

Documentație proiect Programare Procedurală

Trepteanu Narcis-Lucian

Decembrie 2018

1. Conținutul arhivei.

Arhiva trimisă conține următoarele:

- Fișierele sursă *main.c*, *main_encryption.c*, *main_patternMatching.c*
- Fișierele sursă auxiliare *common.c*, *crypt.c* și *pattern.c*.
- Headerele *common.h*, *crypt.h* și *pattern.h*
- Fișierele txt și bmp necesare testării
 - Imaginea *peppers.bmp*
 - Fișierul text *secret_key.txt*
 - Fișierul text *encryptionPaths.txt* ce conține path-urile fișierelor necesare testării modulului de criptare
 - Fișierul text *matchingPaths.txt* ce conține path-urile fișierelor necesare testării modulului de pattern matching
 - Șabloanele *cifra0.bmp*, *cifra1.bmp* ... *cifra9.bmp*
 - Imaginea *test.bmp* pe care va fi rulată operația de matching
- Documentația proiectului.

2. Structura proiectului

Fișierele cod din arhivă au mai multe roluri. Sursa *main.c* conține codul programului care implementează cea de-a 11-a cerință a proiectului. Pentru ușurința depanării, cele 2 module vor avea și main-uri separate.

Sursa *main_encryption.c* conține implementarea cerințelor din primul modul, utilizând și funcțiile descrise în headerele *common.h* și *crypt.h*.

De asemenea, în *main_patternMatching.c* găsim implementarea cerințelor din cel de-al doilea modul, utilizând funcții din headerul *common.h*, cât și noile funcții specifice operației de template matching din *pattern.h*.

În headerele *common.h*, *crypt.h* și *pattern.h* regăsim antetele funcțiilor folosite pe parcursul rezolvării cerințelor, alături de scurte descrieri în ceea ce privește utilitatea și parametrii acestora. *Common.h* conține subprograme uzuale, (precum încărcarea în memorie a unei imagini) ce vor fi de folos în implementarea ambelor module.

În *common.c*, *crypt.c* și *pattern.c* regăsim implementările funcțiilor definite în headerele menționate anterior.

În cele ce urmează, vor fi prezentate funcțiile și structurile de date din ambele header, precum și utilizarea acestora în main-uri.

3. Headerul *common.h*

Structuri de date:

- **Bitmap :**
 - Înălțimea imaginii
 - Lățimea imaginii
 - **Pointer** de tip **unsigned char** către zona de memorie ce va conține headerul imaginii
 - **Pointer** de tip **unsigned int** către zona de memorie ce va conține forma liniarizată a pixelilor imaginii (se vor memora în ordinea în care apar în fișierul binar, deci bottom-top, left-right)

Funcții (ale căror implementări se găsesc în *common.c*):

- **int** `get_padding(int image_width)` primește ca parametru lățimea unei imagini și returnează paddingul acesteia.
- **Bitmap** `load_bitmap(char* source_file)` primește ca parametru path-ul fișierului bitmap ce va fi încărcat în memoria internă. Headerul se va reține ca atare, dimensiunile vor fi extrase din acesta, în timp ce forma liniarizată a imaginii se va reprezenta ca un vector de elemente de tip **unsigned int** în care primul byte nu contează, iar ceilalți 3 reprezintă valorile pe canalele RGB (în această ordine)

- **void free_bitmap(Bitmap bmp)** va elibera zonele de memorie către care pointează câmpurile bmp.header și bmp.linearized_bmp.
- **void write_bitmap(Bitmap bmp, char *dest_file)** va scrie la path-ul "dest_file" imaginea stocată intern în bmp. În cazul în care paddingul este diferit de 0, la finalul fiecărei linii se vor adăuga bytes cu valoarea 0.
- **unsigned int **load_array2D(Bitmap bmp)** este o funcție folosită mai ales în modulul de pattern matching. Ea preia o structură de tip **Bitmap** și va construi în memorie o matrice de elemente de tip unsigned int ce conține pixelii din forma liniarizată așa cum sunt ei dispuși în cadrul imaginii (top-bottom, left-right)

4. Headerul *crypt.h*

Funcții (ale căror implementări se găsesc în *crypt.c*):

