Facultatea de Matematică și Informatică
București

Proiect la Programare Procedurală

Student:
Todirică Oana-Andreea

Modulul de criptare/decriptare

Pentru a reține mai ușor în memorie pixelii unei imagini BMP, am declarat o structură numită *Pixel* ce conține trei câmpuri, câte unul pentru fiecare culoare *r g b* a unui pixel:

```
typedef struct
{
   unsigned char r,g,b;
} Pixel;
```

Funcția *padding* calculează paddingul unei imagini având ca parametru lățimea acesteia:

```
int padding(int W)
{    int padd;
     if(W%4!=0)
        padd=4-(3*W)%4;
    else
        padd=0;
    return padd;
}
```

Funcția *citire_img_liniarizata* are ca parametru de intrare calea imaginii, iar ca parametrii de ieșire un vector de pixeli ce va conține imaginea liniarizată, un vector ce va conține header-ul imaginii, înalțimea și lățimea imaginii. Această funcție va citi imaginea și o va păstra răsturnată în memorie într-un vector de pixeli. Pentru a putea păstra imaginea răsturnată în memorie, am folosit următoarea formulă: *W*H-W*(i+1)+j* pentru a determina poziția fiecărui pixel în parte în vectorul liniarizat:

```
void citire_img_liniarizata(char* nume_fisier_sursa, Pixel **Vector,int **header, int *H, int *W)
{
    int dim;
    FILE *fin;
    fin = fopen(nume_fisier_sursa, "rb");

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

   fseek(fin, 2, SEEK_SET);
   fread(&dim, sizeof(unsigned int), 1, fin);

   fseek(fin, 18, SEEK_SET);
   fread(*(&W), sizeof(unsigned int), 1, fin);
   fread(*(&H), sizeof(unsigned int), 1, fin);

    int padd;
    padd=padding(*W);
```

```
*header=(int *)malloc(sizeof(int)*54);
if((*header)==NULL)
{
    printf("Nu s-a gasit suficienta memorie.");
    return ;
}

int i,x;
fseek(fin, 0, SEEK_SET);
for(i=0;i<54;i++)
{
    fread(&x,1,1,fin);
    (*header)[i]=x;
}


*Vector=(Pixel*)malloc(sizeof(Pixel)*dim/3);
if((*Vector)==NULL)
{
    printf("Nu s-a gasit suficienta memorie");
    return;
}
Pixel P;
int j;

for(i=0;i<*H;i++)
{
    for(j=0;j<*W;j++)
    {
        //citesc culorile pixelului
    char Pixeli[3];
    int poz;
    poz=(*W)*(*H)-(*W)*(i+1)+j;// liniarizam matricea rasturnata (prima linie o punem ultima in vector)
    fread(Pixeli,sizeof(char),3,fin);
    P.r = Pixeli[2];
    P.g = Pixeli[1];
    P.b = Pixeli[0];
    (*Vector)[poz]=P;
    }
    fseek(fin,padd,SEEK_CUR);//sarim peste padding
}

   fclose(fin);
}
```

Funcția *afisare_imagine* primește ca parametrii de ieșire calea fișierului de ieșire, adică imaginea pe care o vom crea, iar ca parametrii de intrare vectorul de pixeli în care am reținut imaginea liniarizată, un vector ce conține header-ul imaginii liniarizate, înălțimea și lățimea imaginii. Funcția afișează imaginea reținută în vectorul liniarizat într-un fișier binar, iar pentru a face acest lucru se folosește de aceeași formula precizată mai sus:

```
void afisare_imagine(char *nume_fisier_iesire, Pixel *Vector,int *header, int H, int W)
{
    FILE *fout;
    fout=fopen(nume_fisier_iesire,"wb");

    int i,j,padd;
    padd=padding(W);
    for(i=0;i<54;i++)
        fwrite(&header[i],1,1,fout);
    for(i=0;i<H;i++)
    {

        for(j=0;j<W;j++)
        {
        int poz;
        poz=W*H-W*(i+1)+j;
        fwrite(&Vector[poz].b,1,1,fout);
        fwrite(&Vector[poz].g,1,1,fout);
        fwrite(&Vector[poz].r,1,1,fout);
        }



        for(j=0;j<padd;j++)
        {
            int x=0;
            fwrite(&x,1,1,fout);//adaugam noile padding
        }

    }
    fclose(fout);
    free(header);
}
```

Funcția *XORSHIFT32* are ca parametru de ieșire un vector R, iar ca parametrii de intrare seed-ul *R0*, lățimea *W* și înălțimea *H* a imaginii. Această funcție generează *2*W*H-1* numere pseudoaleatoare ce ne vor folosi în funcțiile ce vor urma:

```
void XORSHIFT32 (unsigned int **R, unsigned int R0, int W, int H)
{
    unsigned int i,n;
    unsigned int r=R0;
    n=2*W*H-1;
    *R=(unsigned int*)malloc(n*sizeof(unsigned int));
    if((*R)==NULL)
    {
        printf("Nu s-a gasit suficienta memorie");
        return;
    }
    for(i=1;i<=n;i++)
    {
        r=r^r<<13;
        r=r^r>>17;
        r=r^r<<5;
        *(*R+i)=r;
    }

}
```

