КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра обчислювальної математики

Кваліфікаційна робота

на здобуття ступеня магістра

за спеціальністю 113 Прикладна математика

на тему:

СТИСНЕННЯ ЗОБРАЖЕНЬ ТА ВІДЕО

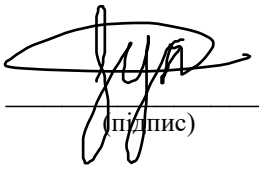
Виконав студент 2-го курсу магістратури
Лук'янець Павло Андрійович

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Матвієнко Володимир Тихонович

_____
(підпис)

_____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент 

_____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри обчислювальної математики
«____» _____ 2023 р.,
протокол № ___
Завідувач кафедри
С. І. Ляшко_____
(підпис)

Київ – 2023

ЗМІСТ

АНОТАЦІЯ

Кваліфікаційна робота: 58 сторінок, 29 рисунків, 15 джерел, 2 додатків. Ключові слова: стиснення даних, обробка зображень, кількість інформації, перетворення, стиснення зображення та відео.

Мета дослідження: створити математичну модель для оцінки ефективності алгоритмів стиснення зображень, оцінити ефективність сучасних алгоритмів стиснення та провести їх дослідження з метою підвищення ступеня стиснення. Розробити програмну модель алгоритму стиснення, яка покращує ефективність за допомогою контексту стиснення під час кодування послідовності зображень та відео. Вивчити можливості використання алгоритмів машинного навчання під час формування контексту стиснення та побудувати повноцінну модель алгоритму стиснення.

Об’єкт дослідження: алгоритми стиснення даних, зображень та відео, методи знаходження кількості інформації в даних.

Предмет дослідження: ефективність алгоритмів стиснення, що мінімізують обсяг переданих і збережених даних з мінімальними втратами інформації.

Методи дослідження: метод аналізу, метод узагальнення, методи спостереження та порівняння.

Завдання дослідження:

1) Вивчити математичну базу алгоритмів стиснення даних і зображень;

2) Побудувати алгоритм стиснення зображення на основі контексту стиснення та потенційно перевершувати за ефективністю існуючі алгоритми стиснення;

3) Зробити програмну реалізацію стиснення зображення та відео за допомогою розробленого алгоритму.

Елементи наукової новизни отриманих результатів: модель алгоритму стиснення зображення, здатного навчатися та адаптуватися до вхідних даних.

Область можливого практичного застосування: передача та зберігання послідовності зображень, підвищення ефективності стиснення відео.

ВСТУП

З поширенням як цифрових медіа, так й інтернету в сучасному світі все частіше постає питання зберігання та передачі даних у цифровому форматі. Запит на програмну оптимізацію процесів зберігання та передачі даних стає все частішим, оскільки потреби користувачів зростають швидше, ніж вдосконалюються апаратні технології для цих цілей. Крім механізмів балансування навантаження, оптимізації операцій вводу-виводу, оптимізації мережевої передачі на транспортному рівні важливу роль відіграють механізми стиснення даних. Стиснення даних передбачає зменшення розміру інформації, що розглядається, використовуючи той факт, що дані здебільшого не є випадковим набором бітів, а підкоряються певному закону. Іншими словами, використовується той факт, що дані є або залежними випадковими змінними, або випадковими змінними, що підлягають певній нерівномірній функції розподілу.

Більшість даних, що передаються через мережу - це фото та відеофайли. Для швидкої передачі цих файлів і їх компактного зберігання потрібні ефективні алгоритми стиснення, які можуть швидко й ефективно стискати окремі зображення та послідовності відеокадрів. Проблемою при стисненні послідовності відеокадрів є обчислювальна складність знаходження областей кореляції зображення, що робиться для збільшення коефіцієнта стиснення шляхом кодування посилання на подібну область зображення та різниці між попереднім і наступним зображеннями, замість цього кодування всього вихідного зображення. Проблема зі стисненням набору зображень полягає у виділенні загального контексту для даних зображення, щоб покращити якість стиснення.

У цій роботі розглянуто теоретичну базу, яка лежить в основі алгоритмів стиснення зображень останніх десятиліть, систематизовано та обґрунтовано основні підходи до стиснення зображень, з’ясовано, що таке межа стиснення та як теоретично вона може бути досягнута, програмно реалізовано алгоритми, що покращують ефективність стиснення послідовності відеокадрів, послідовності зображень, розглянуто використання алгоритмів машинного навчання в області

стиснення зображень і реалізовано модель, яка покращує ефективність стиснення послідовності зображень шляхом навчання попередніх даних і збереження контексту.