- **unsigned int xor_shift(unsigned int *state)** va primi un pointer către variabila ce reține starea curentă a generatorului de numere pseudo-aleatoare. Returnează următoarea stare a generatorului (următorul număr generat)
- **unsigned int *encrypt(Bitmap bmp, char *key_file)** primește ca parametri reprezentarea internă a unui bitmap și path-ul fișierului ce conține cheile secrete R0 și SV.
 - Utilizând funcția xor_shift(), se vor genera $2 * \text{bmp.image_height} * \text{bmp.image_width}$ numere pseudo-aleatoare
 - A doua jumătate a numerelor generate este utilizată în cadrul implementării algoritmului Durstenfeld pentru generarea unei permutări aleatoare.
 - Se permută pixelii liniarizării imaginii bmp
 - Se aplică recurența de criptare

Funcția returnează un pointer către zona de memorie de la care se începe memorarea formei liniarizate a imaginii criptate.

- **void write_encrypted(char *image_file, char *dest_file, char* key_file)** va prelua în memoria internă imaginea bitmap stocată la path-ul dat de primul parametru, va aplica (utilizând funcția de mai sus) criptarea cu cheile de la path-ul dat de al treilea parametru și va scrie imaginea rezultată la path-ul dat de al doilea parametru.

- **unsigned int *decrypt(Bitmap bmp, char *key_file)** va primi ca parametri reprezentarea internă a imaginii criptate și path-ul fișierului ce conține cheile secrete. În cadrul funcției se va implementa “opusul” pașilor de la criptare, și anume:
 - Se vor genera $2 * \text{bmp.image_height} * \text{bmp.image_width}$ numere pseudo-aleatoare folosindu-se funcția *xor_shift()*
 - Se va genera prin algoritmul Durstenfeld o permutare aleatoare de lungime $\text{bmp.image_height} * \text{bmp.image_width}$
 - Se va calcula inversa permutării
 - Se face “inversarea” recurenței din criptare
 - Se permută pixelii la pozițiile lor inițiale, utilizând permutarea inversă generată

Funcția returnează un pointer către zona de memorie de la care se începe memorarea formei liniarizate a pixelilor imaginii decriptate.

- **void write_decrypted(char *source_file, char *decrypted_file, char *key_file)** va prelua în memoria internă imaginea bitmap stocată la path-ul “source_file”, va aplica procesul de decriptare folosind funcția de mai sus și va scrie imaginea rezultată la path-ul “decrypted_file”
- **void print_chi_square(char *source_file)** încarcă în memoria internă imaginea de la path-ul dat de parametru. Pentru fiecare culoare (în ordinea RGB), cu ajutorul formulei oferite și a unui vector de frecvență, se va afișa la consolă valoarea testului chi_square pe canalul respectiv.

5. Headerul pattern.h

Structuri de date:

- Window :
 - Linia de sus ce delimitează fereastra (coordonata y minimă)
 - Linia de jos ce delimitează fereastra (coordonata y maximă)
 - Linia din stânga ce delimitează fereastra (coordonata x minimă)
 - Linia din dreapta ce delimitează fereastra (coordonata x maximă)
 - Dimensiunea în pixeli a ferestrei
 - Media intensităților grayscale ale pixelilor ferestrei
 - Deviația standard a valorilor intensităților grayscale ale pixelilor din fereastră
 - Coeficientul de corelație cu un anume șablon

- Indicele șablonului cu care se face corelația (număr de la 0 la 9, pentru că lucrăm cu șabloane de cifre)

Funcții (implementate în pattern.c)

- **void write_grayscale(char *source_file, char *dest_file)** încarcă în memoria internă imaginea de la path-ul dat de "source_file". Se parcurg pixelii sub forma lor liniarizată, alterându-se valorile RGB astfel încât să fie egale și să dea valoarea intensității grayscale a pixelului. Folosind funcția *write_bmp()* se scrie imaginea obținută în memoria externă.
- **Window *cross_corelation(Bitmap image, Bitmap templateImg, int *howMany, double prag, int whichTemplate)** este cea mai voluminoasă funcție din cadrul modulului și implementează algoritmul de template matching între imaginea dată de primul parametru și șablonul dat de al doilea. Ea va furniza adresa de start a unui vector de tip Window ce conține matching-urile cu corelație mai mare decât pragul (prin return) dar și dimensiunea acestuia (prin intermediul parametrului howMany). Pașii din implementarea acesteia sunt după cum urmează.
 - Se încarcă atât imaginea cât și șablonul sub formă de matrice, folosind funcția *load_array2D()*
 - Se calculează media intensităților pixelilor din șablon
 - Folosind media, se calculează deviația standard a pixelilor din șablon
 - Imaginea peste care se caută șabloanele va fi parcursă ca o matrice normală, fixându-se colțul stânga-sus al ferestrei curente.
 - Se stabilesc cele 4 drepte ce delimitează fereastra și dimensiunea acesteia
 - Se calculează (ca în cazul șablonului) media intensităților grayscale
 - Se calculează (ca în cazul șablonului) deviația standard a pixelilor din fereastră
 - Se parcurg în paralel atât fereastra cât și șablonul, pentru a calcula coeficientul de corelație
 - Atunci când corelația depășește pragul, se adaugă fereastra curentă în vectorul soluție
 - Se actualizează parametrul *howMany* și se realocă optim memoria pentru vectorul soluție
 - Se eliberează memoria folosită de cele 2 reprezentări-matrice
- **void draw_single(Bitmap *bmp, Window window, unsigned int color)** este implementarea cerinței 8. Forma liniarizată a imaginii se transformă în forma matrice. Pozițiilor din matrice corespunzătoare conturului ferestrei li se vor