Funcția *permutare_aleatoare* are aceeași parametrii ca și funcția *XORSHIFT32* și se folosește de aceasta pentru a genera o permutare aleatoare:

```
void permutare_aleatoare( unsigned int **R, unsigned int R0, int W, int H)
{
    XORSHIFT32(R, R0, W,H);
}
```

Funcția *Durstenfeld* are ca parametru de ieșire vectorul *P*, iar ca parametrii de intrare vectorul *R* generat prin funcția *permutare_aleatoare*, lățimea *W* și înălțimea *H* a imaginii. Funcția generează o permutare aleatoare de dimensiune *W*H* în vectorul *P*, folosindu-se de algoritmul lui Durstenfeld, varianta modernă a algoritmului Fisher-Yates și de numerele pseudoaleatoare generate cu algoritmul XORSHIFT32:

```
void Durstenfeld(unsigned int **P, unsigned int *R, int W, int H)
{
    unsigned int n= W*H,r,aux;
    int i;
    *P=(unsigned int *)malloc(n*sizeof(unsigned int));
    if((*P)==NULL)
    {
        printf("Nu s-a gasit suficienta spatiu");
        return;
    }
    for(i=0;i<n;i++)
   {
       (*P)[i]=i;

   }
    for(i=n-1;i>=1;i--)
    {
        r=R[n-i]%(i+1);
        aux=(*P)[r];
        (*P)[r]=(*P)[i];
        (*P)[i]=aux;
    }
}
```

Funcția *imagine_intermediara* are ca parametru de ieșire vectorul de pixeli *Vector2*, iar ca parametrii de intrare vectorul de pixeli ce conține imaginea inițială liniarizată *Vector*, permutarea aleatoare reținută în vectorul *P*, lățimea *W* și înălțimea *H* a imaginii. Funcția permută pixelii imaginii liniarizate *Vector*, conform permutării *P*, obținându-se o imagine intermediară *Vector2*:

```
void imagine_intermediara(Pixel **Vector2, Pixel *Vector, unsigned int *P,int W, int H)
{
    (*Vector2)=(Pixel *)malloc(W*H*sizeof(Pixel));
    if((*Vector2)==NULL)
    {
        printf("Nu s-a gasit suficienta memorie");
        return;
    }

    int i;
    for(i=0;i<W*H;i++)
    {
        (*Vector2)[P[i]]=Vector[i];
    }
}
```

Funcția *criptare* are ca parametrii de intrare calea imaginii sursă **nume_img_sursa* și calea fișierul text ce conține cheia secretă **key*, iar ca parametrii de ieșire are calea fișierului de ieșire în care se va salva imaginea criptată **nume_img_iesire* și vectorul ce conține imaginea liniarizată criptată *C*. Pentru a cripta imaginea, se apelează funcțiile anterioare: *citire_img_liniarizata*, *permutare_aleatoare*, *Durstenfeld*, *imagine_intermediara* și apoi vom determina primul pixel din imaginea criptată aplicând xor între cheia secretă, primul element din imaginea intermediară și primul element din permutarea aleatoare. Vom continua să aflăm restul pixelilor aplicând xor între pixelul precedent celui curent, elementul curent din imaginea intermediară și elementul curent din permutarea aleatoare. Funcția va afișa imaginea criptată în fișierul binar:

```
void criptare(char *nume_img_sursa, char *nume_img_iesire, const char *key, Pixel **C)
{
    int i, H, W, *header;
    Pixel *Vector,*Vector2;//vector=imagine liniarizata, vector2=imaginea intermediara permutata;

    unsigned int R0,SV,*R, *P;//R= secventa de numere intregi aleatoare, P=permutarea aleatoare
    unsigned char *sv;
    unsigned char *r;

    FILE *fin=fopen(key,"r");
    if(fin==NULL)
    {
        printf("Nu am gasit cheia secreta");
        return;
    }

    fscanf(fin,"%u",&R0);
    fscanf(fin,"%u",&SV);

    citire_img_liniarizata(nume_img_sursa,&Vector,&header,&H,&W);
    permutare_aleatoare(&R,R0,W,H);

    Durstenfeld(&P,R,W,H);
    imagine_intermediara(&Vector2, Vector,P,W,H);
```

```
    sv=(unsigned char *)&SV;
    r=(unsigned char *)&R[W*H];

    (*C)=(Pixel *)malloc(W*H*sizeof(Pixel));

    (*C)[0].b=sv[0]^Vector2[0].b^r[0];
    (*C)[0].g=sv[1]^Vector2[0].g^r[1];
    (*C)[0].r=sv[0]^Vector2[0].r^r[2];
    for(i=1;i<W*H;i++)
    {   r=(unsigned char *)&R[W*H+i];
        (*C)[i].b=(*C)[i-1].b^Vector2[i].b^r[0];
        (*C)[i].g=(*C)[i-1].g^Vector2[i].g^r[1];
        (*C)[i].r=(*C)[i-1].r^Vector2[i].r^r[2];

    }

    afisare_imagine(nume_img_iesire,*C,header,H,W);
    free(Vector);
    free(Vector2);
    free(R);
    free(P);
    fclose(fin);
}
```

