Recuperare a imaginilor JPEG

Popescu Alexandru

TAID

2025

1 Introducere

1.1 Contextul problemei

În era digitală, imaginile sunt o parte importantă a vieții de zi cu zi, fiind utilizate pentru păstrarea amintirilor, comunicare și chiar ca dovezi în procese judiciare. Formatul **JPEG** (Joint Photographic Experts Group) este unul dintre cele mai utilizate formate pentru stocarea imaginilor, datorită echilibrului dintre calitate și compresie.

Cu toate acestea, fișierele JPEG nu sunt protejate împotriva ștergerilor accidentale sau a coruperii datelor. În momentul în care un fișier este șters, sistemul de fișiere marchează spațiul respectiv ca disponibil, dar conținutul efectiv al fișierului poate rămâne intact până la rescrierea sa. Acest aspect oferă o oportunitate pentru recuperarea datelor, mai ales dacă fișierele nu au fost fragmentate în mod excesiv.

Una dintre dificultăți constă în faptul că fișierele JPEG nu au o dimensiune fixă sau un format strict liniar. Ele sunt organizate în segmente, fiecare fiind delimitat de markeri specifici. Dacă acești markeri sunt deteriorați sau lipsesc, devine dificil să determinăm unde începe și unde se termină o imagine.

Problema principală în recuperarea imaginilor șterse este identificarea corectă a datelor care aparțin unui fișier complet, mai ales în situațiile în care structura poate fi parțial coruptă sau fragmentată. Acest proiect își propune să abordeze această problemă printr-o metodă bazată pe analiza markerilor JPEG, care delimitează secțiunile importante ale fișierului.

1.2 Scopul proiectului

Scopul acestui proiect este de a dezvolta un algoritm capabil să detecteze, să grupeze și să reconstruiască fișiere JPEG șterse, pornind de la date brute (dump-uri binare) extrase de pe dispozitive de stocare.

Metoda folosită în acest proiect se bazează pe identificarea **markerilor JPEG**—secvențe binare specifice formatului, cum ar fi **SOI (Start of Image)** și **EOI (End of Image)**—care definesc limitele unui fișier JPEG. Abordarea presupune:

1. **Scanarea întregului spațiu de date binare** pentru a identifica toți markerii prezenți, fără a stabili inițial ce marker aparține cui.
2. **Gruparea markerilor detectați** pe baza proximității lor, pornind de la presupunerea că markerii aflați în apropiere spațială (în cadrul datelor brute) aparțin aceluiași fișier.
3. **Parsarea conținutului markerilor** pentru a verifica structura internă a grupurilor formate și pentru a valida că acestea pot constitui fișiere JPEG valide.

Această abordare are avantajul de a funcționa chiar și în cazul fișierelor parțial corupte sau incomplete, deoarece identifică segmente utile de date și încearcă să le reconstituie logic.

2 Fundamente teoretice

2.1 Structura fișierelor JPEG

Formatul JPEG (Joint Photographic Experts Group) este unul dintre cele mai utilizate formate de imagine datorită capacității sale de a comprima datele într-un mod eficient, menținând o calitate vizuală ridicată. Această compresie se bazează pe algoritmi specifici, cum ar fi transformata cosinus discretă (DCT) și codarea Huffman, care permit reducerea dimensiunii fișierului prin eliminarea redundanței datelor.

Un fișier JPEG este alcătuit dintr-o serie de segmente, fiecare având un rol bine definit. Fiecare segment este identificat printr-un marker de doi octeți, reprezentat de un cod hexazecimal care începe întotdeauna cu valoarea **FF**. Marcatorii definesc limitele și tipul fiecărui segment, indicând, de exemplu, începutul sau sfârșitul imaginii, precum și informațiile despre compresie și dimensiuni.

Marcatorul **SOI (Start of Image)**, reprezentat prin secvența **FFD8**, marchează începutul fișierului JPEG, în timp ce **EOI (End of Image)**, identificat prin secvența **FFD9**, marchează sfârșitul acestuia. Între acești markeri, fișierul conține alte segmente, cum ar fi **DQT (Define Quantization Table)**, utilizat pentru definirea tabelelor de cuantizare, și **DHT (Define Huffman Table)**, care conține informații despre tabelele de codare Huffman.

Un aspect important al formatului JPEG este acela că nu impune o dimensiune fixă pentru segmente sau o ordine strictă a acestora, deși există reguli generale care trebuie respectate. Această flexibilitate face ca formatul să fie eficient, dar complică procesul de recuperare, deoarece algoritmul trebuie să detecteze markerii și să analizeze relațiile dintre segmente pentru a reconstrui imaginea.

2.2 Gestionarea fișierelor șterse în sistemele de fișiere

Pentru a înțelege cum pot fi recuperate fișierele JPEG șterse, este esențial să analizăm modul în care sistemele de fișiere gestionează ștergerea datelor. În majoritatea sistemelor de fișiere, cum ar fi **FAT32** și **NTFS**, ștergerea unui fișier nu implică eliminarea imediată a datelor de pe disc. În schimb, sistemul marchează spațiul ocupat de acel fișier ca fiind disponibil pentru utilizare, permițând astfel rescrierea lui.

De exemplu, într-un sistem **FAT32**, intrarea corespunzătoare fișierului din tabelul de alocare este ștearsă, dar datele rămân prezente fizic pe disc până când sunt suprascrise de alte fișiere. În cazul sistemului **NTFS**, fișierele șterse sunt gestionate printr-un jurnal de tranzacții care păstrează informații despre modificările recente.

Această abordare oferă o oportunitate pentru recuperarea fișierelor, dar vine și cu riscuri. Dacă datele sunt fragmentate sau au fost parțial suprascrise, procesul de recuperare poate deveni mai dificil. Fragmentarea este deosebit de problematică pentru fișierele JPEG, deoarece acestea pot fi stocate în bucăți separate pe disc, ceea ce face dificilă reconstituirea lor completă.

2.3 Identificarea markerilor JPEG

Unul dintre aspectele esențiale ale procesului de recuperare implementat în acest proiect este identificarea markerilor JPEG în datele brute. Acești markeri sunt punctele cheie care permit algoritmului să detecteze începutul și sfârșitul fișierelor, precum și să analizeze structura internă a acestora.