atribui culoarea dată ca parametru, și se va reveni la forma liniarizată. Cum, în implementarea de față, desenarea unui contur dreptunghiular implică transformarea din formă liniarizată în formă matrice, și transformarea inversă, desenarea unui singur dreptunghi nu e avantajoasă pentru situația de față (așadar această funcție nu va fi folosită).

- **void draw_rectangle(Bitmap *bmp, Window *windows, int howMany, unsigned int *colors)** va folosi principiul de mai sus pentru a desena “simultan” mai multe ferestre (chiar și de culori diferite). Cele *howMany* ferestre din vectorul ce începe la adresa *windows* se vor desena folosind culoarea *colors[fereastraCurenta.whichTemplate]*, unde *colors* este un vector constant ce conține culorile corespunzătoare șabloanelor (în exemplul dat are 10 elemente)
- **int cmp(const void *a, const void *b)** este funcția de comparare a două ferestre, descrescător în funcție de coeficientul de corelație
- **void sort_detections(Window *arr, int array_sz)** utilizează funcția *qsort()* cu criteriul de comparare *cmp* pentru a sorta vectorul de *array_sz* elemente stocat de la adresa *arr*.
- **int maxim(int a, int b)** returnează maximumul dintre cele două numere stocate pe tip **int** furnizate de parametri.
- **int minim(int a, int b)** returnează minimumul dintre cele două numere stocate pe tip **int** furnizate de parametri.
- **_Bool xIntersect(Window a, Window b)** stabilește dacă cele două ferestre furnizate prin intermediul parametrilor s-ar “intersecta” pe coordonata x.
- **_Bool yIntersect(Window a, Window b)** stabilește dacă cele două ferestre furnizate prin intermediul parametrilor s-ar “intersecta” pe coordonata y.
- **_Bool intersects(Window a, Window b)** stabilește (folosind cele două funcții de mai sus) dacă ferestrele furnizate de cei doi parametri se intersectează.
- **Window get_intersection(Window a, Window b)** va returna fereastra dată de intersecția dintre a și b (în cazul în care aceasta există). De menționat că în cadrul acestei funcții ne interesează doar cele patru drepte ce delimitează conturul, nu și restul câmpurilor structurii.

- **Window *non_maximal_erase(Window *allCorel, int len, double minSupr, int *newLen)** implementează algoritmul de eliminare al non-maximelor:
 - Se sortează vectorul *allCorel* (ce conține toate detecțiile de corelație > prag) descrescător după coeficientul de corelație.
 - Se inițializează un vector boolean *marked*, în care *marked[i]* semnifică faptul că *allCorel[i]* trebuie eliminată
 - Se parcurge vectorul *allCorel*, eliminându-se perechile (i, j) pentru care $i < j$ și suprapunerea dintre ele depășește *minSupr*
 - Se adaugă în vectorul *rectangles* ferestrele care nu au fost eliminate (*allCorel[i]* pentru care *marked[i]* este 0)
 - Se realocă memoria optim pentru memorarea vectorului soluție
 - Se actualizează parametrul *newLen* (dimensiunea vectorului soluție)

6. Fișierul sursă *main encryption.c*

În cadrul acestei surse sunt rezolvate (utilizând funcții din headere) cerințele care țin de modulul de criptare.