Funcția *inversa* calculează inversa permutării aleatoare generată cu ajutorul algoritmului XORSHIFT32 si Durstenfeld. Funcția are ca parametrii de intrare vectorul *P* ce conține permutarea aleatoare, lățimea *W* și înălțimea *H* a imaginii și ca parametrii de ieșire are vectorul *P1*, ce va conține permutarea inversă:

```
void inversa(unsigned int *P, unsigned int **P1, int W, int H )
{
    int n=W*H,i;
    (*P1)=(unsigned int *)malloc(n*sizeof(int));
    if((*P1)==NULL)
    {
        printf("Nu s-a gasit suficienta memorie");
        return;
    }
    for(i=0;i<n;i++)
        (*P1)[P[i]]=i;
}
```

Funcția *decriptare* are ca parametrii de intrare calea imaginii sursă **nume_img_sursa* și calea fișierul text ce conține cheia secretă **key*, iar ca parametri de ieșire are calea fișierului de ieșire în care se va salva imaginea decriptată **nume_img_iesire*. Pentru a decripta o imagine, trebuie apelată funcția criptare, apoi funcțiile anterioare: *citire_img_liniarizata, permutare_aleatoare, Durstenfeld, inversa*, iar apoi vom determina primul pixel din imaginea decriptată aplicând xor între cheia secretă, primul element din imaginea criptată și primul element din permutarea aleatoare. Vom continua să aflăm restul pixelilor aplicând xor între pixelul precedent celui curent din imaginea criptată, elementul curent din imaginea criptată și elementul curent din permutarea aleatoare. Se va apela funcția *imagine_intermediara* folosindu-se de permutarea inversă, iar apoi funcția va afișa imaginea decriptată în fișierul binar:

```
void decriptare(char *nume_img_sursa, char *nume_img_iesire, const char *key)
{
    int i, H, W, *header;
    Pixel *Vector,*Vector2, *C, *Cl, *D;//vector=imagine liniarizata, vector2=imaginea intermediara permutata,
                                // C=imaginea criptata, Cl=imagine criptata intermediara, D=imaginea decriptata;

    unsigned int R0,SV,*R, *P, *Pl;//R= secventa de numere intregi aleatoare, P=permutarea aleatoare,Pl=permutarea inversa
    unsigned char *sv;
    unsigned char *r;
    char img_decriptat[]="peppersdecriptat.bmp";
    printf("numele imaginii decriptate= ");
    scanf("%s",img_decriptat);
    printf("\n");

    criptare(nume_img_sursa,nume_img_iesire,key, &C);

    FILE *fin=fopen(key,"r");
    if(fin==NULL)
    {
        printf("Nu am gasit cheia secreta!!!  ");
        perror(key);
        return;
    }
    fscanf(fin,"%u",&R0);
    fscanf(fin,"%u",&SV);

    citire_img_liniarizata(nume_img_sursa,&Vector,&header,&H,&W);

    permutare_aleatoare(&R,R0,W,H);
    Durstenfeld(&P,R,W,H);
    inversa (P,&Pl,W,H);

    sv=(unsigned char *)&SV;
    r=(unsigned char *)&R[W*H];

    Cl=(Pixel *)malloc(W*H*sizeof(Pixel));
    if(Cl==NULL)
    {
        printf("Nu s-a gasit suficienta memorie");
        return;
    }

    Cl[0].b=sv[0]^C[0].b^r[0];
    Cl[0].g=sv[1]^C[0].g^r[1];
    Cl[0].r=sv[0]^C[0].r^r[2];
    for(i=1;i<W*H;i++)
    {   r=(unsigned char *)&R[W*H+i];
        Cl[i].b=C[i-1].b^C[i].b^r[0];
        Cl[i].g=C[i-1].g^C[i].g^r[1];
        Cl[i].r=C[i-1].r^C[i].r^r[2];
    }

    imagine_intermediara(&D,Cl,Pl,W,H);

    afisare_imagine(img_decriptat,D,header,H,W);

    free(Vector);
    free(Vector2);
    free(C);
    free(R);
    free(P);
    free(Pl);
    free(D);
    free(Cl);
    fclose(fin);
}
```

