Prezentare proiect

Elena-Daniela Buda

Pentru retinerea pixelilor m-am folosit de o structura:

```
typedef struct
{
    unsigned char R;
    unsigned char G;
    unsigned char B;
}RGB;
```

Pentru prima parte (**criptarea/decriptarea**) am folosit urmatoarele functii:
Functia de **liniarizare**:

```
void liniarizare(char* nume_fisier_sursa, RGB **p, unsigned int *latime_img, unsigned int *inaltime_img)
{
    FILE *fin;
    int i, j, k;


    fin = fopen(nume_fisier_sursa, "rb");
    if(fin == NULL)
    {
        printf("Nu am gasit imaginea sursa din care citesc!");
        return;
    }

    fseek(fin, 18, SEEK_SET);
    fread(latime_img, sizeof(unsigned int), 1, fin);
    fread(inaltime_img, sizeof(unsigned int), 1, fin);
    k=((*inaltime_img)*(*latime_img)-1)-(*latime_img-1);

    int padding;

    if((*latime_img) % 4 != 0)
        padding = 4 - (3 * (*latime_img)) % 4;
    else
        padding = 0;


    *p = (RGB*) malloc(((*latime_img)*(*inaltime_img)-1)*sizeof(RGB));// alocam spatiu in p;
    if(p==NULL)
    {
        printf("Nu exista spatiu in heap\n");
        return;
    }

    fseek(fin,54,SEEK_SET);

    for(i = 0; i < *inaltime_img; i++)
    {
        for(j = 0; j < *latime_img; j++)
        {
            fread(&(*p)[k].B, sizeof(unsigned char), 1, fin);
            fread(&(*p)[k].G, sizeof(unsigned char), 1, fin);
            fread(&(*p)[k].R, sizeof(unsigned char), 1, fin);

            k++;
        }
        fseek(fin,padding,SEEK_CUR);
        k=k-(*latime_img)*2;
    }
    fclose(fin);
}
```

Ca parametrii am transmis numele imaginii, pointerul in care se vor introduce informatiile din pixelii imaginii si inaltimea si latimea pentru a fi retinute.

In functie se deschide imaginea pentru a putea fi citite in primul rand latimea si inaltime pozei utilizate la alocarea spatiului pentru valorile pixelilor si pentru calcularea padding-ului, apoi citirea valorilor RGB. Deoarece formatul imaginii este bmp, valorile pixelilor (culorile) se vor citi in ordinea Blue Green Red iar pixelii se vor citi incepand cu coltul jos stanga, de aceea in functie k ia valoarea initiala (H*W-1)-(W-1), iar apoi in for acesta va lua valoarea k=k-W*2.

In acest fel in pointer imaginea in loc sa fie memorata:

a b c
d e f -> g h i d e f a b c
g h i

aceasta va fi memorata:

a b c
d e f -> a b c d e f g h i
g h i

O alta functie folosita este aceea de copiere a imaginii din pointer in imagine:

```
void Copiere_Imagine(RGB *p, char* nume_img_copiata, unsigned int latime_img, unsigned int inaltime_img, char* nume_img_sursa)
{
    FILE* fout = fopen(nume_img_copiata, "wb");
    FILE* fin=fopen(nume_img_sursa, "rb");
    int k=0, i, j, pad=0;
    unsigned char a=0;
    int x;

    for(i = 0; i < 54; i++)
    {
        fread(&x , 1 , 1 , fin);
        fwrite(&x , 1 , 1 , fout);
    }
    fseek(fout,54,SEEK_SET);

    int padding;
    if((latime_img) % 4 != 0)
        padding = 4 - (3 * (latime_img)) % 4;
    else
        padding = 0;
    k=((inaltime_img)*(latime_img)-1)-(latime_img-1);

    for(i = 0; i < inaltime_img; i++)
    {
        for(j = 0; j < latime_img; j++)
        {
            fwrite(&(p)[k].B, sizeof(unsigned char), 1, fout);
            fwrite(&(p)[k].G, sizeof(unsigned char), 1, fout);
            fwrite(&(p)[k].R, sizeof(unsigned char), 1, fout);
            fflush(fout);
            k++;
        }
        if (padding!=0)
        for(j=0;j<padding;j++)
        {
            fwrite(&a, sizeof(unsigned char), 1, fout);
            pad++;
            fflush(fout);
        }
        k=k-(latime_img)*2;
    }
    fclose(fout);
    fclose(fin);
```

Aceasta functie primeste ca parametrii numele imaginii surse de unde va copia header-ul, numele imaginii destinatie, latimea si inaltimea imaginii si pointerul care contine informatiile pixelilor.