Din fișierul text encryptionPaths.txt se vor citi:

- Path-ul fișierului imagine ce va fi criptat
- Path-ul unde se va scrie imaginea rezultată în urma criptării
- Path-ul unde se va scrie imaginea după procesul de decriptare
- Path-ul fișierului ce conține cheile secrete

În urma unor apeluri de funcție, se vor salva în memoria externă rezultatul criptării imaginii inițiale și rezultatul decriptării imaginii criptate obținute anterior.

Mai apoi se vor afișa la consolă valorile testului chi-square pe canalele RGB pentru imaginea criptată și pentru imaginea decriptată (inițială).

Pe mașina pe care a fost testat programul și pe exemplul dat (imaginea peppers.bmp), rezultatele testului chi-square au fost:

- Imaginea originală/decriptată:
 - R: 529917.104800
 - G: 606801.969600
 - B: 1053070.308000
- Imaginea criptată:
 - R: 255.057600

- G: 231.340800
- B: 279.216800

7. Fișierul sursă main_patternMatching.c

În cadrul acestei surse sunt rezolvate (utilizând funcții din headere) cerințele care țin de modulul de recunoaștere de imagini.

Din fișierul `text_matchingPaths.txt` se vor citi:

- Path-ul imaginii pe care se vor căuta șabloanele
- Path-ul la care se va salva imaginea de test după transformarea în grayscale
- Path-ul la care se va scrie imaginea după desenarea ferestrelor de detecții de șabloane
- Numărul de șabloane utilizate
- Path-urile șabloanelor
- Triplete ce reprezintă valorile RGB corespunzătoare culorilor cu care se vor desena ferestrele pentru fiecare sablon.

Este de menționat faptul că fiecare din datele enumerate mai sus se află pe câte o linie separată.

Fiecărui șablon *i* se va mai asocia un fișier care reprezintă imaginea obținută în urma transformării bitmap-ului inițial în grayscale. În implementarea de față denumirile șabloanelor în grayscale se obțin din denumirile inițiale la care se adaugă sufixul "GRAY".

În vectorul `colors` se vor memora culorile din fișierul de citire, reprezentate sub forma unor variabile `unsigned int` în care ultimii 3 bytes reprezintă valorile RGB (de la stanga la dreapta).

Pentru fiecare șablon în parte se va apela funcția `cross_corelation` pentru a rula algoritmul de pattern matching. Ferestrele/detecțiile obținute pentru fiecare șablon se vor adăuga în același vector *allCorel*.

Se apelează funcția *non_maximal_erase* pentru a rula algoritmul de eliminare a non-maximelor.

Într-un final, peste Bitmap-ul generat de imaginea inițială se vor desena detecțiile rămase, printr-un apel al funcției *draw_rectangles*, iar rezultatul va fi scris în memoria externă printr-un apel al funcției *write_bmp*.

Pe mașina pe care a fost scris și testat codul, imaginea obținută pe exemplul dat este:



8. Fișierul sursă main.c

Acest fișier sursă combină rezolvările prezentate în *main_encryption.c* și *main_patternMatching.c*.

Corpul principal al funcției conține doar 2 apeluri de funcție, pentru *encryption* și *patternMatching*.

Procedura *encryption* conține aceeași implementare ca funcția principal din *main_encryption.c*, în timp ce procedura *patternMatching* conține aceeași implementare ca main-ul din *main_patternMatching.c*.

9. Coreșpondența dintre cerințe și funcții

- i. Generatorul de numere este implementat în cadrul funcției *xor_shift()*
- ii. Încărcarea imaginii în formă liniarizată în memorie este implementată în *load_bitmap()*
- iii. Scrierea unei imagini din memoria internă în fișier extern se face prin funcția *write_bmp()*
- iv. Criptarea imaginii se face prin funcția *write_encrypted()* ce utilizează și funcția *encrypt()*
- v. Decriptarea imaginii se face prin funcția *write_decrypted()* ce utilizează și funcția *decrypt()*
- vi. Valorile testului chi-square se afișează la consola prin apelurile funcției *print_chi_square()*
- vii. Implementarea operației de pattern matching este în funcția *cross_corelation()*
- viii. Desenarea unui contur dreptunghiular se poate face prin apelul funcției *draw_single()*
- ix. Funcția *sort_detections()* sortează detecțiile descrescător după coeficientul de corelație, utilizând comparatorul *cmp()*
- x. *non_maximal_erase()* realizează eliminarea non-maximelor
- xi. Ultima cerință este implementată în fișierul sursă *main.c*