Funcția *chipatrat* are ca parametru de intrare calea unei imagini. Funcția calculează valorile testului chi pătrat pentru fiecare canal de culoare al imaginii și le afișează. Pentru a se obține aceste valori, se calculează frecvența valorii i (0<=i<=255) pentru fiecare canal de culoare al imaginii, valorile reținându-se în 3 vectori de frecvență numiți *fr*, *fb*, *fg* și frecvența estimată teoretic a oricărei valori i, notată cu *f* și calculată ca produsul dimensiunilor imaginii impărțit la 256:

```
void chipatrat( char *imagine)
{
    double r,g,b,f;
    r=g=b=0.0;
    int *fr,*fg,*fb,i;
    fr=(int *)calloc(256,sizeof(int));
    fg=(int *)calloc(256,sizeof(int));
    fb=(int *)calloc(256,sizeof(int));
    if(fr==NULL||fg==NULL||fb==NULL)
    {
        printf("Nu s-a gasit suficienta memorie");
        return;
    }

    int W,H;
    FILE *fin;
    fin = fopen(imagine, "rb");

    if(fin == NULL)
    {
        printf("nu am gasit imaginea sursa din care citesc");
        return;
    }
   fseek(fin, 18, SEEK_SET);
   fread(&H, sizeof(unsigned int), 1, fin);
   fread(&W, sizeof(unsigned int), 1, fin);

    int padd;
    padd=padding(W);

    int x;
    fseek(fin, 0, SEEK_SET);
    for(i=0;i<54;i++)
        fread(&x,1,1,fin);

    Pixel P;
    int j;

    f=(float)H*W/256.0;
    for(i=0;i<H;i++)
    {
        for(j=0;j<W;j++)
        {

        char Pixeli[3];
        fread(Pixeli,sizeof(char),3,fin);
        P.r = Pixeli[2];
        P.g = Pixeli[1];
        P.b = Pixeli[0];
        fr[P.r]++;
        fg[P.g]++;
        fb[P.b]++;

        }
        fseek(fin,padd,SEEK_CUR);
    }
```

```
    for(i=0;i<256;i++)
      {
        r=r+(fr[i]-f)*(fr[i]-f)/f;
        g=g+(fg[i]-f)*(fg[i]-f)/f;
        b=b+(fb[i]-f)*(fb[i]-f)/f;
      }

    printf("\nChi-squared test on RGB channels for: %s \nRosu: %.2lf   Verde:%.2lf  Albastru: %.2lf\n",imagine,r,g,b);
    fclose(fin);
}
```

Programul principal citește numele fișierelor de la tastatură și apelează funcțiile *criptare*, *decriptare* și *chipatrat*:

```
int main()
{

    char nume_img_sursa[] = "peppers.bmp";

    printf("numele imaginii sursa = ");
    scanf("%s",nume_img_sursa);
    printf("\n");

    char nume_img_iesire[] = "pepperscriptat.bmp";

    printf("numele imaginii criptate = ");
    scanf("%s",nume_img_iesire);
    printf("\n");

    const char key[]="secret_key.txt";

    printf("numele fisierului ce contine cheia secreta = ");
    scanf("%s",key);
    printf("\n");

    decriptare(nume_img_sursa,nume_img_iesire,key);

    chipatrat(nume_img_sursa);
    chipatrat(nume_img_iesire);

    return 0;
}
```

Modulul de recunoaștere de pattern-uri într-o imagine

Pentru a reține mai ușor în memorie pixelii unei imagini BMP, am declarat o structură numită *Pixel* ce conține trei câmpuri, câte unul pentru fiecare culoare: *r g b* a unui pixel:

```
typedef struct
{
   unsigned char r,g,b;
} Pixel;
```

Pentru a putea reține mai ușor informații despre o fereastră într-o imagine, am declarat structura numită *Fereastra*, ce conține câmpurile: *cor* (corelația unei ferestre), *lin, col* (indicii la care se afla fereastra în matricea imaginii), și *C* (o variabilă de tip *Pixel*, care indică cu ce culoare ar trebui colorată fereastra respectivă):

```
typedef struct
{
    double cor;
    int lin,col;
    Pixel C;
}Fereastra;
```

Funcția *padding* calculează paddingul unei imagini având ca parametru lățimea acesteia:

```
int padding(int W)
{   int padd;
     if(W%4!=0)
        padd=4-(3*W)%4;
    else
        padd=0;
    return padd;
}
```

Funcția grayscale_image are ca parametru de intrare calea imaginii sursă **nume_img_sursa*, iar ca parametrii de ieșire calea fișierului de ieșire în care se va salva imaginea grayscale **nume_img_destinatie*, înălțimea și lățimea imaginii. Funcția transformă o imagine oarecare într-o imagine grayscale:

```
void grayscale_image(char* nume_fisier_sursa,char* nume_fisier_destinatie, int *inaltime_img, int *latime_img)
{
   FILE *fin, *fout;
   unsigned int dim_img;
   unsigned char pRGB[3], header[54], 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);
   fread(&dim_img, sizeof(unsigned int), 1, fin);

   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);
    }
    fclose(fin);

   //calculam padding-ul pentru o linie
   int padd=padding(latime_img);

   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,padd,SEEK_CUR);
   }
   fclose(fout);
}
```

Următoarele două funcții ne vor ajuta în calcularea corelației unei ferestre.

