Criptografia și procesarea digitală

Explicarea structurilor

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

typedef struct
{
    int X1,Y1,X2,Y2;
    int cifra_det;
    double valoare_corelatie;
} fereastra;

typedef struct
{
    fereastra *V;
    unsigned int NumarDetectii;
} ferestre_sablon;
```

Structura denumită **RGB** este folosită pentru a stoca canalele (red, green, blue) pentru fiecare pixel dintr-o imagine de tip BMP și pentru a stoca culorile setate pentru desenarea conturului ferestrelor detectate după operația de template matching și de înlăturare a non-maximelor. Variabilele din structură sunt de tipul unsigned char pentru a stoca valori între 0 și 255, valorile canalelor unui pixel fiin între aceleași 2 valori.

Structura denumită **fereastra** este folosită la operația de template matching pentru a memora colțul stânga sus al ferestrei detectate în punctul de coordonate (X1,Y1), respectiv colțul dreapta jos în punctul de coordonate (X2,Y2). Pe lângă cele două colțuri o să mai ținem minte valoarea corelației dintre o fereastra din imagine și un șablon și cifra din șablon în cazul în care valoarea corelației este mai mare decât pragul stabilit.

Structura denumită **ferestre_sablon** este folosită pentru a reține un pointer la un tablou ce conține toate ferestrele detectate și numărul acestora.

Explicarea funcțiilor

```
unsigned int xorshift32(unsigned int *seed)
{
    unsigned int x;
    x=*seed;
    x^= x << 13;
    x^= x >> 17;
    x^= x << 5;
    *seed=x;
    return x;
}

void grayscale_image(char* nume_fisier_sursa,char* nume_fisier_destinatie)
{
    FILE *fin, *fout;
    unsigned int dim_img, latime_img, inaltime_img;
    unsigned char *pRGB, aux;
    pRGB=(unsigned char*)malloc(3*sizeof(unsigned char));
    if(pRGB==NULL)
    {
        printf("Eroare la alocarea memoriei dinamice in functia grayscale_image./n");
        return ;
    }
    fin = fopen(nume_fisier_sursa, "rb");
```

Funcția **xorshift32** este implementată după algoritmul găsit pe wikipedia (https://en.wikipedia.org/wiki/Xorshift), iar funcția de transformare a unei imagini color BMP în una grayscale este cea primită în folder-ul cod de la proiect, la care am modificat vectorul inițial în unul alocat dinamic.

```
RGB* IncarcareFormaLiniarizata(char *CaleImagine)
{
    FILE *f;
    unsigned int Width,Height,padding;
    RGB *V;
    int i,j;
    f=fopen(CaleImagine,"rb");
    if(f==NULL)
    {
        printf("Eroare la deschiderea imaginii pentru a o liniariza.\n");
        return NULL;
    }
    fseek(f,18,SEEK_SET);
    fread(&Width,sizeof(unsigned int),1,f);
    fread(&Height,sizeof(unsigned int),1,f);
    V=(RGB*)malloc(Width*Height*sizeof(RGB));
    if(V==NULL)
    {
        printf("Eroare la alocarea memoriei dinamice in functia IncarcareFormaLiniarizata.\n");
        return NULL;
    }
    if(Width%4)
        padding=4-(Width*3)%4;
    else
        padding=0;
    fseek(f,54,SEEK_SET);
    for(i=Height-1; i>=0; i--)
    {
        for(j=0; j<Width; j++)
            fread((V+Width*i+j),3,1,f);
        fseek(f,padding,SEEK_CUR);
    }
    fclose(f);
    return V;
}
```

Funcția **IncarcareFormaLiniarizata** primește calea unei imaginii BMP și returnează un tablou de structuri RGB în care stochează intensitățile canalelor pixelilor. În fișierul binar pixelii al imaginii sunt stocați de jos în sus, așa că prima linie citită, e de fapt ultima, așa că o vom salva la sfârșitul tabloului. Astfel liniile citite le vom pune de la final spre început în tablou.

```
unsigned int* generare_permutare_random(unsigned int *R,unsigned int nr)
{
    unsigned int i,r,*permutare,aux;
    permutare=(unsigned int *)malloc(nr*sizeof(unsigned int));
    if(permutare==NULL)
    {
        printf("Eroare la alocarea dinamica a memoriei in functia generare_permutare_random.\n");
        return NULL;
    }
    for(i=0; i<nr; i++)
        *(permutare+i)=i;
    for(i=nr-1; i>=1; i--)
    {
        r=*(R+nr-i-1);
        if(r>i)
            r=r%(i+1);
        aux=*(permutare+r);
        *(permutare+r)=*(permutare+i);
        *(permutare+i)=aux;
    }
    return permutare;
}
```

Funcția **generare_permutare_random** primește un pointer la un tablou de numere generate cu ajutorul funcției de xorshift pornind de la o valoare dată (generarea lor are loc în funcția de criptare, respectiv decriptare înainte de apel), iar variabila nr reprezintă numărul de elemente ale permutării ce trebuie generate. Inițial permutarea este cea identică, urmând să aplicăm algoritmul lui Durstenfeld pentru a crea o permutare aleatoare. La final returnează un pointer către un tablou de dimensiunea nr cu o permutare aleatoare.