În functie copiem deader-ul în noua imagine, apoi pe rand pixelii din pointer si la finalul fiecarui rand, daca exista, adaugam si padding-ul (valori de 0).

Urmatoarele 2 functii sunt functiile de **XORSHIFT**, de **permutari** si de permutari inverse:

```
void xorshift(unsigned int seed, unsigned int latime_img, unsigned int inaltime_img, unsigned int **q)
{
    unsigned int k, r;
    *q=(unsigned int*) malloc((2*latime_img*inaltime_img-1)*sizeof(unsigned int));
    (*q)[0]=r=seed; // R[0] care retine seed-ul (primul numar din fisier
    for(k=1;k<=2*latime_img*inaltime_img-1;k++)
    {
        r=r^r<<13;
        r=r^r>>17;
        r=r^r<<5;
        (*q)[k]=r;
    }
}
void permutari(unsigned int *q, unsigned int latime_img, unsigned int inaltime_img, RGB **p)
{
    unsigned int rn;
    RGB aux;
    unsigned int i;
    for(i=1;i<=latime_img*inaltime_img-1;i++)
    {
        rn=((unsigned)q[latime_img*inaltime_img-i])%(i+1); // numar random intre 0 si i+1
        aux=(*p)[rn];
        (*p)[rn]=(*p)[i];
        (*p)[i]=aux;
    }
}
void permutari_inverse(unsigned int *q, unsigned int latime_img, unsigned int inaltime_img, RGB **p)
{
    unsigned int rn;
    RGB aux;
    unsigned int i;
    for(i=latime_img*inaltime_img-1;i>=1;i--)
    {
        rn=((unsigned)q[latime_img*inaltime_img-i])%(i+1); // numar random intre 0 si i+1
        aux=(*p)[rn];
        (*p)[rn]=(*p)[i];
        (*p)[i]=aux;
    }
}
```

Functia xorshift primeste ca parametrii seed-ul, inaltimea si latimea imaginii folosite la alocarea memoriei si pointerul în care se vor pune valorile.

In functie se aloca memoria necesara în pointerul q, q[0] retine valoarea seed-ului apoi se genereaza valorile care vor fi retinute în q[k].

Functia de permutari primeste ca parametrii pointerul în care sunt retinute valorile din xorshift, inaltimea si latimea imaginii si pointerul care contine valorile pixelilor.

În functie generam un numar random de la 0 la i+1 folosindu-ne de valorile din pointer-ul cu valori din xorshift, *i* fiind pozitia pe care ne aflam în "vectorul de pixeli" apoi interschimbam valoarea pixelului curent cu valoarea pixelului de pe pozitia random generata.

Functia de permutari inverse face acelasi lucru doar ca în sens invers.

Urmatoarea functie este aceea de **criptare**:

```
void criptare(char* nume_fisier_key, RGB **p, unsigned int **q, unsigned int latime_img, unsigned int inaltime_img)
{
    FILE* fkey = fopen(nume_fisier_key, "r");

    unsigned int seed, SV, k , m=~(~0<<8);//m este o masca de care ne folosim pentru a extrage fiecare octet din numar;
    if(fkey == NULL)
    {

        printf("Nu am gasit imaginea sursa din care citesc!");
        return;
    }
    fscanf(fkey,"%u", &seed);
    fscanf(fkey,"%u", &SV);

    xorshift(seed, latime_img, inaltime_img, *&q);

    permutari(*q,latime_img,inaltime_img, *&p);

    (*p)[0].R=(m&(SV>>16))^(*p)[0].R^(m&((*q)[latime_img*inaltime_img]>>16));
    (*p)[0].G=(m&(SV>>8))^(*p)[0].G^(m&((*q)[latime_img*inaltime_img]>>8));
    (*p)[0].B=(m&(SV))^(*p)[0].B^(m&((*q)[latime_img*inaltime_img]));

    for(k=1;k<=latime_img*inaltime_img-1;k++)
    {
        (*p)[k].R=(*p)[k-1].R^(*p)[k].R^(m&((*q)[latime_img*inaltime_img+k]>>16));
        (*p)[k].G=(*p)[k-1].G^(*p)[k].G^(m&((*q)[latime_img*inaltime_img+k]>>8));
        (*p)[k].B=(*p)[k-1].B^(*p)[k].B^(m&((*q)[latime_img*inaltime_img+k]));
    }

    fclose(fkey);

}
```

Functia primeste ca parametrii numele fisierului care contine seed-ul si SV-ul, pointerul ce contine valorile pixelilor, pointerul ce va contine valorile din xorshift si inaltimea si latimea imaginii.