În timpul procesului de scanare a datelor, markerii au fost identificați prin căutarea secvențelor binare specifice, cum ar fi **FFD8** pentru începutul imaginii și **FFD9** pentru sfârșit. Pe lângă acești markeri principali, algoritmul a detectat și alți markeri intermediari, cum ar fi **FFC0** și **FFC2**, care conțin informații despre dimensiunea imaginii și parametrii de compresie.

Pentru a verifica validitatea grupurilor formate, a fost necesară și parsarea datelor asociate fiecărui marker. De exemplu, markerii **DQT** și **DHT** au fost analizați pentru a verifica dacă tabelele de cuantizare și de codare Huffman sunt prezente și complete. În absența acestor informații, fișierele rezultate pot fi incomplete sau corupte.

2.4 Fragmentarea datelor

Un aspect important în recuperarea fișierelor JPEG șterse este fragmentarea datelor. Într-un mediu ideal, fișierele sunt stocate secvențial pe disc, ceea ce face ca recuperarea să fie simplă. Totuși, în practică, multe fișiere sunt fragmentate, mai ales pe discuri utilizate intens, unde spațiile libere sunt dispersate.

Fragmentarea complică procesul de recuperare, deoarece markerii pot fi separați de datele pe care le definesc. De exemplu, un marker **SOI** poate fi urmat de date incomplete sau poate lipsi un marker **EOI**, caz în care algoritmul trebuie să încerce să identifice finalul logic al fișierului pe baza structurii sale interne.

În acest proiect, problema fragmentării a fost abordată prin analizarea proximității markerilor detectați. S-a presupus că markerii aflați în apropiere spațială pe disc fac parte, cel mai probabil, din același fișier. Deși această metodă nu garantează succesul în toate cazurile, ea s-a dovedit suficient de eficientă pentru a recupera imagini în majoritatea scenariilor testate.

3 Metodologie

3.1 Setarea mediului de lucru

Pentru a implementa și testa metoda de recuperare a fișierelor JPEG șterse, a fost necesară crearea unui mediu controlat, care să reproducă scenariile reale întâlnite în procesul de analiză criminalistică. În acest scop, am creat o imagine virtuală de disc cu o dimensiune de **500 KB**, inițializată cu valori nule, formatată folosind sistemul de fișiere **FAT32** și pregătită pentru a permite montarea și analiza în timp real.

3.1.1 Crearea imaginii de disc

Primul pas a fost generarea unui fișier binar care să simuleze un dispozitiv de stocare. Acest lucru a fost realizat cu comanda:

```
dd if=/dev/zero of=/home/alex/proiect/memorie.img bs=512 count=1000
```

Această comandă a creat un fișier numit **memorie.img** cu o dimensiune totală de **500 KB** (512 bytes x 1000 blocuri). Fișierul a fost umplut inițial cu valori **zero**, simulând un dispozitiv de stocare gol, ceea ce mi-a permis să am control total asupra conținutului pe care urma să-l scriu și să-l șterg.

3.1.2 Asocierea imaginii cu un dispozitiv de loopback

Pentru a trata fișierul **memorie.img** ca pe un dispozitiv real, am folosit comanda:

```
sudo losetup -f --show /home/alex/proiect/memorie.img
```

Această comandă a alocat automat un dispozitiv **loopback** disponibil (în cazul meu, **/dev/loop38**) și l-a asociat cu fișierul creat. Această asociere a fost esențială pentru a putea aplica operațiuni specifice unui sistem de fișiere real.

3.1.3 Formatarea imaginii cu sistemul de fișiere FAT32

După ce fișierul a fost asociat ca un dispozitiv de tip bloc, a fost formatat folosind sistemul de fișiere **FAT32**:

```
sudo mkfs.vfat -F 32 /dev/loop38
```

Sistemul **FAT32** a fost ales deoarece este unul dintre cele mai comune sisteme de fișiere, utilizat frecvent pe stick-uri USB și carduri SD. De asemenea, acest format este relativ simplu, ceea ce a facilitat analiza manuală și automatizată a structurii datelor.

3.1.4 Montarea și demontarea imaginii

Pentru a putea lucra cu fișierele din această imagine, am montat sistemul de fișiere în mod accesibil pentru utilizatorul curent:

```
sudo mount /dev/loop38 /media/alex/memorie -o uid=$(id -u),gid=$(id -g),umask=000
```

Această comandă a asigurat că am avut permisiuni complete de citire și scriere asupra imaginii montate, permițând adăugarea și ștergerea fișierelor pentru simularea unui scenariu real. După finalizarea operațiunilor, imaginea a fost demontată cu:

```
sudo umount /media/alex/memorie
```

Iar dispozitivul **loopback** a fost eliberat cu:

```
sudo losetup -d /dev/loop38
```

3.2 Analiza manuală a imaginii de disc

Un pas important în procesul de dezvoltare a metodei a fost analiza manuală a conținutului imaginii pentru a înțelege modul în care datele erau stocate și organizate. Această analiză a fost realizată prin exportarea conținutului imaginii în format hexazecimal:

```
hexdump -C /home/alex/proiect/memorie.img > /home/alex/proiect/memorie.txt
```

Această comandă a produs o reprezentare hexazecimală a conținutului imaginii, salvând rezultatul într-un fișier text ușor de analizat. Acest format a permis examinarea manuală a markerilor JPEG, identificarea secvențelor relevante și verificarea structurii fișierelor în timpul dezvoltării algoritmului.

Prin această etapă, am avut posibilitatea de a studia comportamentul datelor în momentul în care un fișier era creat, șters și eventual suprascris, oferindu-mi un punct de plecare pentru dezvoltarea scriptului Python de recuperare.

3.3 Observații privind mediul de testare

Mediul configurat a oferit mai multe avantaje în procesul de implementare. În primul rând, folosirea unei imagini de disc virtuale a permis simularea unui scenariu realist fără a risca pierderea datelor importante de pe un dispozitiv fizic. În al doilea rând, utilizarea sistemului de fișiere **FAT32** a simplificat procesul de analiză, datorită structurii sale relativ simple și ușor de interpretat.

Un alt avantaj al acestui mediu a fost flexibilitatea sa. Imaginea de disc putea fi resetată rapid, iar conținutul său putea fi analizat atât manual, cât și automat, fără a fi necesare operațiuni complicate. Totodată, lucrul cu date brute a permis o mai bună înțelegere a modului în care fișierele sunt stocate și organizate, facilitând dezvoltarea algoritmilor pentru identificarea markerilor JPEG.

3.4 Parsarea markerilor JPEG

Un pas esențial în procesul de recuperare a imaginilor JPEG constă în detectarea și parsarea markerilor specifici acestui format. Markerii sunt folosiți pentru a delimita structura fișierelor și pentru a identifica informațiile necesare reconstruirii acestora. În acest proiect, markerii pe care îi analizez sunt **SOI**, **APP**, **DQT**, **SOF0**, **DHT**, **SOS** și **EOI**.

Pe măsură ce analizez datele brute, adaug fiecare marker detectat într-o listă globală. Această listă include informații despre poziție, lungime, nume și conținut. Prin organizarea datelor în acest mod, pot accesa rapid markerii și pot verifica relațiile dintre ei, mai ales în cazul grupurilor formate pe baza proximității.

Pentru fiecare marker identificat, salvez atât informațiile esențiale, cât și versiunea sa brută (**raw**). Salvarea datelor în format brut îmi permite, de asemenea, să reconstruiesc fișierul fără pierderi de informație. Acest aspect este important mai ales pentru markerii mai complexi, cum sunt **DQT** și **DHT**, care conțin informații esențiale pentru decodarea imaginii.

3.4.1 Marker-ul APPx

Markerul **APPx** este utilizat pentru a stoca date suplimentare, cum ar fi metadatele EXIF sau profilele ICC. Deși aceste informații nu sunt esențiale pentru afișarea imaginii, ele pot conține detalii importante despre originea fișierului sau setările camerei foto.

În implementarea mea, mă interesează în principal poziția markerului și lungimea secțiunii asociate. Aceste informații sunt utile atât pentru verificarea continuității datelor, cât și pentru reconstrucția fișierului. De asemenea, salvez și conținutul markerului pentru a-l analiza ulterior, dacă este necesar.

Procesarea acestui marker se face prin citirea lungimii secțiunii și extragerea datelor asociate. Deoarece markerii **APPx** pot avea conținut variabil, păstrez secțiunea brută pentru mă asigura că pot reconstrui imaginea.

3.4.2 Marker-ul DQT

Marker-ul **DQT** (Define Quantization Table) este esențial pentru fișierele JPEG, deoarece definește tabelele de cuantizare utilizate în procesul de compresie. Aceste tabele determină modul în care coeficienții de frecvență sunt aproximați în timpul compresiei cu transformata cosinus discretă (**DCT**).

Într-un fișier JPEG, tabelele de cuantizare sunt organizate sub formă de matrice 8x8, iar ordinea în care valorile sunt stocate respectă un model specific, numit **zig-zag scan**. Acest model permite optimizarea compresiei, grupând coeficienții importanți la începutul secvenței.

În implementare, marker-ul **DQT** este detectat și procesat prin citirea lungimii secțiunii și extragerea datelor necesare construirii tabelului de cuantizare. După extragerea datelor, construiesc tabelul de cuantizare pornind de la ordinea zig-zag specifică formatului JPEG. Această ordonare este necesară deoarece datele sunt stocate într-o secvență liniară, iar tabelul trebuie reconstruit într-un aranjament bidimensional de 8x8.

Pentru fiecare valoare din secvență, o inserez în tabelul corespunzător, urmând indicii predefiniți ai scanării în zig-zag. Această etapă asigură restaurarea fidelă a structurii tabelului, care poate fi folosit ulterior pentru validarea markerilor sau reconstrucția fișierului.

Ca și în cazul marker-ului **APP**, salvez informațiile într-o listă globală, păstrând atât datele interpretate (cum ar fi tabelul reconstruit), cât și versiunea brută a marker-ului.

3.4.3 Marker-ul SOF0

Marker-ul **SOF0** (Start of Frame 0) este unul dintre cei mai importanți markeri dintr-un fișier JPEG, deoarece definește structura de bază a imaginii. Acesta conține informații despre dimensiunile imaginii, precizia datelor și modul în care sunt organizate componentele de culoare. În standardul JPEG, marker-ul **SOF0** este utilizat pentru compresia bazată pe **DCT (Discrete Cosine Transform)** și este specific pentru modul **Baseline**—unul dintre cele mai comune formate de codare JPEG.

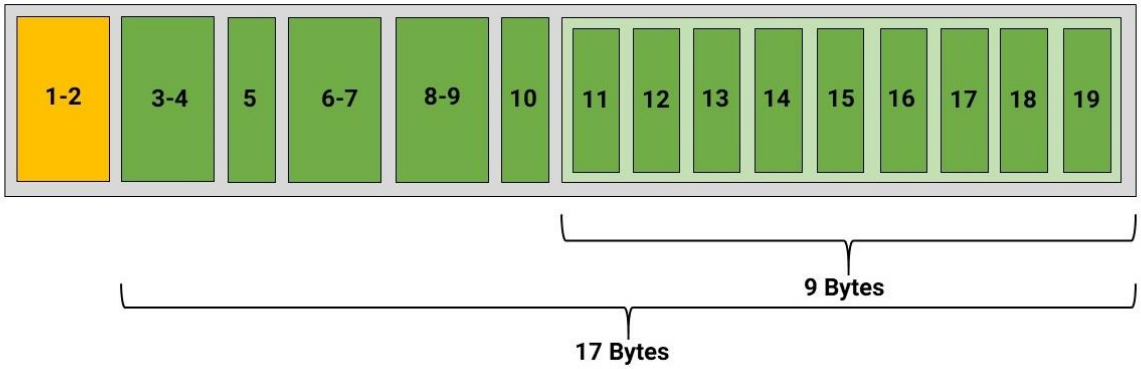

Figura 1: segmentul SOF0

În implementare, marker-ul **SOF0** este detectat și procesat pentru a extrage informațiile esențiale despre imagine. Primul pas este citirea lungimii secțiunii, care variază în funcție de numărul de componente de culoare incluse în imagine. Apoi, se extrag informațiile despre **precizie**, **înălțime** și **lățime**, care definesc dimensiunile imaginii și numărul de biți folosiți pentru fiecare eșantion. În mod obișnuit, fișierele JPEG utilizează o precizie de **8 biți**, suficientă pentru reprezentarea imaginilor color la o calitate ridicată.