Глава 1. ОСНОВНІ ПРИНЦИПИ СТИСНЕННЯ ДАНИХ

1.1 Основні поняття

Розглянемо основні підходи, які використовуються для стиснення даних (включаючи зображення та відео). Для подальшої роботи нам знадобляться такі терміни:

- інформаційна ентропія – непередбачуваність появи будь-якого символу алфавіту;
- формула кількості інформації за Шенноном;
- стиснення без втрат - стиснення, при якому вихідні дані можуть бути точно відновлені після стиснення;
- стиснення з втратами — стиснення, при якому відновлені після стиснення дані можуть дещо відрізнятися від вихідних даних.

1.2 Використовувана література, технології

Перш ніж реалізувати наш власний алгоритм стиснення, ми запровадимо найпопулярніший алгоритм стиснення, щоб заглибитись у деталі реалізації алгоритмів стиснення зображень. Ми реалізуємо JPEG, оскільки він є найпопулярнішим серед алгоритмів стиснення з втратами даних і одним із найскладніших алгоритмів. Детальний опис алгоритму було взято зі статті Стенфордського університету, яка описує стандарт JPEG[7][8][9]. Для реалізації алгоритму будемо використовувати мову Python.

1.3 Сучасні алгоритми стиснення даних

1.3.1 Види стиснення

Сучасні алгоритми стиснення поділяються на два види: без втрат і з втратами. Без втрат використовується для стиснення даних будь-якого формату, тоді як з

втратами даних використовується в основному для стиснення аудіо та медіа файлів, оскільки ці формати даних можна дещо змінити без значного пошкодження даних. Алгоритми з втратами стискають дані набагато ефективніше завдяки тому, що вони значно зменшують ентропію даних, незначно змінюючи ці дані. Розглянемо прийоми, які використовуються для реалізації алгоритмів стиснення без втрат і з втратами [1].

1.3.2 Стиснення з втратами

Стиснення з втратами застосовується для тих форматів даних, для яких незначна втрата інформації не є критичною. Такі формати – це, наприклад, зображення, аудіозаписи, відео. Стиснення із втратами досягається за рахунок видалення інформації, до якої несприйнятливі людські органи почуттів. Варто зауважити, що з точки зору комп'ютера та стандартних математичних метрик типу зображення (відео, аудіо) після декомпресії може значно відрізнятися від вихідного. Внаслідок цього, значна увага в розробці алгоритмів стиснення з втратами приділяється пошуку функцій для вимірювання рівня спотворень після компресії, а також функцій порівняння двох зображень. Одна з найпопулярніших метрик, що використовується для оцінки якості стиснення зображень, - це PSNR :

$$\mathrm{PSNR} = 10\log_{10}\left(\frac{\mathrm{MAX}_\mathrm{i}^2}{\mathrm{RMSE}^2}\right) = 20\log_{10}\left(\frac{\mathrm{MAX}_\mathrm{i}}{\mathrm{RMSE}}\right) \qquad (1.1)$$

де MAX_i - це максимально можливе значення, яке піксель зображення може приймати (255 – якщо це ч/б зображення), а RMSE :

$$\mathrm{RMSE} = \sqrt{\frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left|\mathrm{I}(i,j) - \mathrm{K}(i,j)\right|^2} \qquad (1.2)$$

де I, K - два зображення, одне з яких є зашумленою копією іншого зображення.

Стиснення з втратами досягається в основному шляхом квантування даних. Однак квантування вихідних даних помітно пошкодить їх і створить видимі межі між рівнями квантування. Тому використовуються додаткові методи, які

підвищують ступінь стиснення та якість вихідних даних. Одним із таких прийомів є використання перетворень, які концентрують загальну інформацію в невеликому наборі байтів, залишаючи лише уточнення цієї інформації в решті байтів. Такими алгоритмами є DCT (рис. 1.1) і DWT (рис. 1.2)[5][6]