În functie citim seed-ul si SV-ul din fisier apoi apelam functia de xorshift pentru a genera numerele apoi functia de permutari pentru a genera permutarile.

Ne vom folosi de o masca pentru a extrage bitii din SV apoi vom xora pixelii intre ei si cu valorile generate de xorshift.

Functia de **decriptare** merge pe acelasi principiu doar ca mai intai xoram pixelii intre ei si cu numerele generate de xorshift apoi apelam functia de permutari inverse.

```
void decriptare(char* nume_fisier_key, RGB **p, unsigned int *q, unsigned int latime_img, unsigned int inaltime_img)
{
    FILE* fkey = fopen(nume_fisier_key, "r");
    int k;
    unsigned int seed, SV, m=~(~0<<8);//m este o masca de care ne folosim pentru a extrage fiecare octet din numar;

    if(fkey == NULL)
    {
        printf("Nu am gasit imaginea sursa din care citesc!");
        return;
    }
    fscanf(fkey,"%u", &seed);
    fscanf(fkey,"%u", &SV);
    fclose(fkey);
    RGB* p_copy=(RGB*) malloc(((latime_img)*(inaltime_img)-1)*sizeof(RGB));

    for(k=0;k<=(latime_img)*(inaltime_img)-1;k++)
    {
        p_copy[k].B=(*p)[k].B;
        p_copy[k].G=(*p)[k].G;
        p_copy[k].R=(*p)[k].R; //facem o copie a pixelilor imaginii criptate
    }
    (*p)[0].R=(m&(SV>>16))^p_copy[0].R^(m&((q)[latime_img*inaltime_img]>>16));
    (*p)[0].G=(m&(SV>>8))^p_copy[0].G^(m&((q)[latime_img*inaltime_img]>>8));
    (*p)[0].B=(m&(SV))^p_copy[0].B^(m&((q)[latime_img*inaltime_img]));
    for(k=1;k<=latime_img*inaltime_img-1;k++)
    {
        (*p)[k].R=p_copy[k-1].R^p_copy[k].R^(m&((q)[latime_img*inaltime_img+k]>>16));
        (*p)[k].G=p_copy[k-1].G^p_copy[k].G^(m&((q)[latime_img*inaltime_img+k]>>8));
        (*p)[k].B=p_copy[k-1].B^p_copy[k].B^(m&((q)[latime_img*inaltime_img+k]));

    }
    permutari_inverse(q,latime_img,inaltime_img, *&p);
    free(p_copy);
}
```

Ultima functie folosita pentru partea de criptare si decriptare este functia chi în care se face testul **Chi-squared**:

```
void chi( RGB *p, unsigned int latime_img, unsigned int inaltime_img)
{
    double chiR=0.0, chiG=0.0, chiB=0.0, norm;
    unsigned int *frecvR, *frecvG, *frecvB;
    frecvR=(unsigned int*)calloc(256,sizeof(unsigned int));
    frecvG=(unsigned int*)calloc(256,sizeof(unsigned int));
    frecvB=(unsigned int*)calloc(256,sizeof(unsigned int));
    int i;
    norm=(double)(latime_img*inaltime_img)/256;
    for(i=0;i<=latime_img*inaltime_img-1;i++)
    {
        frecvR[p[i].R]++;
        frecvG[p[i].G]++;
        frecvB[p[i].B]++;
    } // calculam frecventa valorilor de pe fiecare canal
    for(i=0;i<=255;i++)
        {
            chiR+=(double)(pow((double)frecvR[i]-norm,2.0)/norm);
            chiG+=(double)(pow((double)frecvG[i]-norm,2.0)/norm);
            chiB+=(double)(pow((double)frecvB[i]-norm,2.0)/norm);
        }

    printf("R: %.2lf\nG: %.2lf\nB: %.2lf\n",chiR, chiG, chiB);

    free(frecvB);
    free(frecvR);
    free(frecvG);
}
```

Functia primeste ca parametru pointerul ce contine valorile pixelilor imaginii testate, latimea si inaltimea imaginii.

În functie se creaza 3 vectori de frecventa care vor retine pentru fiecare canal frecventa culorii (valorile cuprinse intre 0 si 255).

Apoi pentru fiecare culoare se calculeaza suma Chi-Squared si se afiseaza.

