Proiect la
Programare Procedurală

Nume și prenume: Donici Alexandra-Maria

Grupa:144

Proiectul este structurat pe două părți, prima ce constă în criptarea, respectiv decriptarea unei imagini și cea de-a doua ce urmărește recunoașterea de pattern-uri într-o imagine.

Imaginile color le-am manipulat în limbajul C ca fișiere binare. Acestea, fiind în formatul BMP, stochează 3 octeți pentru fiecare pixel, reprezentând intensitatea acestuia. De aceea, am creat structura de tip Pixel pentru a stoca cei trei octeți R,G,B.

Funcția *void dim(int *latime, int *inaltime, char * cale_imagine);* determină dimensiunea unei imagini în pixeli și este este de forma „latime X inaltime". „latime" reprezintă numărul de pixeli pe lățime și este memorată pe patru octeți fără semn începând cu octetul al 18-lea din header, iar „inaltime" reprezintă numărul de pixeli pe înălțime, memorați începând cu octetul al 22–lea din header. Cele două dimensiuni sunt transmise ca parametri de ieșire.

Funcția *void header(char *cale_imagine, char **h);* salvează în vectorul h header-ul unei imagini. Acesta ocupă primii 54 de octeți ai fișierului și conține informații despre formatul BMP , precum și despre dimensiunea imaginii, numărul de octeți utilizați pentru reprezentarea unui pixel etc.

Funcția *int padding(int latime);* returnează numărul de octeți de padding. Imaginile în format BMP conțin pe fiecare linie un număr de octeți multiplu de 4. Astfel, se adaugă octeți de padding astfel încât numărul total de octeți de pe fiecare linie să fie multiplu de 4. Numărul de octeți de pe o linie este „3 X latime", de aceea, formula de calcul a padding-ului este „4-(3*latime)%4".

Funcția *void liniarizare(char * cale_imagine, pixel **p, int latime, int inaltime);* încarcă în memoria internă (vectorul p) o imagine de tip BMP în formă liniarizată. Pentru liniarizarea corectă a imaginii am ținut cont de următoarele detalii: am calculat padding-ul și la finalul fiecărei linii am sărit peste octeții de padding, am memorat de la dreapta la stânga, adică în ordinea B,G,R, octeții corespunzători celor 3 canale de culoare și am ținut cont că primul octet citit din fișier este defapt primul de pe ultima linie din imagine (imaginea este memorată invers în fișierul binar).

Funcția *void deliniarizare(char *img_finala, pixel *v,int latime,int inaltime,char *h);* salvează în memoria externă o imagine BMP stocată în formă liniarizată în memoria internă (vectorul p). La deliniarizare se va ține cont de aceleași aspecte ca la liniarizare, iar în ceea ce privește padding-ul se vor adăuga octeți egali cu 0. În fișierul binar va fi copiat header-ul imaginii inițiale, salvat în vectorul h, iar apoi vor fi copiați octeții din vectorul p, în ordinea corespunzătoare.

Funcția *void XORSHIFT32(unsigned int **r, unsigned int r0,int latime, int inaltime);* generează o secvență de „2 X latime X inaltime - 1"(notat cu k) numere întregi aleatoare fără semn pe 32 de biți, memorate în vectorul r. Valoarea inițială (seed) este transmisă ca parametru (r0), aceasta fiind preluată din fișierul text „secret_key.txt". Numerele aleatoare se determină prin shiftarea cu 13 poziții la stânga, apoi cu 17 poziții la dreapta și 5 poziții la stânga a

numărului generat anterior. Generatorul XORSHIFT32 a fost propus de George Marsaglia in 2003.

Funcția *void permutare(unsigned int *r, unsigned int **perm, int latime, int inaltime);* generează o permutare memorată în vectorul perm de dimensiune „latime X inaltime”, folosind algoritmul lui Durstenfeld.

Algoritmul lui Durstenfeld are următoarea structură:

```
for(i=0;i<k;i++)
    (*perm)[i]=i;

for (i=k-1;i>=1;i--)
{
    x=r[k-i]%(i+1);

    aux=(*perm)[x];
    (*perm)[x]=(*perm)[i];
    (*perm)[i]=aux;
}
```

.

Se inițializează fiecare element al vectorului cu indicele fiecărei poziții, iar apoi se interschimbă elementul de pe poziția i cu elementul de pe poziția x, unde x este un număr generat aleator cu ajutorul funcției XORSHIFT32, cuprins între 0 și i.