Funcția *mediapixelifereastra* are ca parametrii de intrare o matrice *a* în care este reținuta imaginea, linia și coloana la care se găsește fereastra pentru care calculăm media de pixeli și lățimea și înălțimea ferestrei, fiind aceleași cu dimensiunile șablonului. Funcția returnează, după cum spune și denumirea acesteia, media valorilor pixelilor din fereastră. Am evitat să reținem 3 valori pentru fiecare pixel, deoarece, imaginea fiind grayscale are valori egale pentru cele 3 canale de culoare:

```
double mediapixelifereastra(int **a, int l, int c, int w, int h)
{
    int i,j, S=0,n=w*h;

    for(i=l;i<l+h;i++)
        for(j=c;j<c+w;j++)
            S=S+a[i][j];

    double medie;
    medie=(double)S/(double)n;
    return medie;

}
```

Funcția *deviatia_standard_fereastra* are ca parametrii de intrare o matrice *a* în care este reținută imaginea, linia și coloana la care se găsește fereastra, media pixelilor din fereastră, ce se va calcula cu ajutorul funcției anterioare și lățimea și înălțimea ferestrei, fiind aceleași cu dimensiunile șablonului. Funcția va returna deviația standard a valorilor intensităților grayscale a pixelilor în fereastră:

```
double deviatia_standard_fereastra(int **a,int l, int c, double fmed, int w,int h)
{
    int i,j, n=w*h;

    double fdev=0.0;

     for(i=l;i<l+h;i++)
        for(j=c;j<c+w;j++)
            fdev=fdev+((double)a[i][j]-fmed)*((double)a[i][j]-fmed);
    fdev=fdev/(double)(n-1);
    return sqrt(fdev);

}
```

Funcția *citire* are ca parametrii de intrare lățimea și înălțimea imaginii și calea imaginii, iar ca parametru de ieșire are o matrice în care va fi reținută imaginea. Am evitat să reținem 3 valori pentru fiecare pixel, deoarece, imaginea fiind grayscale are valori egale pentru cele 3 canale de culoare:

```
void citire(int ***a, int w, int h, char *imagine)
{
    FILE *fin=fopen(imagine,"rb");

    int padd=padding(w),x,i,j;

    if(fin==NULL)
    {
        printf("nu am gasit imaginea");
        return;
    }

   for(i=0;i<54;i++)
        fread(&x,1,1,fin);

    *a=(int **)malloc(h*sizeof(int*));

    if(*a==NULL)
       {
           printf("Nu s-a gasit suficienta memorie");
           return;
       }

    for(i=h-1;i>=0;i--)
    {
        (*a)[i]=(int *)malloc(w*sizeof(int));
        if((*a)[i]==NULL)
        {
            printf("Nu s-a gasit suficienta memorie");
            return;
        }

        for(j=0;j<w;j++)
        {
            unsigned char Pixeli[3];
            fread(Pixeli,sizeof(unsigned char),3,fin);
            (*a)[i][j]=Pixeli[0];

        }

        fseek(fin,padd,SEEK_CUR);
    }
    fclose(fin);

}
```

Funcția *template_matching* are ca parametrii de intrare calea imaginii, calea unui șablon, o valoare *ps* reprezentând pragul corelației și culoarea cu care trebuie colorată fereastra, iar ca parametrii de ieșire are calea imaginii transformate în grayscale, calea șablonului transformat în grayscale, un vector ce va conține ferestrele cu o corelație mai mare decât pragul setat și numărul ferestrelor din vector. Funcția începe prin a transforma în grayscale imaginea și șablonul, apoi salvează imaginea și șablonul grayscale în matrice. Calculează media pixelilor șablonului dat, cât și deviația standard a valorilor intensităților grayscale a pixelilor în șablon, apoi calculează corelația pentru fiecare fereastră din matricea imaginii, folosindu-se de funcțiile anterioare: *mediapixelifereastra* și *deviatia_standard_fereastra* și salvează în vectorul de ferestre doar pe acelea ce au corelația mai mare decât pragul setat:

```
void template_matching(char *imagine, char *imagine_gray, char *sablon, char *sablon_gray, double ps, Fereastra **F ,int *nr, Pi
{
    int W,H,w,h,i,j,p,q ;

    grayscale_image(imagine,imagine_gray, &H, &W);
    grayscale_image(sablon,sablon_gray,&h,&w);

    int n=w*h;
    int **img, **sab;

    citire(&img,W,H,imagine_gray);
    citire(&sab,w,h,sablon_gray);

    double fmed,Smed, Sdev=0.0, fdev,cor;
    double S=0.0;

    for(i=0;i<h;i++)
           for(j=0;j<w;j++)
            S=S+(double)sab[i][j];

    Smed=S/(double)n;

    for(i=0;i<h;i++)
        for(j=0;j<w;j++)
            Sdev=Sdev+((double)sab[i][j]-Smed)*((double)sab[i][j]-Smed);

    Sdev=Sdev/(double)(n-1);
    Sdev=sqrt(Sdev);
    *F=NULL;
    Fereastra *aux;
    *nr=0;
    for(i=0;i<H-h+1;i++)
         for(j=0;j<W-w+1;j++)
         {
           fmed=mediapixelifereastra(img,i,j,w,h);
           fdev=deviatia_standard_fereastra(img,i,j,fmed,w,h);
           cor=0.0;

           for(p=0;p<h;p++)
             for(q=0;q<w;q++)
                {
                     int l,c;
                     l=i+p;
                     c=j+q;
                     cor=cor+((img[l][c]-fmed)*(sab[p][q]-Smed))/(Sdev*fdev);
                }
             cor=cor/n;
             if(cor>ps)
             {   ++(*nr);
                 aux=(Fereastra*)realloc((*F),(*nr)*sizeof(Fereastra));
                 if(aux==NULL)
                 {
                     printf("Nu exista suficienta memorie");
                     return;
                 }
```

```
                .
                else
                {
                    (*F)=aux;
                    (*F)[(*nr)-1].lin=i;
                    (*F)[(*nr)-1].col=j;
                    (*F)[(*nr)-1].cor=cor;
                    (*F)[(*nr)-1].C=cul;

                }
            }

        }

        for(i=0;i<H;i++)
            free(img[i]);
        free(img);
        for(i=0;i<h;i++)
            free(sab[i]);
        free(sab);

}
```