```
void SalvareExternFormaLiniarizata(char *cale_imagine,char *cale_imagine_destinatie,RGB* VectorImagine)
{
    FILE *f,*g;
    int i,j;
    unsigned int Width,Height,padding;
    unsigned char *header,val=0;
    header=(unsigned char*)malloc(54);
    if(header==NULL)
    {
        printf("Eroare la alocarea memoriei dinamice in functia SalvareExternFormaLiniarizata.\n");
        return ;
    }
    f=fopen(cale_imagine,"rb");
    if(f==NULL)
    {
        printf("Eroare la deschiderea imaginii sursa.\n");
        return;
    }
    fread(header,54,1,f);
    fseek(f,18,SEEK_SET);
    fread(&Width,sizeof(unsigned int),1,f);
    fread(&Height,sizeof(unsigned int),1,f);
    fclose(f);
    if(Width%4)
        padding=4-(Width*3)%4;
    else
        padding=0;
    g=fopen(cale_imagine_destinatie,"wb");
    if(g==NULL)
    {
        printf("Eroare la deschiderea fisierului destinatie a imaginii.\n");
        return;
    }
    fwrite(header,54,1,g);
```

Funcția **SalvareExternFormaLiniarizata** primește calea imaginii inițiale pentru a putea copia din ea header-ul, calea imaginii unde se va salva noua imagine și un pointer la un vector de structuri RGB care conține pixelii ce trebuie scriși în noua imagine. Copierea pixelilor din tablou în nou imagine se face conform formatului BMP, ultima linie din imagine de pixeli trebuind să fie scrisă prima în fișier și tot așa.

```
void criptare(char *cale_imagine,char *cale_imagine_criptata,char *cheia_secreta)
```

Acesta este antetul funcției de **criptare**.

```
    R=(unsigned int*)malloc((2*Width*Height-1)*sizeof(unsigned int));
    if(R==NULL)
    {
        printf("Eroare la alocarea memoriei dinamice in functia de criptare.\n");
        return ;
    }
    for(i=0; i<2*Width*Height-1; i++)
        *(R+i)=xorshift32(&seed);

//generare permutare random cu ajutor primilor Width*Height-1 numere generate mai sus
    Permutare=generare_permutare_random(R,Width*Height);
    if(Permutare==NULL)
        return;
//permutam pixelii imaginii liniarizate conform permutarii generate mai sus
    P_permutat=(RGB*) malloc((Width*Height)*sizeof(RGB));
    if(P_permutat==NULL)
    {
        printf("Eroare la alocarea memoriei dinamice in functia de criptare.\n");
        return ;
    }
    for(i=0; i<Width*Height; i++)
        *(P_permutat + *(Permutare+i)) = *(P+i);

//XOR-am pixelii imaginii liniarizate
    unsigned char *aux1,*aux2;
    aux1=(unsigned char*)&SW;
    aux2=(unsigned char*)(R+Width*Height-1);
    P_permutat->R = (*(aux1+2)) ^ (P_permutat->R) ^ (*(aux2+2));
    P_permutat->G = (*(aux1+1)) ^ (P_permutat->G) ^ (*(aux2+1));
    P_permutat->B = (*aux1) ^ (P_permutat->B) ^ (*aux2);
    for(i=0; i<Width*Height-1; i++)
    {
        aux2=(unsigned char*)(R+Width*Height+i);
        (P_permutat+i+1)->R = ((P_permutat+i)->R) ^ ((P_permutat+i+1)->R) ^ (*(aux2+2));
        (P_permutat+i+1)->G = ((P_permutat+i)->G) ^ ((P_permutat+i+1)->G) ^ (*(aux2+1));
        (P_permutat+i+1)->B = ((P_permutat+i)->B) ^ ((P_permutat+i+1)->B) ^ (*aux2);
    }
    SalvareExternFormaLiniarizata(cale_imagine,cale_imagine_criptata,P_permutat);
```

Asta este partea mai importantă din funcția de **criptare**. Inițial generăm numerele aleatoare cu funcția xorshift32 si le stocăm într-un tablou alocat dinamic. Apoi creăm o permutare aleatoare cu ajutorul funcției generare_permutare_random. Acum urmează să permutăm pixelii imaginii inițiale conform permutării random generate. Rezultatul acestei operații fiind salvat într-un tablou alocat dinamic de structuri RGB. Pasul final al criptării este să schimbăm valorile intensităților canalelor pixelilor. Pentru asta ne vom folosi de restul numerelor generate mai sus și de o valoare dată în fișier. Acum în tabloul ce începe de la adresa memorată în pointer-ul P_Permutat avem valorile pixelilor permutați și criptați, urmând să creăm noua imagine pe baza acestui tablou.