Pentru a retine corelatiile si valorile coordonatelor din "matricea de pixeli" am folosit o structura:

```
typedef struct
{
    unsigned int lin;
    unsigned int col;
    double inf;
}cor;
```

Functiile folosite pentru a 2-a parte (Template Matching) sunt:

Functia **grayscale** primita în fisierul cu proiectul:

```
void grayscale_image(char* nume_fisier_sursa,char* nume_fisier_destinatie)
{
   FILE *fin, *fout;
   unsigned int latime_img, inaltime_img;
   unsigned char pRGB[3], aux;

   fin = fopen(nume_fisier_sursa, "rb");
   if(fin == NULL)
    {
        printf("nu am gasit imaginea sursa din care citesc");
        return;
    }

   fout = fopen(nume_fisier_destinatie, "wb+");

   fseek(fin, 2, SEEK_SET);

   fseek(fin, 18, SEEK_SET);
   fread(&latime_img, sizeof(unsigned int), 1, fin);
   fread(&inaltime_img, sizeof(unsigned int), 1, fin);

   //copiaza octet cu octet imaginea initiala in cea noua
    fseek(fin,0,SEEK_SET);
    unsigned char c;
    while(fread(&c,1,1,fin)==1)
    {
        fwrite(&c,1,1,fout);
        fflush(fout);
    }
    fclose(fin);

    //calculam padding-ul pentru o linie
    int padding;
    if(latime_img % 4 != 0)
        padding = 4 - (3 * latime_img) % 4;
    else
        padding = 0;

    fseek(fout, 54, SEEK_SET);
    int i,j;
    for(i = 0; i < inaltime_img; i++)
    {
        for(j = 0; j < latime_img; j++)
        {
            //citesc culorile pixelului
            fread(pRGB, 3, 1, fout);
            //fac conversia in pixel gri
            aux = 0.299*pRGB[2] + 0.587*pRGB[1] + 0.114*pRGB[0];
            pRGB[0] = pRGB[1] = pRGB[2] = aux;
            fseek(fout, -3, SEEK_CUR);
            fwrite(pRGB, 3, 1, fout);
            fflush(fout);
        }
        fseek(fout,padding,SEEK_CUR);
    }
    fclose(fout);
}
```

Am folosit si o functie care face o matrice dintr-un vector:

```
void imagine_matrice(RGB ***matrice, RGB *p, unsigned int latime_img, unsigned int inaltime_img)
{
    unsigned i, j, k=0;

    for(i = 0; i < inaltime_img; i++)
    {
        for(j = 0; j < latime_img; j++)
        {
            (*matrice)[i][j]=p[k];
             k++;
        }
    }
}
```

Pentru a lucra mai usor la partea de corelatie.

Functia de **conturare**:

```
void conturare(cor *D, RGB culoare, unsigned int lungime, RGB **p, unsigned int latime_poza, unsigned int latime_sablon, unsigned int inaltime_sablon)
{
    unsigned int i, j, k;
    //printf("%d", lungime);
    for(i=0;i<=lungime;i++)
    {
        for(j=0;j<latime_sablon;j++)
        {
            (*p)[D[i].lin*latime_poza+D[i].col+j]=culoare;
            (*p)[(D[i].lin+(inaltime_sablon-1))*latime_poza+D[i].col+j]=culoare;
        }
        for(j=1;j<inaltime_sablon-1;j++)
        {
            (*p)[(D[i].lin+(j))*latime_poza+D[i].col]=culoare;
            (*p)[(D[i].lin+(j))*latime_poza+D[i].col+(latime_sablon-1)]=culoare;
        }
    }

}
```

Functia primeste ca parametru pointerul D în care se afla "coordonatele" din matricea creata cu valorile pixelilor imaginii din coltul sus stanga si corelatia calculata, numarul de valori din pointer, pointerul p (vector) care contine valorile pixelilor din imagine liniarizate si dimensiunile pozei si al sablonului pentru a determina pozitiile care trebuie sa fie colorate în imaginea finala.