Funcția *citire_dimensiuni* are ca parametru de intrare calea unei imagini, iar ca parametrii de ieșire dimensiunile ei, lățimea și înălțimea:

```
void citire_dimensiuni( char *imagine,int *w, int *h)
{
    int i;

    FILE *fin=fopen(imagine,"rb");
    if(fin==NULL)
    {
        printf("nu am gasit imaginea");
        return;
    }
    fseek(fin, 18, SEEK_SET);
   fread(w, sizeof(unsigned int), 1, fin);
   fread(h, sizeof(unsigned int), 1, fin);
   fclose(fin);


}
```

Funcția *citire_matrice* are ca parametri de intrare calea unei imagini, dimensiunile sale (lățimea și înălțimea), iar ca parametrii de ieșire o matrice de pixeli și header-ul imaginii. Funcția salvează o imagine în memorie sub forma unei matrice de pixeli, pentru a putea colora mai ușor ferestrele:

```
void citire_matrice(char *imagine, int w, int h, Pixel ***a, unsigned char **header)
{
    unsigned char x;
    int i,j;

    FILE *fin=fopen(imagine,"rb");
    if(fin==NULL)
    {
        printf("nu am gasit imaginea ");
        return;
    }

    *header=(unsigned char *)malloc(sizeof(unsigned char)*54);
    if(*header==NULL)
    {
        printf("Nu s-a gasit suficienta memorie.");
        return ;
    }

    for(i=0;i<54;i++)
        {
        fread(&x,1,1,fin);
        (*header)[i]=x;
        }

    int padd=padding(w);

    *a=(Pixel **)malloc(h*sizeof(Pixel*));

    if(*a==NULL)
        {
            printf("Nu s-a gasit suficienta memorie");
            return;
        }

    for(i=h-1;i>=0;i--)
    {
        (*a)[i]=(Pixel *)malloc(w*sizeof(Pixel));

        if((*a)[i]==NULL)
        {
            printf("Nu s-a gasit suficienta memorie");
            return;
        }

        for(j=0;j<w;j++)
        {
            unsigned char Pixeli[3];
            fread(Pixeli,sizeof(unsigned char),3,fin);
            (*a)[i][j].b=Pixeli[0];
            (*a)[i][j].g=Pixeli[1];
            (*a)[i][j].r=Pixeli[2];
        }

        fseek(fin,padd,SEEK_CUR);
    }
    fclose(fin);
}
```

Funcția *colorare* are ca parametri de intrare o fereastră și dimensiunile acesteia și calea unei imagini ce este atât parametru de intrare, cât și de ieșire. Funcția citește dimensiunile imaginii, apoi o salvează în memorie sub forma unei matrice de pixeli și colorează marginile ferestrei date cu o anumită culoare, folosindu-se de matricea de pixeli. La final matricea este tranformată din nou într-o imagine:

```
void colorare(char *imagine, Fereastra F, int lat, int lung)
{
    int w,h,i,j;
    Pixel **a;
    unsigned char *header;

    citire_dimensiuni(imagine,&w,&h);
    citire_matrice(imagine,w,h,&a,&header);


   FILE *fout=fopen(imagine,"wb");
    if(fout==NULL)
    {
        printf("nu am gasit imaginea");
        return;
    }

    for(j=F.col;j<F.col+lat;j++)

     {
        a[F.lin][j].r=F.C.r;
        a[F.lin][j].g=F.C.g;
        a[F.lin][j].b=F.C.b;
        a[F.lin+lung-1][j].r=F.C.r;
        a[F.lin+lung-1][j].g=F.C.g;
        a[F.lin+lung-1][j].b=F.C.b;
     }
    for(i=F.lin;i<F.lin+lung;i++)
     {  a[i][F.col].r=F.C.r;
        a[i][F.col].g=F.C.g;
        a[i][F.col].b=F.C.b;
        a[i][F.col+lat-1].r=F.C.r;
        a[i][F.col+lat-1].g=F.C.g;
        a[i][F.col+lat-1].b=F.C.b;
     }

     for(i=0;i<54;i++)
        fwrite(&header[i],1,1,fout);

    for(i=h-1;i>=0;i--)
     {
         for(j=0;j<w;j++)
        {

        fwrite(&a[i][j].b,1,1,fout);
        fwrite(&a[i][j].g,1,1,fout);
        fwrite(&a[i][j].r,1,1,fout);
        }
         for(j=0;j<padd;j++)
        {
            int x=0;
            fwrite(&x,1,1,fout);//sărim peste padding
        }

     }

    for(i=0;i<h;i++)
        free(a[i]);
    free(a);

    fclose(fout);
}
```