```
void decriptare(char *cale_imagine_criptata,char *cale_imagine_decriptata,char *cheia_secreta)
```

Acesta este antetul funcției de **decriptare**.

```
    R=(unsigned int*)malloc((2*Width*Height-1)*sizeof(unsigned int));
    if(R==NULL)
    {
        printf("Eroare la alocarea memoriei dinamice in functia de decriptare.\n");
        return ;
    }
    for(i=0; i<2*Width*Height-1; i++)
        *(R+i)=xorshift32(&seed);

    P=IncarcareFormaLiniarizata(cale_imagine_criptata);
    if(P==NULL)
    {
        printf("Eroare la liniarizarea imaginii.\n");
        return;
    }
/XOR-am pixelii imaginii liniarizate pentru a ajunge la valorile initiale ale pixelilor

    for(i=Width*Height-1; i>=1; i--)
    {
        aux2=(unsigned char*)(R+Width*Height+i-1);
        (P+i)->R = ((P+i-1)->R) ^ ((P+i)->R) ^ (*(aux2+2));
        (P+i)->G = ((P+i-1)->G) ^ ((P+i)->G) ^ (*(aux2+1));
        (P+i)->B = ((P+i-1)->B) ^ ((P+i)->B) ^ (*aux2);
    }
    aux2=(unsigned char*)(R+Width*Height-1);
    aux1=(unsigned char*)&SW;
    P->R = (*(aux1+2)) ^ (P->R) ^ (*(aux2+2));
    P->G = (*(aux1+1)) ^ (P->G) ^ (*(aux2+1));
    P->B = (*aux1) ^ (P->B) ^ (*aux2);

/generare permutare random cu ajutor primilor Width*Height-1 numere generate mai sus
    Permutare=generare_permutare_random(R,Width*Height);
    if(Permutare==NULL)
        return ;
/permutam pixelii imaginii liniarizate conform permutarii inverse
    P_permutat=malloc((Width*Height)*sizeof(RGB));
```

Aici procedeul este exact invers față de cel de la criptare. După ce liniarizăm imaginea criptată, generăm aceeași secvență de numere aleatoare folosindu-ne de aceeași valoare de start ca la criptare. Și cu ajutorul acestora decriptăm pixelii pornind de la sfârșit spre primul pixel. După ce am restaurat valorile intensităților pixelilor, urmează să generăm permutarea aleatoare folosită la criptare și să îi calculăm inversa. Apoi ne rămâne să permutăm pixelii conform permutării inverse și să o salvăm.

```
Vector_frecvente_B=(unsigned int*)calloc(256,sizeof(unsigned int));
if(Vector_frecvente_B==NULL)
{
    printf("Eroare la alocarea memoriei dinamice in functia test_chi_patrat.\n");
    return ;
}
double fmed,chi_patrat_R=0,chi_patrat_G=0,chi_patrat_B=0;
fmed=(Width*Height)/256;
Vector_imagine=IncarcareFormaLiniarizata(cale_imagine);
if(Vector_imagine==NULL)
{
    printf("Eroare la liniarizarea imaginii.\n");
    return;
}
for(i=0; i<Width*Height; i++)
{
    *(Vector_frecvente_R+((Vector_imagine+i)->R))=*(Vector_frecvente_R+((Vector_imagine+i)->R))+1;
    *(Vector_frecvente_G+((Vector_imagine+i)->G))=*(Vector_frecvente_G+((Vector_imagine+i)->G))+1;
    *(Vector_frecvente_B+((Vector_imagine+i)->B))=*(Vector_frecvente_B+((Vector_imagine+i)->B))+1;
}
for(i=0; i<=255; i++)
{
    chi_patrat_R+=((*(Vector_frecvente_R+i)-fmed)*(*(Vector_frecvente_R+i)-fmed))/fmed;
    chi_patrat_G+=((*(Vector_frecvente_G+i)-fmed)*(*(Vector_frecvente_G+i)-fmed))/fmed;
    chi_patrat_B+=((*(Vector_frecvente_B+i)-fmed)*(*(Vector_frecvente_B+i)-fmed))/fmed;
}
printf("(%lf,%lf,%lf)\n",chi_patrat_R,chi_patrat_G,chi_patrat_B);
```

Asta este partea mai complexă a funcției ce calculează valorile testului **chi-pătrat** și le afișează pe ecran. Avem nevoie de 3 tablouri alocate dinamic cu calloc pentru că le vom folosi ca tablouri de frecvențe și avem nevoie ca valorile inițiale să fie 0, ce vor reține numărul de apariții a fiecărei intensități de pe fiecare canal de culoare a pixelilor. Folosindu-ne de aceste 3 tablouri vom putea calcula valorile testului **chi-pătrat** pe fiecare canal conform formulei:

$$\chi^2 = \sum_{i=0}^{255} \frac{(f_i - \bar{f})^2}{\bar{f}}$$