Functia **glisare** este functia în care se calculeaza corelatia:

```
unsigned int glisare(cor **D, char* nume_sablon, double prag)
{
    unsigned int i, j, k, latime_p, inaltime_p, latime_s, inaltime_s, l, m=0, n;
    double corr, sum, meds=0.0, medp=0.0, sigmap, sigmas, sump, sums=0.0;
    char nume_img_sursa[] = "test.bmp",\
    nume_img_gs[]="test-gs.bmp", nume_sablon_gs[]="cifra0-gs.bmp";
    RGB **poza=NULL;
    RGB **sabl=NULL, *p, *q;

    grayscale_image(nume_img_sursa,nume_img_gs);
    grayscale_image(nume_sablon,nume_sablon_gs);

    liniarizare(nume_img_gs, &p, &latime_p, &inaltime_p);
    liniarizare(nume_sablon_gs, &q, &latime_s, &inaltime_s);

    poza=(RGB**)malloc((inaltime_p)*sizeof(RGB*));

    for(i = 0; i < inaltime_p; i++)
    {
        (poza)[i]=(RGB*)calloc(latime_p,sizeof(RGB));
    }
    sabl=(RGB**)malloc((inaltime_s)*sizeof(RGB*));

    for(i = 0; i < inaltime_s; i++)
    {
```

```
        (sabl)[i]=(RGB*)calloc(latime_s,sizeof(RGB));
    }
    imagine_matrice(&poza, p,latime_p, inaltime_p);
    imagine_matrice(&sabl, q, latime_s, inaltime_s);

    free(p);
    p=NULL;
    free(q);
    q=NULL;

    n=latime_s*inaltime_s;
    *D=(cor*)malloc((latime_p-latime_s)*(inaltime_p-inaltime_s)*sizeof(cor));
    for(i=0;i<inaltime_s;i++)
        for(j=0;j<latime_s;j++)
        {
            meds+=sabl[i][j].R;
        }
    meds=(double)meds/(double)(n);

    for(i=0;i<inaltime_s;i++)
        for(j=0;j<latime_s;j++)
            sums=(double)sums+(double)pow((double)(sabl[i][j].R-(double)meds), 2.0);

    sigmas=(double)sqrt((double)sums/(n-1));
    ///calculam ce ne trebuie pentru sablon in afara forului care merge pe imaginea mare deoarece sablonul nu se modifica
    ///ci doar fereastra
    for(k=0;k<=inaltime_p-inaltime_s;k++)
        for(l=0;l<=latime_p-latime_s;l++)
        {
            medp=0.0;
            sump=0.0;
            sigmap=0.0;
            corr=0.0;
            sum=0.0;
        for(i=0;i<inaltime_s;i++)
            for(j=0;j<latime_s;j++)
            {
                medp+=poza[i+k][j+l].R;
            }
        medp=(double)medp/(double)(n);

        for(i=0;i<inaltime_s;i++)
            for(j=0;j<latime_s;j++)
                sump=(double)sump+(double)pow((double)(poza[i+k][j+l].R-(double)medp), 2.0);

        sigmap=(double)sqrt((double)sump/(n-1));
        for(i=0;i<inaltime_s;i++)
            for(j=0;j<latime_s;j++)
            {
                sum+=((double)(poza[i+k][j+l].R-medp)*(sabl[i][j].R-meds))/(double)(sigmas*sigmap);
            }
            corr=(double)sum/(double)n;

            if((double)corr>=prag)
                {
                   // printf("%.2f\n",corr);
                    (*D)[m].col=l;
                    (*D)[m].lin=k;
                    (*D)[m].inf=corr;
                   // printf("%d %d %.2f\n", (*D)[m].col, (*D)[m].lin, (*D)[m].inf);
                    m++;
                }
        }
    free2Darray(sabl,inaltime_s);
    free2Darray(poza,inaltime_p);


    if(m>0)
        m=m-1;
    return m;
}
```

Functia primeste ca parametrii pointerul D pentru a retine corelatiile, numele sablonului si pragul de corelatie.

În funtie calculam separat valorile necesare pentru sablon deoarece acesta nu se modifica apoi intr-un for care merge pe inaltimea imaginii mari, insa nu pana jos ca sa evitam cazul în care sablonul iese din imagine si intr-un for care merge pe latimea imaginii cu aceeasi restrictie

calculam datele necesare pentru imaginea mare folosind for-uri care se folosesc de dimensiunle sablonului dat pentru a calcula doar datele din fereasta nu din toata imaginea (sigma, media etc.) apoi calculam corelatia, iar daca aceasta este mai mare sau egala cu pragul o adaugam în vectorul de corelatii si returnam dimensiunea vectorului (numarul de corelatii gasite).

Ultima functie folosita este cea care se foloseste la **qsort** (functia de comparare):

```
int cmpCor(const void *a, const void *b)
{
    if((double)(((cor*)a)->inf)-(double)(((cor*)b)->inf)<0)
        return 1;
    return -1;
}
```

Main-ul este creat pentru a testa daca functiile merg.

Rezultatele dupa aplicarea functiilor:

Testul Chi-Squared:

```
Testul chi pentru imaginea initiala:
R: 13653913.26
G: 13875675.57
B: 13857265.15

Se liniarizeaza imaginea criptata. . .


Testul chi pentru imaginea criptata:
R: 261.88
G: 241.78
B: 255.16
```

Template matching aplicat pentru sablonul cifra0.bmp fara eliminarea suprapunerilor cu prag 0.5: