Temă proiect la disciplina Programare Procedurală

Student:

Muscalu Elena-Claudia

Seria:

14

Modulul de criptare/decriptare

Explicitarea funcțiilor și variabilelor folosite

1. Criptarea

Cum în acest proiect am lucrat doar cu imagini color, am ales să folosesc o structură pentru a gestiona cu ușurință pixelii dintr-o anumită imagine, dar și octeții pe fiecare canal de culoare: R (red), G (green), B(blue):

```
5  typedef struct {unsigned char B;
6                  unsigned char G;
7                  unsigned char R;
8                 }pixel;
9
```

În următoarele linii de cod am scris două funcții pentru calculul lățimii și înălțimii, pentru a nu fi nevoită să le calculez de fiecare dată, acestea au fost apelate în main și apoi folosite ca parametri în alte funcții.

Lățimea imaginii exprimată în pixeli este memorată pe patru octeți fără semn începând cu octetul al 18-lea din header, de aceea m-am poziționat utilizând **fseek** la acesta, în cazul în care a existat o eroare la deschiderea fișierului binar pentru citire, voi returna 0, urmând să verific în main dacă lățimea este diferită de 0, în acest caz voi afișa un mesaj pe ecran. De asemenea, am procedat asemănător și pentru calculul înălțimii, însă poziționându-mă la octetul al 22-lea din header.

Mai jos putem observa antetul celor două funcții anterior explicate:

unsigned int latime_imagine (char * nume_fisier_sursa)

unsigned int inaltime_imagine (char * nume_fisier_sursa)

În formatul BMP, există o zonă de date de dimensiune fixă , 54 de octeți, ce conține informații despre imagine. De aceea am folosit o funcție ce memorează header-ul unei imaginii într-un tablou unidimensional alocat dinamic, numit H.

Antetul funcției ce memorează header-ul unei imagini:

void header_imagine_sursa (char * nume_fisier_sursa, unsigned char **H)

Următoarea funcție conține transformarea zonei de date cu dimensiune variabile, unde sunt pixelii imaginii propriu-zise într-o formă liniară alocată dinamic, vectorul în care vom păstra pixelii imaginii va fi de tipul structurii declarate – pixel, numit L.

Antetul funcției:

void liniarizare_imagine (char * nume_fisier_sursa, pixel **L, unsigned int H_imagine, int W_imagine)

Pentru liniarizarea imaginii ne vom poziționa în fișierul binar la octetul 54 (sărim peste heder-ul imaginii), pentru a putea citi pixelii imaginii, vom aloca dinamic pentru vectorul L lățime*înăltime pixeli, pe parcursul programului am folosit pentru lățimea și înălțimea imaginii următoarele notații: W_imagine, respectiv H_imagine. Alipirea liniilor din imaginea sursă s-a efectuat de sus în jos, având grijă să sar peste padding la fiecare linie. (cerința 2)

Deliniarizarea presupune construirea unei noi imagini pornind de la cea inițială cu ajutorul vectorului L, în care am păstrat pixelii și a vectorul H, în care am păstrat header-ul acesteia.

Antetul funcției de deliniarizare:

void deliniarizare(char *nume_fisier_sursa, char * nume_fisier_destinatie, pixel *L ,unsigned int H_imagine, unsigned int W_imagine)

În această funcție am trimis calea fișierului sursă pentru a putea memora header-ul și calea unui fișier destinație, unde va apărea noua imagine creată, vectorul în care s-a realizat liniarizarea și lățimea și înălțimea imaginii sursă. La deliniarizare trebuie să avem grijă la scrierea corectă a pixelilor în fișierul binar și la octeții de completare pentru fiecare linie, în cazul în care există padding, astfel dacă padding-ul este diferit de 0 vom scrie în fișierul binar octeți de 0. La finalul funcției vom dezaloca memoria pentru vectorul în care păstrăm header-ul, deoarece din acest moment nu vom mai avea nevoie de el. (cerința 3)

Următoarele două funcții descriu operația de *sau_exclusiv* dintre doi pixeli, respectiv dintre un pixel și un întreg fără semn:

pixel P1_sau_exclusiv_P2 (pixel P1,pixel P2)

Prima operație va returna un nou pixel, numit P_nou.

pixel P_sau_exclusiv_X (pixel P, unsigned int X)

Cea de-a doua operație va returna tot un nou pixel, fără a utiliza octetul cel mai semnificativ al numărului întreg fără semn. Pentru a gestiona cu ușurință octeții numărului întreg fără semn am declarat o uniune, apoi am aplicat operația de *sau_exclusix.*

Funcția implementată ulterior folosește generatorul de numere pseudo-aleatoare pentru a genera 2*lățime*înălțime numere perfect aleatoare. Numerele le voi păstra în vectorul R.(cerința 1)

void XORSHIFT32 (unsigned int ** R, char * nume_fisier_text, unsigned int WxH_imagine)

Ca parametri avem vectorul ce păstrează numerele aleatoare, numele fișierului text ce conține cheia secretă și înălțimea*lățimea imaginii, notată WxH_imagine. Pe poziția 0 în vectorul R vom păstra seed-ul (cheia secretă citită din fișierul text), pornind de la acesta urmează să generăm numere întregi fără semn pe 32 de biți, cu un caracter pseudo-aleator foarte bun, folosind operații de deplasare pe biți.

Funcția ce urmează generează cu ajutorul **algoritmului lui Durstenfeld** și a funcției anterioare (**XORSHIFT32**) o permutare (Permutare) de lungime lățime*înălțime cu numere aleatoare din intervalul (0,k).

Antetul funcției:

void Durstenfeld(unsigned int **Permutare, unsigned int *R, unsigned int WxH_imagine_R)

Ca parametri folosiți avem vectorul Permutare în care vom memora permutarea aleatoare, vectorul R de numere aleatoare și înălțimea*lățimea.

În umătoarea funcție vom permuta pixelii imaginii.

Antet funcției ce permută pixelii imaginii:

void permutare_pixeli(unsigned int *Permutare, pixel *L, pixel **Lpermutat, unsigned int WxH_imagine_Permutare)

Ca parametri folosiți avem vectorul Permutare în care vom memora permutarea aleatoare, vectorul L de pixeli, un nou vector Lpermutat ce va păstra pixelii permutați și înălțimea*lățimea. După aplicarea permutării vom obține o nouă imagine. Astfel funcția **imagine_permutata** construiește o imagine cu pixelii permutați, numită **pepperspermutat.bmp**.

Următorul pas este criptarea imaginii **peppers.bmp**, astfel funcția următoare realizează acest lucru.(cerința 4)

void criptare (char * nume_fisier_text, pixel *Lpermutat,unsigned int *R, pixel **C, unsigned int W_imagine, unsigned int H_imagine, char * nume_fisier_sursa, char * nume_imagine_criptata)

Ca parametri avem numele fișierului text ce conține SV (starting value), vectorul ce păstrează pixelii permutați (Lpermutat), vectorul cu numere aleatoare (R),

vectorul în care se va păstra imaginea criptată, numit C și lățimea și înălțimea imaginii sursă, precum și calea fișierului sursă și calea fișierului ce va păstra imaginea criptată.

Pentru criptare vom aplica relația de substituție folosind funcțiile de *sau_exclusiv* dintre pixeli doi pixeli și dintre un pixel și un număr întreg fără semn pe 32 de biți. Având ca parametri calea imaginii inițiale și calea imaginii destinație putem realiza tot în această funcție și deliniarizarea vectorului ce păstrează imaginea criptată. În main vom apela funcția de criptare numele imaginii criptate fiind citit de la tastatură.

2. Decriptarea

Complementar algoritmului de criptare este cel de decriptare, astfel în cele ce urmează voi prezenta funcțiile folosite la decriptarea imaginii.(cerința 5)

Vom folosi funcțiile XORSHIFT32 și Durstenfeld deja prezentate.

O altă funcție folosită este următoarea:

void inversa (unsigned int **Inversa, char * nume_fisier_text, unsigned int WxH_imagine)

Aceasta folosește permutarea generată cu ajutorul algoritmului lui Durstenfeld și calculează inversa (Inversa).

După aceasta se aplică pentru fiecare pixel din imaginea criptată inversa relației de substituție folosită în procesul de criptare, obținându-se o imagine intermediară, numită C_prim.

void modificare_imgcriptata(pixel *C, unsigned int *R, pixel **C_prim, unsigned int WxH_imagine, char * nume_fisier_text)

Decriptarea are loc în funcția următoare, imaginea decriptată fiind memorată in vectorul D, de asemenea vom folosi și imaginea intermediară păstrată în vectorul C_prim.

Antetul funcției, unde putem observa parametrii folosiți, se poate observa mai jos:

void decriptare (pixel *C, pixel **D, unsigned int *R, unsigned int W_imagine,unsigned int H_imagine, char * nume_fisier_sursa, char * nume_imagine_decriptata,char * nume_fisier_text)

Ca parametri folosiți avem vectorul ce păstrează imaginea criptată, un vector ce va păstra imaginea decriptată (D), vectorul de numere aleatoare R, lățimea imaginii, înălțimea imaginii, calea fișierului sursă, calea fișierului în care se va

face deliniarizarea imaginii decriptate păstrate în vectorul D și calea fișierului text. În interiorul funcției am apelat funcția ce calculează inversa, **inversa**, funcția ce creează imaginea intermediară, **modificare_imgcriptata** și funcția de deliniarizare, **deliniarizare.**

3.Testul chi-pătrat

Funcția care afișează valorile testului chi-pătrat pe fiecare canal de culoare este următoarea:

void test_chi_patrat(unsigned int W_imagine,unsigned int H_imagine,char * cale_imagine)

Drept parametri avem lățimea imaginii, înălțimea imaginii și calea unui fișier binar.

În funcție folosim 3 vectori de frecvență, pentru care folosim **calloc** pentru alocarea memoriei și inițializarea cu 0. Vom mări frecvența pentru fiecare octet pentru un canal de culoare, calculăm f_teoretic și aplicăm formula chi_pătrat. La final vom afișa valorile testului chi_pătrat pe canalele de culoare RGB. În main vom apela această funcție o dată pentru imaginea în clar și o dată pentru imaginea criptată.

În main vom apela funcțiile pentru a cripta și decripta o imagine și pentru a afișa valorile lui chi-pătrat. Numele fișierului sursă ce urmează a fi criptat este citit de la tastatură (**peppers.bmp)**, la fel și numele fișierului ce va conține imaginea criptată, numele fișierului ce va conține imaginea decriptată și numele fișierului text ce conține cheia secretă (**secret_key.txt**).

Pe parcursul programului vom avea grijă să dezalocăm memoria în fiecare funcție în care este nevoie, să închidem fișierele, să verificăm de asemenea dacă alocarea memoriei a avut loc.

Modulul de recunoaștere de pattern-uri (cifre scrise de mână)

Explicitarea funcțiilor și variabilelor folosite

Asemenea primei părți a proiectului, cea de criptare și decriptare, am folosit și aici o structură ce memorează pentru fiecare pixel fiecare canal de culoare: red, green, blue. Tot de la început am declarat o structură ce va fi necesară mai târziu, ce memorează linia și coloana pentru o fereastră (linia și coloana din colțul stânga sus pentru o ferestră), pragul de corelație și o culoare ce este de tip pixel.

```
6   typedef struct { unsigned char R;
7                    unsigned char G;
8                    unsigned char B;}pixel;
9
10  typedef struct { unsigned int x;
11                   unsigned int y;
12                   float prag;
13                   pixel culoare;
14                  }detectie;
15
```

Următoarele trei funcții sunt descrise detaliat în prima parte a proiectului:

1. Memorarea unei imagini sub forma unei matrice

unsigned int latime_imagine (char * nume_fisier_sursa)

unsigned int inaltime_imagine (char * nume_fisier_sursa)

void header_imagine_sursa (char * nume_fisier_sursa, unsigned char **H)

Funcția următoare păstrează pixelii unei imaginii într-o matrice, având grijă la padding pentru fiecare linie.

void transform_matrice (char * nume_fisier_sursa, pixel ***M, unsigned int H_imagine, int W_imagine)

Parametri folosiți reprezintă calea fișierului sursă, matricea de tip pixel M, în care se vor păstra pixelii imaginii sursă, precum și înălțimea (H_imagine) și lățimea (W_imagine). În matricea M, pixelii imaginii se vor păstra în ordinea citirii de jos în sus.

Funcția **deliniarizare_matrice** ajută la construirea unei noi imagini cu ajutorul matricei ce conține pixelii și cu ajutorul header-ului memorat în H. Vom parcurge matricea de jos in sus (de la ultima linie H_imagine -1) și vom scrie in fișierul binar octet cu octet, la fiecare final de linie vom avea grijă la padding, iar când acesta este diferit de 0, vom completa cu octeți egali cu 0 la final de linie. La finalul funcției vom dezaloca memoria pentru vectorul ce păstrează header-ul, deoarece nu ne va mai fi necesar.

2. Transformarea unei imagini color în una grayscale

Funcția ce umează transformă o imagine color într-una grayscale.

void grayscale (char * nume_fisier_sursa, char * nume_fisier_destinatie, unsigned int W_imagine, unsigned int H_imagine)

Parametri funcției sunt calea imaginii color, calea imaginii în care se va memora imaginea grayscale, lățimea și înălțimea imaginii. Vom citi câte 3 octeți din imaginea sursă le vom aplica tuturor celor 3 aceeași valoare și îi vom scrie în fișierul destinație, pentru octeții de padding vom scrie octeți negri în fișierul binar.

3. Calculul corelației

Pentru calculul corelației am folosit mai multe funcții:

float medie_intensitati_fI(pixel **M, unsigned i, unsigned j, unsigned W_sablon, unsigned H_sablon)

Antetul de mai sus este al funcției ce calculează media intensităților grayscale a pixelilor din fereastra fI (media celor 165 de pixeli din fereastra fI).

Umătoarea funcției folosită returnează deviația standard a valorilor intensităților grayscale a pixelilor din fereastra fI. În această funcție vom folosi și funcția anterioară **medie_intensitati_fI.**

float deviatia_standard_fI (pixel **M, unsigned int i, unsigned int j, unsigned int W_sablon, unsigned int H_sablon)

Drept parametri avem: matricea M ce conține pixelii imaginii, coordonatele ferestrei, respectiv linia i și coloanal j și lățimea și înălțimea unui șablon, dimensiunile șablonului ne sunt necesare pentru a incadra o fereastră în matricea M.

Funcția **medie_intensitati_sablon** calculează media intensităților grayscale a pixelilor pentru un șablon S.

float medie_intensitati_sablon(pixel **S, unsigned int W_sablon, unsigned int H_sablon)

Parametrii folosiți reprezintă matricea ce memorează pixelii șablonului, lățimea și înălțimea șablonului.

Funcția **deviatia_standard_sablon** calculează deviația standard a valorilor intensităților grayscale a pixelilor în șablonul S.

float deviatia_standard_sablon (pixel **S, unsigned int W_sablon, unsigned int H_sablon)

Toate cele patru funcții explicate mai sus sunt reunite în funcția **corelatie,** ce calculează corelația dintre șablonul S și o fereastră fI.

float corelatie (pixel **M, pixel **S, unsigned int W_sablon, unsigned int H_sablon,unsigned int i, unsigned int j)

Parametri folosiți pentru această funcție sunt matricea M ce memorează imaginea sursă (**test.bmp**), matricea S, ce memorează un șablon cu una dintre cifre și linia i și coloana j.

4. Template matching

void template_matching(pixel **I, unsigned int W_imagine, unsigned int H_imagine, pixel **S, unsigned int W_sablon, unsigned int H_sablon, float prag_furnizat, detectie **D, unsigned int *k, pixel C, char * cale_sablon)

Funcția **template_matching** implementează operația de template matching dintre o imagine I și un șablon S, aceasta furnizează ferestrele fI care au corelația mai mare de un anumit prag, toate aceste ferestre vor fi memorate într-un vector de tip structură (**detecție**), păstrând pentru fiecare fereastră linia și coloana corespunzătoare colțului din stânga sus. De asemenea, în vectorul D de tip structură, vom mai păstra pragul de corelație și o culoare specifică fiecărui șablon.

Parametrii folosiți sunt imaginea memorată în matricea I, lățimea și înălțimea acesteia, W_imagine, respective H_imagine, matricea S, ce memorează pixelii șablonului, precum și dimensiunile acesteia W_sablon și H_sablon, un prag de corelație furnizat, vectorul de tip detecție D ce va păstra toate detecțiile mai mari de un anumit prag, lungimea vectorului D, o anumită culoare C și calea unui anumit șablon.

In main am rulat operația de template matching pentru toate cele 10 șabloane existente folosind un prag de corelație de 0.5. Am folosit un switch ce are 10 cazuri de la 0 la 9 iar pentru fiecare caz, adică fiecare șablon se va memora o culoare diferită, coordonatele ferestrei și pragul de corelație în vectorul D de tip corelație și vom apela funcția **template matching** prezentată mai sus.

O altă funcție este **colorare** ce desenează conturul ferestrei fI în imaginea I, aceasta primește ca parametri matricea M ce memoreaza pixelii imaginii I, linia i și j reprezentând coordonatele ferestrei (colțul din stânga sus al ferestrei), culoarea cu care se va executa conturul și dimensiunile șablonului.

void colorare (pixel **M, unsigned int i, unsigned int j,pixel C, unsigned int W_sablon, unsigned int H_sablon)

Funcția **coloreaza** este folosită pentru a colora imaginea I cu ajutorul funcției **colorare** și a vectorului D de tip detecție, ce memorează coordonatele fiecărei ferestre, culoarea cu care se va realiza colorarea și pragul de corelație.

void coloreaza (detectie *D, unsigned int k, pixel **M,unsigned int W_sablon,unsigned int H_sablon)

În main vom apela funcția **coloreaza** pentru a colora imaginea I folosind toate șabloanele.

5. Sortarea vectorului de tip detectie D

Funcția **cmpdescrescator** este folosită în qsort pentru a ordona în ordine descrescătoare vectorul de tip detecție D în funcție de pragul de corelație.

6. Eliminare non_maxime

Pentru calculul suprapunerii am folosit mai întâi o funcție ce calculează aria intersecției dintre două ferestre.

unsigned int arie_intersectie(unsigned int i, unsigned int j, unsigned int p, unsigned int q, unsigned int W_sablon, unsigned int H_sablon)

Parametrii funcției reprezintă linia și coloana colțului din stânga sus pentru două ferestre, precum și dimensiunile unui șablon, având dimensiunile șabloanelor putem calcula cu ușurință și restul colțurilor. (i și j sunt linia și coloana pentru prima fereastră, iar p și q sunt linia și coloana pentru a doua fereastră)

Pentru calculul arie am considerat toate cazurile când cele două ferestre se intersectează, în cazul care cele două nu se intersectează am returnat 0.

Funcția următoare calculează suprapunerea:

float suprapunere(unsigned int i,unsigned int j, unsigned int p, unsigned int q, unsigned int W_sablon, unsigned int H_sablon)

Parametrii folosiți sunt cu aceeași semnificație ca cei de la calculul intersecției.

În cazul în care aria intersecției este 0, suprapunerea va fi tot 0, altfel vom calcula cu ajutorul formulei suprapunerea dintre cele două ferestre.

Eliminarea non-maximelor se realizează în următoarea funcție:

void eliminare_non_maxime(detectie **D, unsigned int *k, unsigned int W_sablon, unsigned int H_sablon)

Parametrii funcției sunt vectorul de tip detecție D, numărul de elemente din vectorul D și dimensiunile șablonului. Vectorul D apelat în această funcție a fost sortat descrescător anterior în funcție de pragul de corelație, tabloul se va parcurge de la stânga la dreapta și se va calcula suprapunerea pentru toate perechile din vector, eliminându-se toate toate detecțiile ce se suprapun, din perechea (i,j) rămâne în vector elementul i, deoarece are pragul de corelație mai mare. Vom folosi funcția **realloc**, pentru a realoca memorie de fiecare dată când vom șterge din vector.

Ultima funcție este **eliberare_memorie** și eliberează memoria pentru toate variabilele folosite de-a lungul programului.

void eliberare_memorie (detectie **D, pixel ***M, pixel ***A, pixel ***S, pixel ***P, unsigned int W_imagine, unsigned int H_imagine, unsigned int W_sablon, unsigned int H_sablon)

7. Programul principal

În programul principal calea imaginii și a șabloanelor se va citi dintr-un fișier text, numit **fisier.txt,** iar numele fișierelor ce vor păstra imaginea colorată cu toate cele 10 șabloane, cea în care se aplică algoritmul de non-maxime, cele ce păstrează imaginile grayscale sunt denumite în interiorul programului principal.

Se vor citi căile fișierelor binare din fișierul text și se va afișa pe ecran ceea ce s-a citit din fișierul text, numele imaginii și a șabloanelor.

În continuare se vor calcula dimensiunile imaginii și dimensiunile unui șablon.

Se va aplica algoritmul de grayscale celor 10 șabloane și imaginii. Transformăm imaginea color într-o matrice, de două ori, matricea A și P, imaginea grayscale în matricea M și șablonul în matricea S.

Colorările se vor realiza pe două copii ale imaginii color transformate în matricele, A și P.

În următoarele linii de cod are loc aplicarea algoritmului de template matching rulând toate șabloanele, colorarea imaginii după aceasta, imaginea obținută numindu-se **coloraretotala.bmp.**

Se sortează vectorul D descrescător în funcție de corelație folosindu-ne de funcția **qsort**. Se apelează funcția de eliminare a non-maximelor, apoi cea de deliniarizare, pentru a afișa imaginea dorită, numită **nonmaxime.bmp.**

Se apelează funcția de eliberare a memoriei și se sterg de pe hard disk toate fișierele grayscale, deoarece nu mai avem nevoie de ele, se verifică dacă fișierele au fost șterse cu succes, afișându-se pe ecran rezultatul printr-un mesaj.