```
double mediavalorilor(RGB *V,unsigned int Width,unsigned int Height)
{
    int i;
    unsigned int n;
    double Smed=0;
    n=Width*Height;
    for(i=0; i<Height*Width; i++)
        Smed+=(V+i)->R;
    Smed=Smed/n;
    return Smed;
}

double deviatia(RGB *V,unsigned int Width,unsigned int Height,double med)
{
    int i;
    unsigned int n;
    double S=0;
    n=Width*Height;
    for(i=0; i<Height*Width; i++)
        S+=((V+i)->R-med)*((V+i)->R-med);
    S=S/(n-1);
    S=sqrt(S);
    return S;
}
```

Aceste două funcții ajută la calcularea corelației dintre un șablon și o fereastră de dimensiunea șablonului din imagine. Funcția **mediavalorilor** primește un pointer la un tablou de structuri RGB ce reprezintă pixelii unei ferestre dintr-o imagine grayscale, lățimea ferestrei și înălțimea ferestrei și returnează valoarea intensității medie. Funcția **deviatia** are primele 3 argumente la fel ca funcția mediavalorilor, iar ultimul argument reprezintă valoarea returnată de funcția mediavalorilor pentru fereastra respectivă. Returnează valoare deviației conform formulei:

$$\sigma_S = \sqrt{\frac{1}{n-1}\sum_{(i,j)\in S}(S(i,j) - \bar{S})^2}$$

```
double corelatie(RGB *S,unsigned int WidthS,unsigned int HeightS,RGB *I,unsigned int Width,unsigned int Height,int Iy,int Ix)
{
    RGB *ImagineSubSablon;
    ImagineSubSablon=(RGB *)malloc(WidthS*HeightS*sizeof(RGB));
    if(ImagineSubSablon==NULL)
    {
        printf("Eroare la alocarea memoriei dinamice in functia de calculare a corelatiei.\n");
        return -2;
    }
    int Sx,Sy,i,j,nr=0;
    Sx=Ix-(WidthS/2);
    Sy=Iy-(HeightS/2);

