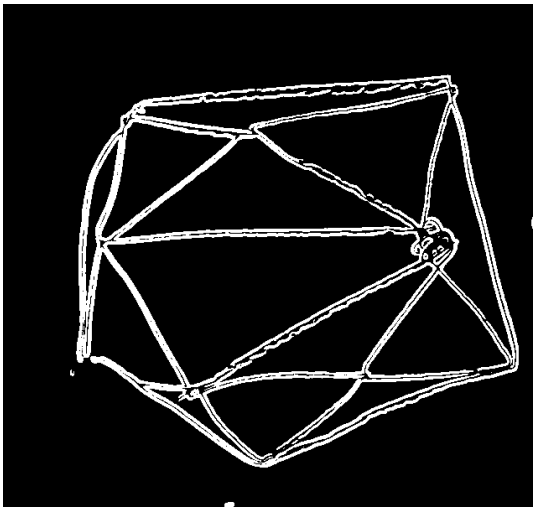


Slika 14: FED output

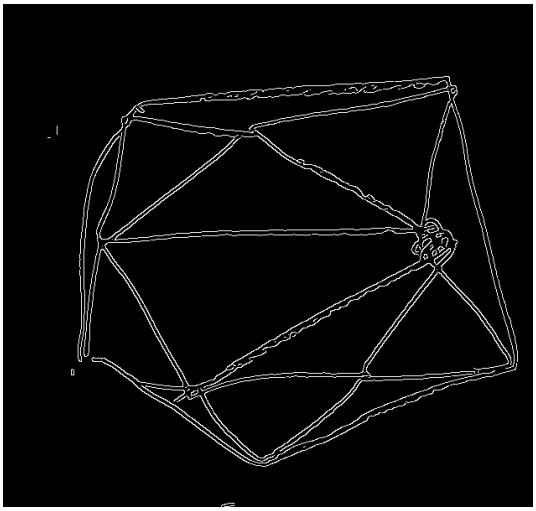


Slika 15: Canny output

Iako naizgled izgleda da FED daje bolje rezultate, ivice kod Canny algoritma su tanje (što je poželjno) i to možemo videti tako što ćemo prikazati uvećane delove Slika 14 i 15.



Slika 16: FED output



Slika 17: Canny output

U poređenju sa CED, FED je dosta brži. Naime, za obradu Slike 9 sto puta, FED algoritmu je bilo potrebno 60 sekundi, dok CED algoritmu 110. Naravno, ograničavamo se na ručnu implementaciju CED-a autora koja je najverovatnije sporija od implementacije istog algoritma u bibliotekama.

Još jedna prednost FED u odnosu na CED je to što se mnogo lakše implementira.

4 Zaključak

Teorija fazi skupova i fazi logika imaju značajnu primenu u procesiranju slika. Neke od tih primena smo videli i u ovom radu - binarizacija slike i detektovanje ivica. Postoje još mnogi algoritmi kao što su algoritmi za isticanje kontrasta, λ osvetljenja, λ negativa itd. Glavna prednost ovakvog pristupa procesiranju slika jeste jednostavnost njihove implementacije. Pored prednosti ovi pristupi imaju i mane. Nekad to može biti brzina izvršavanja, ili pak memorija. Takođe, i sami rezultati se mogu dosta razlikovati od rezultata algoritama koji nisu bazirani na fazi logici, kao što je to bio slučaj u detekciji ivica.

Literatura

[1] Nebojša Perić, "Neke primene teorije fazi skupova i fazi logike u procesiranju slika", Matematički fakultet u Beogradu, 2014.

[2] Mario I. Chacon M, "Fuzzy binarization and segmentation of text images for opcr", Mexico New Mexico State University