Funcția *void permutaPixeli(pixel *p, int latime, int inaltime, pixel **pp, unsigned int *perm);* permută pixelii imaginii liniarizate în vectorul p, obținându-se o imagine liniarizată intermediară p’(vectorul pp). Permutarea pixelilor se realizează după următoarea formulă, folosind și permutarea generată de funcția anterioară (vectorul perm):

```
for(i=0;i<k;i++)
    (*pp)[perm[i]]=p[i];
```

Funcția *pixel xor_numar(pixel x, unsigned int a);* generează operația XOR dintre un pixel și un număr întreg fără semn x pe 32 de biți format din octeții fără semn (x_3 , x_2 , x_1 , x_0) obținuți cu ajutorul uniunii de tip Bytes (t.b[0] = x_0; t.b[1] = x_1 ; t.b[2] = x_2). Astfel operația XOR va fi aplicată octetului R cu x_2 , octetului G cu x_1 și octetului B cu x_0 , octetul x_3 nefiind utilizat.

Funcția *pixel xor_pixeli(pixel x, pixel y);* generează operația XOR dintre doi pixeli. Operația XOR va fi aplicată octetului R a primului pixel cu octetul R al celui de-al doilea pixel, analog pentru octeții G și B.

Funcția *void criptare(char *imagine_initiala, char * imagine_criptata, char * cheie_secreta);* criptează o imagine de tip BMP. Se liniarizează imaginea inițială și se obține vectorul p, se generează numerele aleatoare din vectorul r cu ajutorul funcției XORSHIFT32, dar și permutarea perm, iar apoi se permută pixelii formându-se imaginea intermediară stocată în vectorul pp. R0 și sv sunt preluate din fișierul text ce conține cheia secretă. Imaginea criptată c se obține aplicând fiecărui pixel al imaginii intermediare următoarea relație de substituție:

```
c[0]=xor_numar(pp[0],sv);
c[0]=xor_numar(c[0],r[k]);

for(i=1;i<k;i++)
{
   c[i]=xor_pixeli(c[i-1],pp[i]);
   c[i]=xor_numar(c[i],r[k+i]);
}
```

În urma deliniarizării vectorului c se obține imaginea criptată.

Funcția *void inversa_permutarii(unsigned int *perm,unsigned int **perm_inv, int latime, int inaltime);* generează inversa permutării perm stocată în vectorul perm_inv folosind formula:

```
for(i=0;i<k;i++)
     (*perm_inv)[perm[i]]=i;
```

Funcția *void substitutie (pixel **cp, pixel *c, unsigned int *r, unsigned int sv,int latime, int inaltime);* aplică fiecărui pixel din imaginea criptată c, inversa relației de substituție folosită în procesul de criptare, obținându-se o imagine intermediară cp:

```
(*cp)[0]=xor_numar(c[0],sv);
(*cp)[0]=xor_numar((*cp)[0],r[k]);

for(i=1;i<k;i++)
{
   (*cp)[i]=xor_pixeli(c[i-1],c[i]);
   (*cp)[i]=xor_numar((*cp)[i],r[k+i]);
}
```

Funcția *void decriptare(char * imagine_criptata, char * imagine_decriptata, char * cheie_secreta);* generează imaginea decriptată d. Se liniarizează imaginea criptată și se obține vectorul c, se generează numerele aleatoare din vectorul r cu ajutorul funcției XORSHIFT32, dar

și permutarea perm, iar apoi se determină inversa permutării perm salvată în vectorul perm_inv. Se aplică funcția de substituție imaginii c, se obține imaginea intermediară cp. R0 și sv sunt preluate din fișierul text ce conține cheia secretă. Imaginea decriptată d se obține permutând pixelii imaginii intermediare cp conform permutării perm_inv după următoarea formulă:

```
for(i=0;i<k;i++)
    d[perm_inv[i]]=cp[i];
```

Funcția *void chi(char *cale_imagine);* afișează valorile testului χ^2 pentru o imagine BMP pe fiecare canal de culoare. Testul χ^2 este un instrument statistic de evaluare a uniformității distribuției valorilor dintr-un șir. Am folosit 3 vectori de frecvență r,g,b pentru calcularea frecvenței valorii i (0 <= i <= 255) pe un canal de culoare al imaginii (r pentru red, g pentru green, b pentru blue). „f_bar" reprezintă frecvența estimată teoretic a oricărei valori i, calculată după formula: *f_bar=(latime*inaltime)/256.0;*. După determinarea celor trei vectori de frecvență și a frecvenței f_bar, χ^2 se calculează după formula:

```
for(i=0;i<=255;i++)
  {
    xr=xr+(((r[i]-f_bar)*(r[i]-f_bar))/f_bar);
    xg=xg+(((g[i]-f_bar)*(g[i]-f_bar))/f_bar);
    xb=xb+(((b[i]-f_bar)*(b[i]-f_bar))/f_bar);
  }
```

Programul principal (main) conține citirea căilor imaginilor și a fișierului ce conține cheia secretă, dar și apelul funcțiilor de criptare, de decriptare și de afișare a valorilor testului χ^2 pentru imaginea inițială și cea criptată.