Funcția *detectii* are ca parametrii de ieșire un vector de ferestre și numărul ferestrelor din vector și calea imaginii grayscale, iar ca parametrii de intrare are calea unei imagini. Funcția citește șabloanele dintr-un fișier și aplică funcția de *template_matching* pentru fiecare șablon în parte, adăugând detecțiile fiecăruia în vectorul *D*:

```
void detectii(Fereastra **D, int *n, char *imagine, char *imagine_gray)
{
    *D=NULL;
    *n=0;
    Fereastra *aux, *F;
    Pixel cul;
    FILE *fin=fopen("nume_sabloane.txt","r");
    if(fin==NULL)
    {
        printf("Eroare la deschiderea fisierului");
        return;
    }
    int i,j;
    double ps=0.5;
    char sablon[]="cifra0.bmp";
    char sablon_gray[]="cifra0_gray.bmp";
    for(i=0;i<10;i++)
    {
        if(i==0)
        {
            strcpy(sablon,"cifra0.bmp");
            fscanf(fin,"%s",sablon);
            strcpy(sablon_gray,"cifra0_gray.bmp");
            cul.r=255;
            cul.g=0;
            cul.b=0;
        }
        if(i==1)
        {
            strcpy(sablon,"cifra1.bmp");
            fscanf(fin,"%s",sablon);
            strcpy(sablon_gray,"cifra1_gray.bmp");
            cul.r=255;
            cul.g=255;
            cul.b=0;
        }
        if(i==2)
        {
            strcpy(sablon,"cifra2.bmp");
            fscanf(fin,"%s",sablon);
            strcpy(sablon_gray,"cifra2_gray.bmp");
            cul.r=0;
            cul.g=255;
            cul.b=0;
        }
        if(i==3)
        {
            strcpy(sablon,"cifra3.bmp");
            fscanf(fin,"%s",sablon);
            strcpy(sablon_gray,"cifra3_gray.bmp");
            cul.r=0;
            cul.g=255;
            cul.b=255;
        }
```

```
if(i==4)
{
     strcpy(sablon,"cifra4.bmp");
     fscanf(fin,"%s",sablon);
     strcpy(sablon_gray,"cifra4_gray.bmp");
     cul.r=255;
     cul.g=0;
     cul.b=255;
}
if(i==5)
{
     strcpy(sablon,"cifra5.bmp");
     fscanf(fin,"%s",sablon);
     strcpy(sablon_gray,"cifra5_gray.bmp");
     cul.r=0;
     cul.g=0;
     cul.b=255;
}
if(i==6)
{
     strcpy(sablon,"cifra6.bmp");
     fscanf(fin,"%s",sablon);
     strcpy(sablon_gray,"cifra6_gray.bmp");
     cul.r=192;
     cul.g=192;
     cul.b=192;
}
if(i==7)
{
    strcpy(sablon,"cifra7.bmp");
    fscanf(fin,"%s",sablon);
    strcpy(sablon_gray,"cifra7_gray.bmp");
    cul.r=255;
    cul.g=140;
    cul.b=0;
}
if(i==8)
{
    strcpy(sablon,"cifra8.bmp");
    fscanf(fin,"%s",sablon);
    strcpy(sablon_gray,"cifra8_gray.bmp");
    cul.r=128;
    cul.g=0;
    cul.b=128;
}
if(i==9)
{
    strcpy(sablon,"cifra9.bmp");
    fscanf(fin,"%s",sablon);
    strcpy(sablon_gray,"cifra9_gray.bmp");
    cul.r=128;
    cul.g=0;
    cul.b=0;
}
```

```
        int nr;
        template_matching(imagine,imagine_gray,sablon,sablon_gray,ps,&F,&nr,cul);

        aux=(Fereastra*)realloc((*D),((*n)+nr)*sizeof(Fereastra));
        if(aux==NULL)
        {
            printf("Nu exista suficienta memorie");
            return;
        }
        else
        {
            (*D)=aux;
            for(j=0;j<nr;j++)
                (*D)[(*n)+j]=F[j];
            (*n)=(*n)+nr;
        }

    free(F);

    }
    fclose(fin);

}
```

Funcția *cmp* are ca parametrii două constante void. Această funcție ne va ajuta pentru a sorta vectorul de ferestre în ordine descrescătoare a corelațiilor, folosind algoritmul *qsort*:

```
int cmp( const void *a, const void *b)
{
    Fereastra va=*(Fereastra *)a;
    Fereastra vb=*(Fereastra *)b;
    if(va.cor>vb.cor)
        return -1;
    else
        if(va.cor<vb.cor)
        return 1;
    else
        return 0;

}
```