    for(i=Sy; i<Sy+(int)HeightS; i++)
        for(j=Sx; j<Sx+(int)WidthS; j++)
        {
            if(i>=0 && i<Height && j>=0 && j<Width)
                (ImagineSubSablon+nr)->R=(I+i*Width+j)->R;
            else
                (ImagineSubSablon+nr)->R=0;
            nr++;
        }
    double DS,DI,corel=0,Smed,Imed;
    Smed=mediavalorilor(S,WidthS,HeightS);
    Imed=mediavalorilor(ImagineSubSablon,WidthS,HeightS);
    DS=deviatia(S,WidthS,HeightS,Smed);
    DI=deviatia(ImagineSubSablon,WidthS,HeightS,Imed);
    for(i=0; i<HeightS; i++)
        for(j=0; j<WidthS; j++)
            corel+=(((ImagineSubSablon+i*WidthS+j)->R - Imed) * ((S+i*WidthS+j)->R - Smed)) / (DS*DI);
    corel=corel/(WidthS*HeightS);
    free(ImagineSubSablon);
    return corel;
}
```

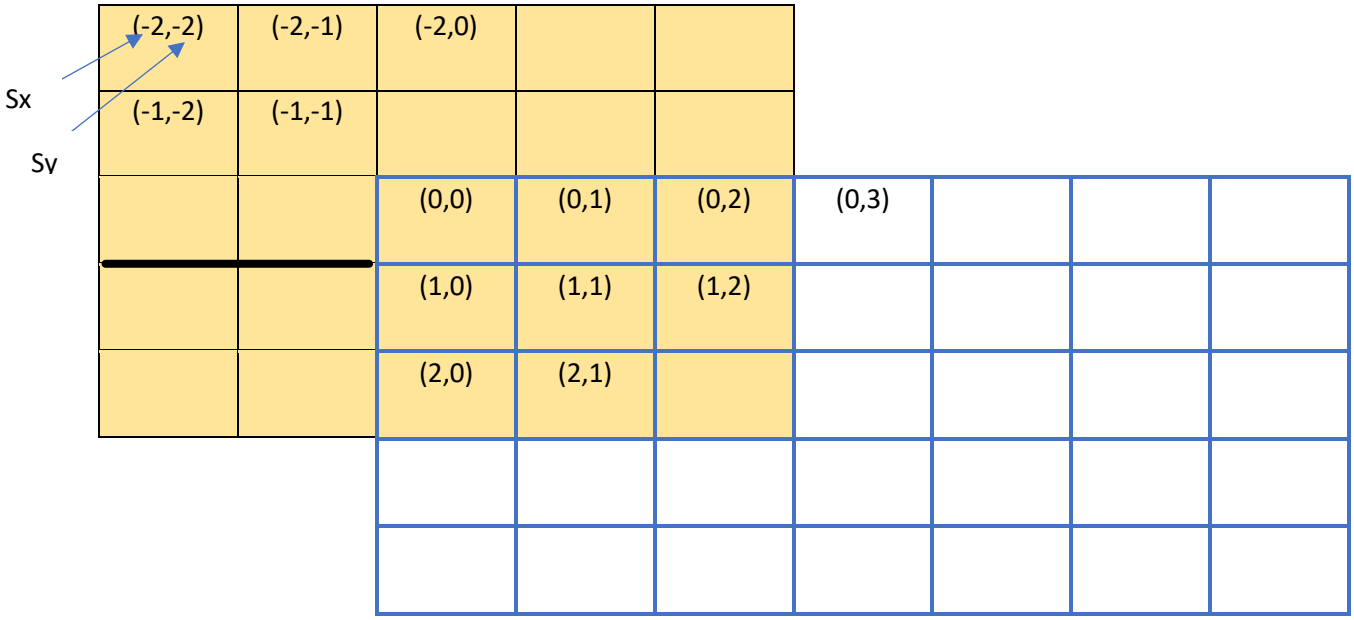
Funcția **corelatie** calculează gradul de potrivire dintre un șablon grayscale și o fereastră de dimensiuni egale dintr-o imagine grayscale și îl returnează. Primele 3 argumente reprezintă pointerul la tabloul de structuri RGB a șablonului și lățimea, respectiv înălțimea ei, iar celelalte 3 ale imaginii. Ultimele două valori reprezintă punctul din imagine de coordonate (Ix,Iy) în care se centrează șablonul.

Să spunem că aceștia ar fi pixelii imaginii. Fiecare cu coordonatele lui, care sunt indicii liniei și ai coloanei. Șablonul trebuie centrat pe rând în fiecare punct și calculată corelația.

(0,0)	(0,1)	(0,2)	(0,3)						
(1,0)	(1,1)	(1,2)							
									(10,9)

Când centrăm șablonul în punctele aflate în apropierea marginii din stânga sau a celei de sus, o parte a șablonului suprapus ar fi înafara imaginii noastre. Așa că determinăm valorile colțului din stânga sus care ar fi înafara imaginii în variabilele Sx, respectiv Sy.

Presupunem că șablonul ar avea dimensiunile de 5x5 și îl centrăm prima dată în (0,0). Așa că valorile colțului din stânga sus al șablonului ar fi: (-2,-2). Imaginea pe care vom plasa șablonul este tabelul cu marginile albastre iar șablonul suprapus în (0,0) ar fi cel desenat.



Astfel voi crea un vector alocat dinamic care va reprezenta imaginea de sub șablon. Pentru pozițiile care se află în imaginea inițială vom copia valoarea intensității, iar pentru celelalte vom pune valoarea 0. Acum că am creat imaginea de sub șablon aflăm corelația dintre ea și șablonul respectiv folosindu-ne de formula:

$$corr(S, f_I) = \frac{1}{n} \sum_{(i,j) \in S} \frac{1}{\sigma_{f_I} \sigma_S} \left(f_I(i,j) - \bar{f_I}\right) \left(S(i,j) - \bar{S}\right)$$

```
ferestre_sablon template_matching(char *cale_imagine,char *cale_sablon,double ps)
,
for(i=0; i<Height; i++)
    for(j=0; j<Width; j++)
    {
        rez=corelatie(VectorS,WidthS,HeightS,VectorI,Width,Height,i,j);
        if(rez>=ps)
        {
            aux=(fereastra *) malloc((A.NumarDetectii)*sizeof(fereastra));
            if(aux==NULL)
            {
                printf("Eroare la alocarea memoriei dinamice in functia de template matching.\n");
                ferestre_sablon eroare;
                eroare.NumarDetectii=-1;
                return eroare;
            }
            for(k=0; k<A.NumarDetectii; k++)
            {
                (aux+k)->X1=(A.V+k)->X1;
                (aux+k)->Y1=(A.V+k)->Y1;
                (aux+k)->X2=(A.V+k)->X2;
                (aux+k)->Y2=(A.V+k)->Y2;
                (aux+k)->valoare_corelatie=(A.V+k)->valoare_corelatie;
            }
            free(A.V);
            A.NumarDetectii++;
            A.V=(fereastra *)malloc(A.NumarDetectii*sizeof(fereastra));
            if(A.V==NULL)
            {
                printf("Eroare la alocarea memoriei dinamice in functia de template matching.\n");
                ferestre_sablon eroare;
                eroare.NumarDetectii=-1;
                return eroare;
            }
            for(k=0; k<A.NumarDetectii-1; k++)
            {
                (A.V+k)->X1=(aux+k)->X1;
                (A.V+k)->Y1=(aux+k)->Y1;
                (A.V+k)->X2=(aux+k)->X2;
                (A.V+k)->Y2=(aux+k)->Y2;
                (A.V+k)->valoare_corelatie=(aux+k)->valoare_corelatie;
```

Asta este partea mai interesantă a funcției de template_matching care primește calea unei imagini grayscale și a unui șablon, împreună cu un grad de corelatie minim și returnează o variabilă de tip ferestre_sablon ce conține numărul de detecții și un poiner la un tablou de ferestre din imagine ce corespund șablonului. Variabila A este cea pe care o vom returna la final. Astfel, începem prin a centra șablonul în fiecare punct din imagine și să calculăm corelația. Dacă aceasta este mai mare sau egală decât pragul stabilit ne folosim de un tablou alocat dinamic cu dimensiunea actuală a tabloului A (aux) în care copiem ferestrele deja găsite, după care eliberăm memoria ocupată de tabloul dinamic din A.V și îl alocăm cu 1 spațiu pentru o fereastră în plus. Urmează să copiem din tabloul aflat la adresa pointerului aux peste tabloul dinamic din A.V și fereastra tocmai găsită, după care eliberăm memoria folosită de tabloul de la adresa pointerului aux.

```
void creare_tablou(ferestre_sablon *D,ferestre_sablon *Template,int cifra)
```

```
for(i=0; i<D->NumarDetectii; i++)
{
    (aux+i)->valoare_corelatie=(D->V+i)->valoare_corelatie;
    (aux+i)->X1=(D->V+i)->X1;
    (aux+i)->Y1=(D->V+i)->Y1;
    (aux+i)->Y2=(D->V+i)->Y2;
    (aux+i)->X2=(D->V+i)->X2;
    (aux+i)->cifra_det=(D->V+i)->cifra_det;

}
D->NumarDetectii=(D->NumarDetectii) + (Template->NumarDetectii);
free(D->V);
D->V=(fereastra *)malloc(D->NumarDetectii * sizeof(fereastra));
if(D->V==NULL)
{
    printf("Eroare la alocarea memoriei dinamice in functia de creare tablou cu toate detectiile.\n");
    return;
}
for(i=0; i<((D->NumarDetectii) - (Template->NumarDetectii)); i++)
{
    (D->V+i)->valoare_corelatie=(aux+i)->valoare_corelatie;
    (D->V+i)->X1=(aux+i)->X1;
    (D->V+i)->Y1=(aux+i)->Y1;
    (D->V+i)->Y2=(aux+i)->Y2;
    (D->V+i)->X2=(aux+i)->X2;
    (D->V+i)->cifra_det=(aux+i)->cifra_det;
}
free(aux);
for(i=(D->NumarDetectii) - (Template->NumarDetectii); i<D->NumarDetectii; i++)
{
    (D->V+i)->valoare_corelatie=(Template->V+i-(D->NumarDetectii - Template->NumarDetectii))->valoare_corelatie;
    (D->V+i)->X1=(Template->V+i-(D->NumarDetectii - Template->NumarDetectii))->X1;
    (D->V+i)->Y1=(Template->V+i-(D->NumarDetectii - Template->NumarDetectii))->Y1;
    (D->V+i)->Y2=(Template->V+i-(D->NumarDetectii - Template->NumarDetectii))->Y2;
    (D->V+i)->X2=(Template->V+i-(D->NumarDetectii - Template->NumarDetectii))->X2;
    (D->V+i)->cifra_det=cifra;
}
free(Template->V);
```

Aceasta este funcția ce creează un tablou cu toate ferestrele detectate după operația de template_matching cu toate șabloanele. Primește doi pointeri la variabile de tip ferestre_sablon, primul fiind cel în care se va afla rezultatul final, iar al doilea conținând ferestrele detectate cu șablonul ce reprezintă valoarea din variabila cifra (asta va ajuta la desenarea fiecărei cifre cu o anumită culoare). Doar copiem conținutul din tabloul alocat dinamic la adresa de la pointerul Template.V în continuare celui de la adresa pointerului D.V, asigurându-mă ca înainte să măresc memoria celui de al doilea ca să încapă noile ferestre ce trebuie adăugate.