Cea de-a doua parte urmărește recunoașterea cifrelor scrise de mână, detectarea acestora într-o imagine prin suprapunerea unor șabloane și încadrarea acestora cu o culoare specifică fiecărei cifre.

Funcțiile *dim, header* și *padding* sunt identice cu cele din prima parte și au același efect.

Funcția *void grayscale_image (char * nume_fisier_sursa, char * nume_fisier_destinatie);* transformă o imagine color într-o imagine în tonuri de gri (grayscale) înlocuind valorile (red, green,blue) cu valorile (red',green',blue') definite astfel:

*y = red' = green' = blue' = 0.299*red+0.587*green+0.114*blue;*

Astfel, imaginea grayscale obținută utilizează tot 3 octeți pentru reprezentarea fiecărui pixel, doar că aceștia au valori egale.

Funcția *void matrice_imagine(char *cale_imagine, pixel ***v);* încarcă în memoria internă (matricea v) o imagine de tip BMP. Pentru conversia corectă a imaginii am ținut cont de următoarele detalii: am calculat padding-ul și la finalul fiecărei linii am sărit peste octeții de padding, am memorat de la dreapta la stânga, adică în ordinea B,G,R octeții corespunzători celor 3 canale de culoare și am ținut cont că primul octet citit din fișier este defapt primul de pe ultima linie din imagine (imaginea este memorată invers în fișierul binar).

Funcția *void imagine(char *cale_imagine, char *cale_imagine_finala, pixel **v););* salvează în memoria externă o imagine BMP stocată în memoria internă (matricea v). La conversie se va ține cont de aceleași aspecte ca la transformarea din imagine în matrice, iar în ceea ce privește padding-ul se vor adăuga octeți egali cu 0 la sfârșitul fiecărei linii, unde este cazul. În fișierul binar va fi copiat header-ul imaginii inițiale, salvat în vectorul h, iar apoi vor fi copiați octeții din matricea v, în ordinea corespunzătoare.

Funcția *double s_bar(pixel **s,unsigned int latime, unsigned int inaltime);* calculează media valorilor intensităților grayscale a pixelilor în fereastra s (media celor 165 de pixeli din șablonul s):

```
for(i=0;i<inaltime;i++)
  for(j=0;j<latime;j++)
       sum=sum+s[i][j].red;

sb=sum/(latime*inaltime);
```

*Funcția double sigma_s(pixel **s, unsigned int latime, unsigned int inaltime);* calculează deviația standard r a valorilor intensităților grayscale a pixelilor în șablonul s după formula:

```
for(i=0;i<inaltime;i++)
     for(j=0;j<latime;j++)
       sum=sum+(s[i][j].red-s_bar(s,latime,inaltime))*(s[i][j].red-s_bar(s,latime,inaltime));

sum=sum/(latime*inaltime-1);
r=sqrt(sum);
```

Funcția *double fI_bar(pixel **a, unsigned int latime, unsigned int inaltime, unsigned int pi, unsigned int pj);* calculează media valorilor intensităților grayscale a pixelilor din fereastra f (media celor 165 de pixeli din fereastra f). pi și pj sunt coordonatele colțului din stânga sus a ferestrei f care aparține imaginii a. Media valorilor notată fb se determină astfel:

```
for(i=pi;i<pi+inaltime;i++)
for(j=pj;j<pj+latime;j++)
sum=sum+a[i][j].red;
```

```
fb=sum/(latime*inaltime);
```

Funcția *double sigma_fI(pixel **a, unsigned int latime, unsigned int inaltime, unsigned int pi, unsigned int pj);* reprezintă deviația standard r a valorilor intensităților grayscale a pixelilor în fereastra f, pi și pj fiind coordonatele colțului din stânga sus a ferestrei f care aparține imaginii a:

```
fI=fI_bar(a,latime,inaltime,pi,pj);
  for(i=pi;i<pi+inaltime;i++)
    for(j=pj;j<pj+latime;j++)
    sum=sum+(a[i][j].red-fI)*(a[i][j].red-fI);

  sum=sum/(latime*inaltime-1);
  r=sqrt(sum);
```