Un aspect important al marker-ului **SOF0** este descrierea componentelor de culoare. Aceste componente sunt folosite pentru reprezentarea informațiilor de luminozitate (**Y**) și crominanță (**Cb** și **Cr**) în cazul imaginilor color, sau pentru nivelurile de gri în cazul imaginilor monocrome.

În cadrul implementării, fiecare componentă este procesată individual, extrăgându-se următoarele informații:

- **ID-ul componentei** – identificator numeric care definește rolul componentei (de exemplu, **1 pentru Y**, **2 pentru Cb** și **3 pentru Cr**).
- **Factorul de subeșantionare** – definește raportul dintre rezoluțiile orizontale și verticale ale componentei, utilizat pentru reducerea dimensiunii fișierului prin eșantionarea diferită a crominanței față de luminozitate.
- **Tabelul de cuantizare** – indică tabela de cuantizare utilizată pentru compresia fiecărei componente, făcând legătura cu marker-ul **DQT**.

Toate aceste informații sunt salvate într-o structură organizată, permițând analizarea detaliată a imaginii și validarea grupurilor detectate ulterior.

3.4.4 Marker-ul DHT

Marker-ul **DHT** (Define Huffman Table) joacă un rol esențial în procesul de compresie al fișierelor JPEG. Acesta definește tabelele Huffman folosite pentru codificarea entropiei, o metodă care reduce dimensiunea fișierului prin reprezentarea frecventelor mai mari cu coduri mai scurte și a celor mai rare cu coduri mai lungi. Fișierele JPEG utilizează două tipuri de tabele Huffman:

- **DC (Direct Current)** – folosite pentru coeficienții de frecvență joasă (luminozitate de bază).
- **AC (Alternating Current)** – folosite pentru coeficienții de frecvență mai mare (detaliile imaginii).

În implementare, marker-ul **DHT** este detectat și procesat pentru a extrage informațiile necesare construirii tabelelor Huffman. Primul pas constă în determinarea lungimii secțiunii și extragerea datelor brute pentru păstrarea fidelității informației. Apoi, marker-ul este analizat pentru a identifica componenta imaginii (luminozitate **Y** sau crominanță **Cb/Cr**) și tipul tabelului (**DC** sau **AC**). Aceste informații sunt esențiale pentru asocierea corectă a tabelelor cu componentele procesate anterior în marker-ul **SOF0**.

Un aspect important al procesării marker-ului **DHT** este reconstruirea tabelelor Huffman pe baza informațiilor despre lungimile codurilor și simbolurile asociate. În acest scop, lungimile codurilor sunt analizate pe 16 nivele, iar simbolurile sunt extrase secvențial. Aceste date sunt apoi folosite pentru a genera codurile binare corespunzătoare, urmând principiul codificării Huffman, în care codurile sunt generate în ordine crescătoare de lungime. Funcția **`generare_codare`** se ocupă de această etapă, creând codurile Huffman pe baza structurii tabelelor. Codurile rezultate sunt asociate simbolurilor corespunzătoare și salvate într-o structură de dicționar, permițând accesul rapid la datele de codificare.

Marker-ul **DHT** este adăugat în lista globală împreună cu informațiile despre componenta asociată, tipul de tabel (DC sau AC), structura tabelului Huffman și codurile generate.

3.4.5 Marker-ul SOS

Marker-ul **SOS** (Start of Scan) este un punct critic în structura fișierelor JPEG, deoarece marchează începutul datelor comprimate ale imaginii. Acest marker indică ce componente sunt incluse în scanarea curentă și specifică tabelele Huffman utilizate pentru decodare. Spre deosebire de markerii anteriori, care definesc metadate și parametrii structurali, marker-ul **SOS** este urmat direct de datele efective ale imaginii, codificate conform tabelelor definite anterior prin markerii **DHT** și **DQT**. Acest aspect îl face esențial pentru identificarea începutului datelor comprimate și asocierea lor cu parametrii corespunzători.

În implementarea mea, marker-ul **SOS** este detectat și procesat pentru a extrage informațiile despre componentele incluse în scanare și tabelele Huffman asociate acestora. Primul pas constă în citirea lungimii secțiunii și a numărului de componente. Fiecare componentă este apoi analizată pentru a determina ID-ul său și indexul tabelelor Huffman utilizate pentru coeficienții **DC** și **AC**. Datele extrase din marker-ul **SOS** sunt salvate în lista globală împreună cu informațiile despre poziție, lungime, numărul de componente și tabelele asociate.

3.5 Gruparea și validarea markerilor

3.5.1 Funcția `asamblareImagine`

Funcția `asamblareImagine` este responsabilă pentru gruparea markerilor din jurul unui marker **SOI** (Start of Image) și încercarea de a construi o imagine validă. Procesul începe prin inițializarea unei structuri goale pentru o imagine, care conține câmpuri pentru markerii principali si datele brute comprimate.

Selecția markerilor relevanți

În această etapă, markerii sunt căutați în apropierea poziției marker-ului **SOI**. Se folosește o distanță de **1024 octeți** ca limită pentru proximitate, presupunând că markerii care alcătuiesc o imagine validă vor fi relativ apropiați în memorie. Markerii identificați sunt adăugați în structura imaginii doar dacă respectă restricțiile definite în variabila `frecventa_markeri`, care stabilește numărul maxim de apariții permise pentru fiecare tip de marker. De exemplu, un fișier JPEG valid poate avea maximum **1 marker APP**, **2 markeri DQT**, **4 markeri DHT** etc.

3.5.2 Funcția `validareImagine`

Funcția `validareImagine` este responsabilă pentru verificarea consistenței și validității structurii markerilor dintr-o imagine grupată. Prima parte a funcției verifică prezența markerilor esențiali. Dacă markerii obligatorii precum **APP**, **DQT**, **SOF0**, **DHT**, **SOS** sau **EOI** lipsesc, imaginea este considerată invalidă.

Verificarea componentelor

Validarea continuă prin analiza detaliată a componentelor imaginii, bazându-se pe numărul acestora:

- **Imagini monocrome (1 componentă)** – Se verifică asocierea corectă a tabelelor de cuantizare (**DQT**) și Huffman (**DHT**) cu componenta **Y**. De asemenea, se asigură că numărul componentelor din marker-ul **SOS** corespunde cu definiția din **SOF0**.
- **Imagini color (3 componente)** – Se verifică existența componentelor **Y**, **Cb** și **Cr**, împreună cu tabelele lor asociate. Totodată, se analizează relația dintre tabelele **DC** și **AC** pentru fiecare componentă.

Testele logice

Validarea include și verificări logice, cum ar fi coerența între tabelele Huffman (**DHT**) și componentele definite în **SOF0**. De exemplu, o imagine cu 3 componente trebuie să aibă exact 4 tabele Huffman (2 pentru **Y** și 2 pentru **Cb/Cr**). Dacă aceste condiții nu sunt îndeplinite, imaginea este marcată ca invalidă.

3.6 Observații asupra procesului de grupare și validare

Procesul de grupare și validare descris aici este optimizat pentru fișiere JPEG standard, dar poate gestiona și cazuri parțial corupte sau incomplete. Un aspect important este flexibilitatea algoritmului, care utilizează atât analiza proximității markerilor, cât și verificări detaliate ale structurii interne pentru a asigura validitatea imaginilor detectate. Totuși, această metodă are și limite. Dacă markerii sunt sever fragmentați sau lipsesc secțiuni esențiale, imaginea poate fi considerată invalidă, chiar dacă anumite date sunt recuperabile. Acest lucru subliniază importanța utilizării unor metode suplimentare, cum ar fi analiza frecvențelor binare sau reconstruirea probabilistică a datelor lipsă.

3.8 Scrierea markerilor în fișiere JPEG

După ce markerii sunt grupați și validați, aceștia sunt organizați într-o secvență care respectă structura standard a unui fișier JPEG. Această secvență este apoi scrisă într-un fișier nou pentru fiecare imagine detectată. Procesul de scriere a fișierelor JPEG finalizează întregul algoritm, generând imagini valide care pot fi afișate sau analizate ulterior. Un aspect important al acestei etape este păstrarea ordinii corecte a markerilor și verificarea datelor brute înainte de scriere. În cazul în care markerii sunt incompleți sau ordinea lor nu respectă standardul JPEG, fișierul rezultat poate fi corupt sau imposibil de citit. Metoda este eficientă și directă, însă depinde în mare măsură de validitatea grupurilor formate anterior. Orice eroare în identificarea markerilor sau în validarea lor poate duce la fișiere incomplete sau incorecte.

4 Rezultate și analiză

4.1 Descrierea experimentului

Pentru a evalua eficiența algoritmului implementat, am creat un mediu de testare controlat în care două fișiere JPEG, reprezentând imagini distincte (**Lena** și o imagine cu o floare), au fost scrise în memoria simulată. După scriere, fișierele au fost șterse, iar în aceeași memorie a fost scris un fișier de tip DOCX. Această configurare a permis observarea modului în care datele șterse sunt gestionate în sistemul de fișiere FAT32 și a demonstrat că, după ștergere, datele fișierelor JPEG nu sunt efectiv eliminate, ci marcate ca spațiu liber, ceea ce face posibilă recuperarea lor. Scrierea ulterioară a fișierului DOCX a continuat din punctul unde se încheiaseră datele precedente, lăsând o mare parte din datele imaginilor intacte.

4.2 Rezultate obținute

După aplicarea algoritmului de recuperare, am reușit să identific markerii specifici pentru ambele fișiere JPEG șterse. Aceștia au fost grupați, validați și utilizați pentru reconstrucția imaginilor, rezultând două fișiere JPEG valide. Ambele imagini recuperate au putut fi deschise și afișate fără erori, ceea ce confirmă că markerii identificați erau compleți și corect asamblați. Fișierul DOCX scris

ulterior nu a afectat în mod semnificativ integritatea datelor JPEG, ceea ce evidențiază faptul că ștergerea și rescrierea parțială permit, în multe cazuri, recuperarea completă a fișierelor șterse.

5 Concluzii

Proiectul de recuperare a fișierelor JPEG șterse a demonstrat că utilizarea markerilor specifici formatului, combinată cu analiza logică și validarea datelor, poate oferi o metodă eficientă de reconstrucție a imaginilor digitale. Prin identificarea și gruparea markerilor precum **SOI**, **DQT**, **SOF0**, **SOS** și **EOI**, algoritmul propus a reușit să recupereze complet două fișiere JPEG șterse, chiar și în prezența unei rescrieri parțiale. Rezultatele obținute subliniază faptul că datele șterse rămân intacte pe disc până la momentul rescrierii lor complete, iar analiza markerilor poate oferi o abordare robustă pentru identificarea și reconstrucția fișierelor valide. Totuși, eficiența metodei depinde de integritatea markerilor și de gradul de fragmentare a fișierelor, ceea ce limitează aplicabilitatea în scenarii mai complexe.

6 Bibliografie

1. Skyler Rankin. (n.d.). **JPEG Decoder**. Disponibil pe GitHub. https://github.com/eilam-ashbell/jpeg-dump.git

Anexa

```
import struct
import numpy as np

# Deschiderea fișierului binar în modul citire
with open("./memorie.img", 'rb') as f:
    memorie = f.read()

# Lista markeri
markeri = [0xD8, 0xE0, 0xE1, 0xE2, 0xDB, 0xC0, 0xC4, 0xDA, 0xD9]

# Markerii identificati
markeri_gasiti = []

def SOI(index):
    # RAW
    raw = memorie[index: index + 2]

    markeri_gasiti.append({
        'marker': 0xFFD8,
        'nume': 'SOI',
        'pozitie': index,
        'raw': raw
    })

def APP(index, marker):
    # Aflăm lungimea secțiunii (2 octeți după etichetă)
    lungime = struct.unpack('>H', memorie[index + 2: index + 4])[0]

    # RAW
    raw = memorie[index: index + 2 + lungime]

    # Extragem datele din secțiune
    continut = memorie[index + 4: index + 2 + lungime]

    # Salvăm informațiile despre etichetă
    markeri_gasiti.append({
        'marker': 0xFF00 + marker,
        'nume': 'APP',
        'pozitie': index,
        'lungime': lungime,
        'continut': continut,
        'raw': raw
    })
```

```
def DQT(index):
    # Aflăm lungimea secțiunii (2 octeți după etichetă)
    lungime = struct.unpack('>H', memorie[index + 2: index + 4])[0]

    if lungime != 67:
        return

    # RAW
    raw = memorie[index: index + 2 + lungime]

    # Componenta careia i se adreseaza
    comp = memorie[index + 4]
    if comp == 0x00:
        comp = 'Y'
    elif comp == 0x01:
        comp = 'Cb/Cr'

    # Extragem datele din secțiune
    continut = memorie[index + 5: index + 2 + lungime]

    # Construirea tabelului de cuantizare
    zigzag_indici = [(0, 0), (0, 1), (1, 0), (2, 0), (1, 1), (0, 2), (0, 3), (1, 2),
                     (2, 1), (3, 0), (4, 0), (3, 1), (2, 2), (1, 3), (0, 4), (0, 5),
                     (1, 4), (2, 3), (3, 2), (4, 1), (5, 0), (6, 0), (5, 1), (4, 2),
                     (3, 3), (2, 4), (1, 5), (0, 6), (0, 7), (1, 6), (2, 5), (3, 4),
                     (4, 3), (5, 2), (6, 1), (7, 0), (7, 1), (6, 2), (5, 3), (4, 4),
                     (3, 5), (2, 6), (1, 7), (2, 7), (3, 6), (4, 5), (5, 4), (6, 3),
                     (7, 2), (7, 3), (6, 4), (5, 5), (4, 6), (3, 7), (4, 7), (5, 6),
                     (6, 5), (7, 4), (7, 5), (6, 6), (5, 7), (6, 7), (7, 6), (7, 7)]

    tabel = [[0]*8 for _ in range(8)]

    for i in range(len(continut)):
        tabel[zigzag_indici[i][0]][zigzag_indici[i][1]] = continut[i]

    # Salvăm informațiile despre etichetă
    markeri_gasiti.append({
        'marker': 0xFFDB,
        'nume': 'DQT',
        'pozitie': index,
        'lungime': lungime,
        'componenta': comp,
        'tabel': np.array(tabel),
        'continut': continut,
        'raw': raw
    })
```

```
def SOF0(index):
    # lungimea = 8 + (componente * 3)
    lungime = struct.unpack('>H', memorie[index + 2: index + 4])[0]
    precizie = memorie[index + 4]
    inaltime = struct.unpack('>H', memorie[index + 5: index + 7])[0]
    latime = struct.unpack('>H', memorie[index + 7: index + 9])[0]

    # RAW
    raw = memorie[index: index + 2 + lungime]

    #1 = grey scale, 3 = YCbCr, 4 = CMYK
    nr_componente = memorie[index + 9]

    j = 10
    componente = []
    for i in range(int((lungime - 8) / 3)):
        #1 = Y, 2 = Cb, 3 = Cr, 4 = I, 5 = Q
        id_componenta = memorie[index + j]
        j += 1
        factor_subesantionare = memorie[index + j]
        j += 1
        table_cuantizare = memorie[index + j]
        j += 1
        componente.append({
            'id_componenta': id_componenta,
            'factor_subesantionare': factor_subesantionare,
            'table_cuantizare': table_cuantizare
        })
    markeri_gasiti.append({
        'marker': 0xFFC0,
        'nume': 'SOF0',
        'pozitie': index,
        'lungime': lungime,
        'precizie': precizie,
        'inaltime': inaltime,
        'latime': latime,
        'nr_componente': nr_componente,
        'componente': componente,
        'raw': raw
    })

def generare_codare(tabel):
    coduri = []
    var = 0
    for i in range(1, tabel['ultimul non-zero'] + 2):
        if tabel['lungimi'][i-1] != 0:
            for _ in range(tabel['lungimi'][i-1]):
```

```
            cod = "{0:b}".format(var)
            cod = (i - len(cod))*"0" + cod
            coduri.append(cod)
            var += 1
        var <<= 1

    return coduri

def DHT(index):
    # Aflăm lungimea secțiunii (2 octeți după etichetă)
    lungime = struct.unpack('>H', memorie[index + 2: index + 4])[0]

    # RAW
    raw = memorie[index: index + 2 + lungime]

    # Aflam numarul de identificare (1 octet)
    nr_identificare = memorie[index + 4]
    plan = nr_identificare & 0xF
    DCAC = nr_identificare >> 4

    if plan == 0:
        plan = 'Y'
    else:
        plan = 'Cb/Cr'

    if DCAC == 0:
        DCAC = 'DC'
    else:
        DCAC = 'AC'

    j = 5
    lungimi = []
    ultimul = None
    for i in range(16):
        l = memorie[index + j]
        j += 1
        lungimi.append(l)
        if l != 0:
            ultimul = i

    simboluri = []
    for i in range(16):
        if lungimi[i] > 0:
            for k in range(lungimi[i]):
                simboluri.append(memorie[index + j])
                j += 1
```

```
    tabel = {
        'lungimi': lungimi,
        'simboluri': simboluri,
        'ultimul non-zero': ultimul
    }

    codare = dict(zip(generare_codare(tabel), tabel['simboluri']))

    markeri_gasiti.append({
        'marker': 0xFFC4,
        'nume': 'DHT',
        'pozitie': index,
        'lungime': lungime,
        'componenta': plan,
        'DCAC': DCAC,
        'tabelHuffman': tabel,
        'codare': codare,
        'raw': raw
    })

def SOS(index):
    # Aflăm lungimea secțiunii (2 octeți după etichetă)
    lungime = struct.unpack('>H', memorie[index + 2: index + 4])[0]
    nr_componente = memorie[index + 4]

    # RAW
    raw = memorie[index: index + 2 + lungime]

    j = 5
    componente = []
    for i in range(nr_componente):
        id_componenta = memorie[index + j]
        j += 1
        tabele = memorie[index + j]
        j += 1

        componente.append({
            'id': id_componenta,
            'AC': tabele & 0xF,
            'DC': tabele >> 4
        })

    markeri_gasiti.append({
        'marker': 0xFFDA,
        'nume': 'SOS',
        'pozitie': index,
        'lungime': lungime,
```

```
        'nr_componente': nr_componente,
        'componente': componente,
        'raw': raw
    })

def EOI(index):
    # RAW
    raw = memorie[index: index + 2]

    markeri_gasiti.append({
        'marker': 0xFFD9,
        'nume': 'EOI',
        'pozitie': index,
        'raw': raw
    })

# Cauta markeri
def cautare_markeri():
    index = 0
    while index < len(memorie):
        # Verificare dacă avem un marker JPEG (0xFF)
        if memorie[index] == 0xFF:
            tag = memorie[index + 1]  # Codul etichetei
            if tag == 0xD8:
                SOI(index)
            elif tag == 0xE0 or tag == 0xE1 or tag == 0xE2:
                APP(index, tag)
            elif tag == 0xDB:
                DQT(index)
            elif tag == 0xC0:
                SOF0(index)
            elif tag == 0xC4:
                DHT(index)
            elif tag == 0xDA:
                SOS(index)
            elif tag == 0xD9:
                EOI(index)
            else:
                pass
        index += 1

def gasesteImagini():
    def asamblareImagine(pozitieSOI):
        imagine = {
            'SOI': None,
            'APP': None,
            'DQT': [],
```

```
        'SOF0': None,
        'DHT': [],
        'SOS': None,
        'EOI': None,
        'data': [],
        'rezultat': None
    }

    frecventa_markeri = {
        'APP': 1,
        'DQT': 2,
        'SOF0': 1,
        'DHT': 4,
        'SOS': 1,
        'EOI': 1
    }

    for marker in markeri_gasiti:
        if abs(marker['pozitie'] - pozitieSOI) < 1024:
            if marker['nume'] in frecventa_markeri.keys():
                if frecventa_markeri[marker['nume']] > 0:
                    if marker['nume'] == 'APP':
                        imagine['APP'] = marker
                        frecventa_markeri['APP'] -= 1
                    elif marker['nume'] == 'DQT':
                        imagine['DQT'].append(marker)
                        frecventa_markeri['DQT'] -= 1
                    elif marker['nume'] == 'SOF0':
                        imagine['SOF0'] = marker
                        frecventa_markeri['SOF0'] -= 1
                    elif marker['nume'] == 'DHT':
                        imagine['DHT'].append(marker)
                        frecventa_markeri['DHT'] -= 1
                    elif marker['nume'] == 'SOS':
                        imagine['SOS'] = marker
                        frecventa_markeri['SOS'] -= 1

    return imagine

def validareImagine(imagine):
    if imagine['APP'] == None:
        return 0
    if len(imagine['DQT']) == 0:
        return 0
    if imagine['SOF0'] == None:
        return 0
    if len(imagine['DHT']) in [0, 3]:
```

```
        return 0
    if imagine['SOS'] == None:
        return 0

    if imagine['SOF0']['nr_componente'] == 1:
        if len(imagine['DQT']) == 2 or len(imagine['DHT']) == 4:
            return 0
        if imagine['DQT']['componenta'] == 'Cb/Cr':
            return 0
        if imagine['DHT'][0]['componenta'] == 'Cb/Cr' or imagine['DHT'][0]['componenta'] == 'Cb/Cr':
            return 0
        if imagine['SOS']['nr_componente'] != 1:
            return 0

        dc_dft = ac_dft = 0
        if imagine['DHT'][0]['DCAC'] == 'DC' or imagine['DHT'][1]['DCAC'] == 'DC':
            dc_dft += 1
        if imagine['DHT'][0]['DCAC'] == 'AC' or imagine['DHT'][1]['DCAC'] == 'AC':
            ac_dft += 1
        if dc_dft != 1 or ac_dft != 1:
            return 0

    if imagine['SOF0']['nr_componente'] == 3:
        if len(imagine['DQT']) == 1 or len(imagine['DHT']) == 2:
            return 0
        if imagine['SOS']['nr_componente'] != 3:
            return 0

        y_dqt = c_dqt = 0
        if imagine['DQT'][0]['componenta'] == 'Y' or imagine['DQT'][1]['componenta'] == 'Y':
            y_dqt += 1
        if imagine['DQT'][0]['componenta'] == 'Cb/Cr' or imagine['DQT'][1]['componenta'] == 'Cb/Cr':
            c_dqt += 1
        if y_dqt != 1 or c_dqt != 1:
            return 0

        ydc_dft = yac_dft = cdc_dft = cac_dft = 0
        if (imagine['DHT'][0]['DCAC'] == 'DC' and imagine['DHT'][0]['componenta'] == 'Y') or
(imagine['DHT'][1]['DCAC'] == 'DC' and imagine['DHT'][1]['componenta'] == 'Y') or
(imagine['DHT'][2]['DCAC'] == 'DC' and imagine['DHT'][2]['componenta'] == 'Y') or
(imagine['DHT'][3]['DCAC'] == 'DC' and imagine['DHT'][3]['componenta'] == 'Y'):
            ydc_dft += 1
        if (imagine['DHT'][0]['DCAC'] == 'AC' and imagine['DHT'][0]['componenta'] == 'Y') or
(imagine['DHT'][1]['DCAC'] == 'AC' and imagine['DHT'][1]['componenta'] == 'Y') or
(imagine['DHT'][2]['DCAC'] == 'AC' and imagine['DHT'][2]['componenta'] == 'Y') or
(imagine['DHT'][3]['DCAC'] == 'AC' and imagine['DHT'][3]['componenta'] == 'Y'):
            yac_dft += 1
```

```
            if (imagine['DHT'][0]['DCAC'] == 'DC' and imagine['DHT'][0]['componenta'] == 'Cb/Cr') or
(imagine['DHT'][1]['DCAC'] == 'DC' and imagine['DHT'][1]['componenta'] == 'Cb/Cr') or