```
int cmp(const void *a,const void *b)
{
    double rez=((fereastra *)b)->valoare_corelatie - ((fereastra *)a)->valoare_corelatie;
    if(rez<0)
        return -1;
    else
    {
        if(rez==0)
            return 0;
        else
            return 1;
    }
}

double suprapunere(fereastra *a,fereastra *b)
{
    int i,j,A_intersectie=0,A_a,A_b;
    for(i=a->Y1; i<=a->Y2; i++)
        for(j=a->X1; j<=a->X2; j++)
            if(i>=b->Y1 && i<=b->Y2 && j>=b->X1 && j<=b->X2)
                A_intersectie++;
    A_a=(mod((a->Y2) - (a->Y1)) + 1)*(mod((a->X2) - (a->X1)) + 1);
    A_b=(mod((b->Y2) - (b->Y1)) + 1)*(mod((b->X2) - (b->X1)) + 1);
    double rez;
    rez=((double)A_intersectie)/(double)(A_a  + A_b - A_intersectie);
    return rez;
}
```

Funcția **cmp** este folosită pentru a sorta tabloul de la adresa pointerului D.V, ce are toate ferestrele, descrescător în funcție de valoarea de corelație.

Funcția **suprapunere** este folosită pentru a returna gradul de suprapunere dintre două ferestre date.

```
void desenare(char *cale_imagine,fereastra A,RGB c)
{
    int i,j;
    for(i=A.Y1; i<=A.Y2; i++)
    {
        j=A.X1;
        if(i>=0 && i<Height && j>=0 && j<Width)
        {
            (V+i*Width+j)->R=c.R;
            (V+i*Width+j)->G=c.G;
            (V+i*Width+j)->B=c.B;
        }
        j=A.X2;
        if(i>=0 && i<Height && j>=0 && j<Width)
        {
            (V+i*Width+j)->R=c.R;
            (V+i*Width+j)->G=c.G;
            (V+i*Width+j)->B=c.B;
        }
    }
    for(j=A.X1; j<=A.X2; j++)
    {
        i=A.Y1;
        if(i>=0 && i<Height && j>=0 && j<Width)
        {
            (V+i*Width+j)->R=c.R;
            (V+i*Width+j)->G=c.G;
            (V+i*Width+j)->B=c.B;
        }
        i=A.Y2;
        if(i>=0 && i<Height && j>=0 && j<Width)
        {
            (V+i*Width+j)->R=c.R;
            (V+i*Width+j)->G=c.G;
            (V+i*Width+j)->B=c.B;
        }
    }
    SalvareExternFormaLiniarizata(cale_imagine,cale_imagine,V);
    free(V);
}
```

Funcția de **desenare** a conturului unei ferestre primește imaginea în care să deseneze, o variabilă ce conține informațiile despre variabila curentă și o culoare cu care să deseneze conturul care e o variabilă de tip RGB. Aceasta modifică imaginea și apoi scrie înapoi imaginea la aceeași cale cu fereastra desenată.

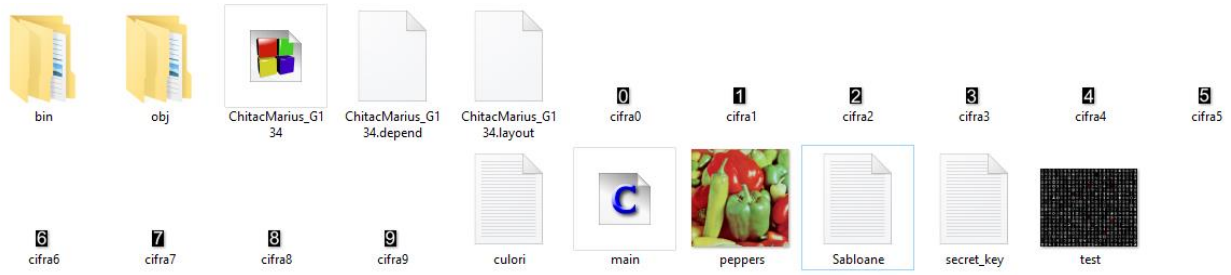
```
void eliminare_nonmaxime(ferestre_sablon *D)
{
    double prag=0.2;
    int i,j;
    qsort(D->V,D->NumarDetectii,sizeof(fereastra),cmp);
    for(i=0; i<(D->NumarDetectii) - 1; i++)
        for(j=i+1; j<D->NumarDetectii; j++)
            if((D->V+i)->valoare_corelatie!=-2 && suprapunere((D->V+i),(D->V+j))>prag)
                (D->V+j)->valoare_corelatie=-2;
}
```