Funcția *double corr(pixel **s, pixel **a, unsigned int pi, unsigned int pj, unsigned int latime, unsigned int inaltime);* calculează corelația dintre șablonul s și fereastra f care aparține imaginii a determinată de coordonatele pi și pj (colțul din stânga, sus). Pentru calculul corelației sum avem nevoie de media valorilor intensităților grayscale a pixelilor în fereastra s (sb), deviația standard a valorilor intensităților grayscale a pixelilor în șablonul s (sigmas), media valorilor intensităților grayscale a pixelilor din fereastra f (fI) și deviația standard a valorilor intensităților grayscale a pixelilor în fereastra f (sigmaf). Corelația sum se determină după următoarea formulă:

```
for(i=0;i<inaltime;i++)
      for(j=0;j<latime;j++)
        sum=sum+(double)(((a[pi+i][pj+j].red-fI)*(s[i][j].red-sb))/(sigmas*sigmaf));

  sum=sum/(inaltime*latime);
```

Funcția *void colorare(pixel **a, unsigned int pi, unsigned int pj,unsigned int latime, unsigned int inaltime, pixel c);* colorează utilizând culoarea c, conturul ferestrei f din imaginea a, determinate de coordonatele pi si pj a colțului din stânga, sus.

```
for(i=pi;i<pi+inaltime;i++)
  {
    a[i][pj]=c;
    a[i][pj+latime-1]=c;
  }

  for(j=pj;j<pj+latime;j++)
```

```
{
   a[pi][j]=c;
   a[pi+inaltime-1][j]=c;
}
```

Funcția *void culori (pixel **c);* completează vectorul c de tip pixel cu octeții pentru cele 10 culori folosite la încadrarea detecțiilor șabloanelor.

Funcția *void template_matching(char * cale_imagine, char * cale_sablon, double ps, fereastra **d,unsigned int *l, pixel colour);* detectează toate suprapunerile șablonului cale_sablon cu ferestrele din imaginea cale_imagine care au corelația mai mare decât pragul ps. Pentru ca în momentul suprapunerii șablonul să încapă în imagine se calculează p și q, coordonatele maxime până la care se parcurge matricea imaginii. Când corelația dintre șablon și o fereastră din imagine este mai mare decât pragul ps, se adaugă coordonatele în vectorul d de tip fereastra (o structură) care memorează atât cele două coordonate, cât și corelația și culoarea c cu care trebuie colorat conturul ferestrei. „l" reprezintă lungimea vectorului d.

Funcția *int cmp_descrescator (const void *a, const void *b*); este funcția de comparare folosită pentru apelul funcției qsort în cazul sortării descrescătoare a unui vector de tip fereastră în funcție de corelație. Are ca parametri doi pointeri către void și returnează un întreg care este mai mic decât 0 (-1) în cazul în care primul argument este mai mare decât cel de-al doilea, un întreg mai mare dacât 0 (1) în cazul în care primul argument este mai mic decât cel de-al doilea și 0 când numerele sunt egale.

Funcția *double suprapunere(unsigned int x1,unsigned int y1,unsigned int x2,unsigned int y2,unsigned int latime, unsigned int inaltime);* calculează suprapunerea spațială dintre două detecții determinate de coordonatele (x1,y1) și (x2,y2). Se calculează lățimea l și înălțimea h a intersecției dintre cele două detecții în funcție de poziția punctelor în imagine (în total 9 cazuri), se calculează aria intersecției (*a_intersectie=l*h*;) și aria reuniunii, care este suma ariilor celor două ferestre din care scădem aria intersecției. Suprapunerea s se determină în funcție de aria intersecției și a reuniunii după formula: *s=(double)a_intersectie/a_reuniune;*

Funcția *void eliminare_non_maxime(fereastra **d,unsigned int *l, unsigned int latime_sablon, unsigned int inaltime_sablon);* elimină detecțiile d[j] care au o suprapunere sup mai mare de 0.2 cu detecția d[i], unde i<j. În momentul ștergerii unei detecții din vectorul d, se actualizează lungimea acestuia l și i se realocă memorie.

```
if(sup>0.2)
{ for(k=j;k<(*l)-1;k++)
(*d)[k]=(*d)[k+1];
```

*(*l)--;*

Programul principal conține citirea dintr-un fișier text a căilor imaginii test și a celei grayscale, cât și a șabloanelor și a copiilor lor grayscale. De asemenea se apelează funcția *template_matching* pentru fiecare șablon și se creează astfel vectorul d al detecțiilor. Acesta este sortat descrescător în funcție de corelație cu ajutorul funcției qsort și este apelată funcția *eliminare_non_maxime.* Pentru detecțiile rămase, desenăm în imagine conturul acestora cu o culoare specifică fiecărei cifre. Se ține cont pe tot parcursul programului de eliberarea memoriei vectorilor alocați dinamic și de ștergerea imaginilor grayscale după utilizarea acestora.