(imagine['DHT'][2]['DCAC'] == 'DC' and imagine['DHT'][2]['componenta'] == 'Cb/Cr') or
(imagine['DHT'][3]['DCAC'] == 'DC' and imagine['DHT'][3]['componenta'] == 'Cb/Cr'):
                cdc_dft += 1
            if (imagine['DHT'][0]['DCAC'] == 'AC' and imagine['DHT'][0]['componenta'] == 'Cb/Cr') or
(imagine['DHT'][1]['DCAC'] == 'AC' and imagine['DHT'][1]['componenta'] == 'Cb/Cr') or
(imagine['DHT'][2]['DCAC'] == 'AC' and imagine['DHT'][2]['componenta'] == 'Cb/Cr') or
(imagine['DHT'][3]['DCAC'] == 'AC' and imagine['DHT'][3]['componenta'] == 'Cb/Cr'):
                cac_dft += 1
            if ydc_dft != 1 or yac_dft != 1 or cdc_dft != 1 or cac_dft != 1:
                return 0

        return 1

    # Lista imaginilor gasite
    imagini = []

    for marker in markeri_gasiti:
        if marker['nume'] == 'SOI':
            imagine = asamblareImagine(marker['pozitie'])
            imagine['SOI'] = marker

            if validareImagine(imagine):
                imagini.append(imagine)

            # Adaugam EOI
            index = imagine['SOS']['pozitie'] + imagine['SOS']['lungime']
            while True:
                try:
                    if memorie[index] == 0xFF and memorie[index+1] == 0xD9:
                        for m in markeri_gasiti:
                            if m['pozitie'] == index and m['nume'] == 'EOI':
                                imagine['EOI'] = m
                                break
                    index += 1
                except:
                    break

            # Adaugam datele raw
            for index in range(imagine['SOS']['pozitie'] + imagine['SOS']['lungime'] + 2,
imagine['EOI']['pozitie']):
                imagine['data'].append(memorie[index])
            print(len(imagine['data']))

            imagine['data'] = bytes(imagine['data'])
```

```
# Reordonam listele ca sa fie usor de procesat imaginea
aux = []
if imagine['SOF0']['nr_componente'] == 1:
    if imagine['DHT'][0]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][0])
        aux.append(imagine['DHT'][1])
    else:
        aux.append(imagine['DHT'][1])
        aux.append(imagine['DHT'][0])
    imagine['DHT'] = aux
else:
    if imagine['DQT'][0]['componenta'] == 'Y':
        aux.append(imagine['DQT'][0])
        aux.append(imagine['DQT'][1])
    else:
        aux.append(imagine['DQT'][1])
        aux.append(imagine['DQT'][0])
    imagine['DQT'] = aux

    aux = []
    if imagine['DHT'][0]['componenta'] == 'Y' and imagine['DHT'][0]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][0])
    elif imagine['DHT'][1]['componenta'] == 'Y' and imagine['DHT'][1]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][1])
    elif imagine['DHT'][2]['componenta'] == 'Y' and imagine['DHT'][2]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][2])
    elif imagine['DHT'][3]['componenta'] == 'Y' and imagine['DHT'][3]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][3])

    if imagine['DHT'][0]['componenta'] == 'Y' and imagine['DHT'][0]['DCAC'] == 'AC':
        aux.append(imagine['DHT'][0])
    elif imagine['DHT'][1]['componenta'] == 'Y' and imagine['DHT'][1]['DCAC'] == 'AC':
        aux.append(imagine['DHT'][1])
    elif imagine['DHT'][2]['componenta'] == 'Y' and imagine['DHT'][2]['DCAC'] == 'AC':
        aux.append(imagine['DHT'][2])
    elif imagine['DHT'][3]['componenta'] == 'Y' and imagine['DHT'][3]['DCAC'] == 'AC':
        aux.append(imagine['DHT'][3])

    if imagine['DHT'][0]['componenta'] == 'Cb/Cr' and imagine['DHT'][0]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][0])
    elif imagine['DHT'][1]['componenta'] == 'Cb/Cr' and imagine['DHT'][1]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][1])
    elif imagine['DHT'][2]['componenta'] == 'Cb/Cr' and imagine['DHT'][2]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][2])
    elif imagine['DHT'][3]['componenta'] == 'Cb/Cr' and imagine['DHT'][3]['DCAC'] == 'DC':
        aux.append(imagine['DHT'][3])
```

```
            if imagine['DHT'][0]['componenta'] == 'Cb/Cr' and imagine['DHT'][0]['DCAC'] == 'AC':
                aux.append(imagine['DHT'][0])
            elif imagine['DHT'][1]['componenta'] == 'Cb/Cr' and imagine['DHT'][1]['DCAC'] == 'AC':
                aux.append(imagine['DHT'][1])
            elif imagine['DHT'][2]['componenta'] == 'Cb/Cr' and imagine['DHT'][2]['DCAC'] == 'AC':
                aux.append(imagine['DHT'][2])
            elif imagine['DHT'][3]['componenta'] == 'Cb/Cr' and imagine['DHT'][3]['DCAC'] == 'AC':
                aux.append(imagine['DHT'][3])

            imagine['DHT'] = aux

    return imagini



cautare_markeri()
imagini = gasesteImagini()

index = 1
for imagine in imagini:
    with open(f'{index}.jpeg', "wb") as f:
        f.write(imagine['SOI']['raw'])  # SOI - Start of Image

        # APP0 marker
        f.write(imagine['APP']['raw'])  # Marker APP0

        # Adaugă mai multe tabele DQT
        for dqt in imagine['DQT']:  # Lista cu toate markerii DQT
            f.write(dqt['raw'])

        # SOF0 marker
        f.write(imagine['SOF0']['raw'])  # Marker SOF0

        # Adaugă mai multe tabele DHT
        for dht in imagine['DHT']:  # Lista cu toate markerii DHT
            f.write(dht['raw'])

        # SOS marker
        f.write(imagine['SOS']['raw'])  # Marker SOS

        # Datele efectiv codificate Huffman
        f.write(imagine['data'])

        # EOI marker
        f.write(imagine['EOI']['raw'])  # EOI - End of Image
    index += 1
```

```
from PIL import Image
import matplotlib.pyplot as plt

for index in range(1, len(imagini)+1):
    img = Image.open(f'{index}.jpeg')

    plt.figure()
    plt.imshow(img)
```