```
void eliminare_nonmaxime(ferestre_sablon* D)
```

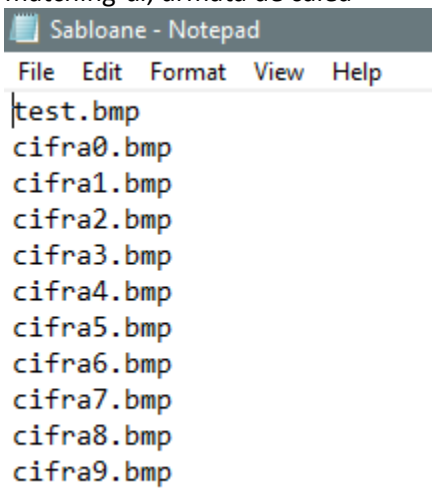
Funcția **de eliminare a detecțiilor non-maxime** sortează descrescător tabloul ce conține toate ferestrele după care se procesează tabloul D sortat de la stânga (detecții cu scor foarte mare) la dreapta (detecții cu scor foarte mic) astfel: toate detecțiile dj care se suprapun spațial cu detecția curentă di, cu i < j și deci și scor(di) > scor(dj) se elimină, marcându-le având valoarea corelației -2. Aceasta fiind între -1 și 1, oricare ar fi gradul minim de corelație dat.

```
printf("*****************************************************************************************************\n");
printf("1. Cripteaza o imagine color BMP si o salveaza in memorie externa.\n");
printf("2. Decripteaza o imagine color BMP criptata si o salveaza in memorie externa.\n");
printf("3. Afiseaza valorile testului chi-patrat pentru imaginea initiala si cea criptata.\n");
printf("4. Ruleaza operatia de template matching petru o imagine color BMP si o colectie de sabloane color BMP.\n");
printf("5. Ruleaza functia de eliminare a non-maximelor si deseneaza intr-o imagine copie a primei imagini pentru a o nu altera detectiile ramase cu o culoare specific
printf("6. Terminare program.\n");
scanf("%d",&opt);
switch(opt)
```

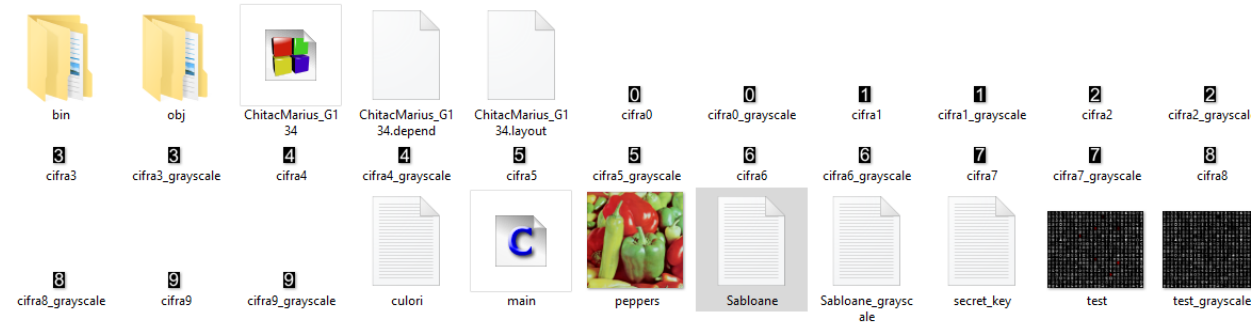
În main am scris un meniu cu toate operațiile pe care le poate efectua programul. Pentru primele 3 operații căile imaginilor și a fișierului text se citesc de la tastatură, întâmpinate cu un mesaj. Pentru celelalte două citirea se face din fișiere. Așa arată folder-ul inițial.



Fișierul Sabloane.txt conține calea imaginii pe care trebuie aplicat template matching-ul, urmată de calea șabloanelor.



După ce se execută operația de template matching, adică a 4 opțiune, programul meu va crea imaginile grayscale corespunzătoare celor din fișier la adrese diferite, care vor fi scrise în alt fișier text, pentru a le folosi ulterior. Și așa arată folder-ul după ce se efectuează operația:



Operația de eliminare a non-maximelor se poate efectua doar după ce a fost rulată operația de template matching. Iar în urma acesteia în folder va apărea și imagine desenată.