Funcția *sortare* are ca parametrii de intrare vectorul de ferestre și numărul de elemente ale acestuia. Funcția sortează elementele vectorului în ordine descrescătoare a corelației ferestrelor:

```
void sortare(Fereastra *D, int n)
{
    qsort(D,n,sizeof(Fereastra),cmp);
}
```

Funcțiile *minim* și *maxim* au ca parametrii două numere și îl returnează pe cel mai mic, respectiv cel mai mare. Aceste funcții ne vor folosi pentru a calcula aria intersecției a două ferestre:

```
int minim(int a, int b)
{
    if(a<b)
        return a;
    else
        return b;
}


int maxim(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

Funcția *arie_intersectie* primește ca parametrii de intrare două ferestre și dimensiuile acestora și returnează aria intersecției celor două folosindu-se de funcțiile minim și maxim. Funcția verifică mai întâi dacă cele două ferestre au puncte de intersecție:

```
int arie_intersectie(Fereastra a, Fereastra b, int lat, int lung)
{
    if(abs(a.lin-b.lin)>=lat||abs(a.col-b.col)>=lung)
        return 0;
    int l1=minim(a.lin,b.lin);
    int c1=maxim(a.col,b.col);
    int l2=maxim(a.lin+lung-1,b.lin+lung-1);
    int c2=minim(a.col+lat-1,b.col+lat-1);
    int l= abs(l1-l2);
    int L= abs(c1-c2);
    return l*L;
}
```

Funcția *arie_reuniune* are ca parametrii de intrare două ferestre și dimensiunile acestora și returnează aria reuniunii celor două folosindu-se de funcția *arie_intersectie:*

```
int arie_reuniune(Fereastra a, Fereastra b, int lat, int lung)
{
    return 2*lat*lung-arie_intersectie(a,b, lat, lung);
}
```

Funcția *eliminarea_non_maximelor* are ca parametrii de intrare dimensiunile unei ferestre, iar ca parametrii de ieșire vectorul de ferestre din care se va face eliminarea și numărul elementelor acestuia. Funcția se folosește de funcțiile anterioare: *arie_intersectie* și *arie_reuniune* pentru a calcula suprapunerea a două ferestre și le elimină pe acelea ce au suprapunerea mai mare decât 0.2:

```
void eliminarea_non_maximelor(Fereastra **D, int *n, int lat, int lung)
{
    double suprapunere;
    int i,j,k;

    for(i=0;i<(*n)-1;i++)
        for(j=i+1;j<(*n);j++)
    {
        suprapunere= (double)arie_intersectie((*D)[i],(*D)[j],lat, lung)/(double)arie_reuniune((*D)[i],(*D)[j], lat, lung);

        if(suprapunere>0.2)
        {
            for(k=j+1;k<(*n);k++)
                (*D)[k-1]=(*D)[k];
            (*n)--;
            j--;


          (*D)=(Fereastra*)realloc((*D),(*n)*sizeof(Fereastra));
          if((*D)==NULL)
          {
              printf("Nu s-a gasit suficienta memorie");
              return;
          }


        }
    }
}
```

Funcția *colorare_total* primește ca parametrii de intrare vectorul de ferestre, lungimea acestuia și dimensiunile unei ferestre, iar ca parametru de ieșire primește o imagine pe care se vor desena toate detecțiile după eliminarea non-maximelor:

```
void colorare_total( Fereastra *D, int n, char *imagine, int lat, int lung)
{
    int i;
    for(i=0;i<n;i++)
        colorare(imagine, D[i], lat, lung);

}
```

Programul principal citește din fișier numele imaginilor și apelează funcțiile: *detectii, sortare, eliminarea_non_maximelor* și *colorare_total*:

```
int main()
{
    Fereastra *D;
    int n,i,lat,lung;

    char imagine[]="test.bmp";
    char imagine_gray[]="test_gray.bmp";
    char sablon[]="cifra0.bmp";
    citire_dimensiuni(sablon,&lat, &lung);

    FILE *fin=fopen("nume_imagini.txt", "r");
    if(fin==NULL)
    {
        printf("eroare la deschiderea fisierului");
        return;
    }

    printf("Numele imaginii sursa este: ");
    fscanf(fin,"%s", imagine);
    printf("%s", imagine);

    printf("\nNumele imaginii pe care se vor desena detectiile sabloanelor este: ");
    fscanf(fin,"%s", imagine_gray);
    printf("%s", imagine_gray);
    printf("\nVom considera cele 10 sabloane aflate in fisier. \nAsteptati...");

    detectii(&D,&n,imagine,imagine_gray);

    sortare(D,n);

    eliminarea_non_maximelor(&D,&n, lat, lung);

    colorare_total(D,n,imagine_gray, lat, lung);

    free(D);
    fclose(fin);

    return 0;
}
```