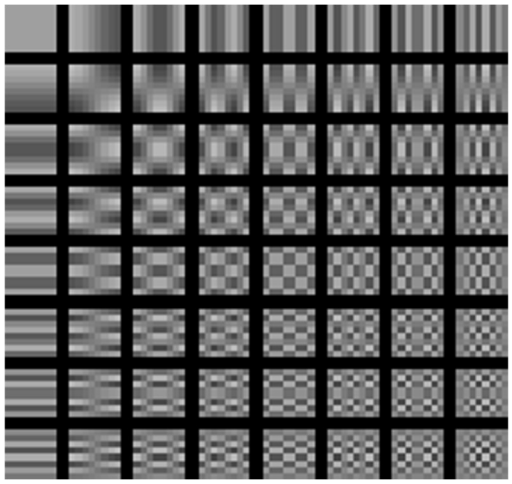


Рисунок 1.1 - Ілюстрація коефіцієнтів DCT-2

Рисунок 1.2 - Ілюстрація одного проходу ДВП-2Д

1.3.3 Стиснення без втрат

Стиснення без втрат у загальному розумінні цього слова може бути застосоване до будь-яких даних, представлених у цифровій формі. Стиснення без

втрат може бути реалізовано за допомогою усунення таких видів надмірності, як [1]:

– кодова надмірність (ентропійна надмірність). Код - послідовність символів, яка використовується для представлення інформації. Послідовність кодових символів є кодовим словом. 8, 24 і 32-бітові коди, які використовуються для представлення значень у більшості двовимірних масивів яскравості, як правило, містять більше бітів, ніж необхідно. Простіше кажучи, якщо зображення містить тільки синій і зелений кольори, немає необхідності виділяти окремий байт для червоного. Символи, які найчастіше зустрічаються, кодуються меншою кількістю бітів, найрідкісніші — більшою.

– просторова або часова надмірність. Оскільки значення пікселів у більшості двовимірних масивів яскравості просторово корельовані (тобто значення кожного пікселя подібне до значень сусідніх пікселів), інформація без потреби дублюється в корельованому піксельному представленні. У відеопослідовності інформація також дублюється в пікселях, корельованих у часі (тобто тих, чиї значення подібні або залежать від значень пікселів у сусідніх кадрах).

Основними компонентами алгоритмів стиснення без втрат є алгоритми послідовного кодування та алгоритми ентропійного стиснення даних. Прикладом алгоритмів послідовного кодування є RLE, будь-який словниковий алгоритм. Основним принципом є стиснення повторюваних підмножин розглянутого набору символів. Реалізується за допомогою посилань на попередній аналогічний фрагмент, або за допомогою вказівки кількості повторень символів. Прикладами ентропійних кодів є коди Хаффмана, коди Голомба-Райса[2], арифметичне кодування. Основний принцип полягає в тому, щоб зберігати символи, які найчастіше зустрічаються в коротких кодах, і, відповідно, символи, що рідко зустрічаються, в довгих кодах.

Крім того, під час стиснення зображень використовується техніка передбачення наступного значення на основі вже декодованих (зазвичай наступний піксель прогнозується на основі лівого та верхнього пікселів, оскільки лише декодер знатиме їх під час декодування поточного пікселя). Такі прогнози

використовують той факт, що зображення є двовимірним об’єктом і сусідні пікселі здебільшого не сильно відрізняються один від одного.[3][4].

1.4 Особливості стиснення зображень та відео

Нарешті, стиснення зображення часто використовує переваги людського зору, а саме той факт, що людське око більш чутливе до кількості світла в зображенні, ніж до його кольору. Також при кодуванні зображень особлива увага приділяється переходам між кольорами сусідніх пікселів.

Майже всі сьогодні найпоширеніші алгоритми стиснення зображень засновані на методах, описаних вище, а саме, вони є лише комбінацією тієї чи іншої підмножини багатьох методів, описаних вище (рис. 1.3).

Рисунок 1.3 - Демонстрація складових найпопулярніших на сьогоднішній день алгоритмів

Говорячи про алгоритми стиснення відео, варто сказати, що найчастіше вони засновані на алгоритмах стиснення зображення. Єдина відмінність полягає в тому, що відеоалгоритми використовують інформацію про попередньо стиснуті кадри і спираються на те, що зображення у відеопотоці відрізняються здебільшого лише зміщенням деяких пікселів на кілька позицій. Це призводить до появи двох додаткових методів: компенсації руху та різницевого кодування. Перший

використовується для знаходження руху областей і компенсації цього руху, другий використовується для кодування не самого кадру, а різниці між поточним кадром і попереднім кадром після компенсації руху (рис. 1.4).

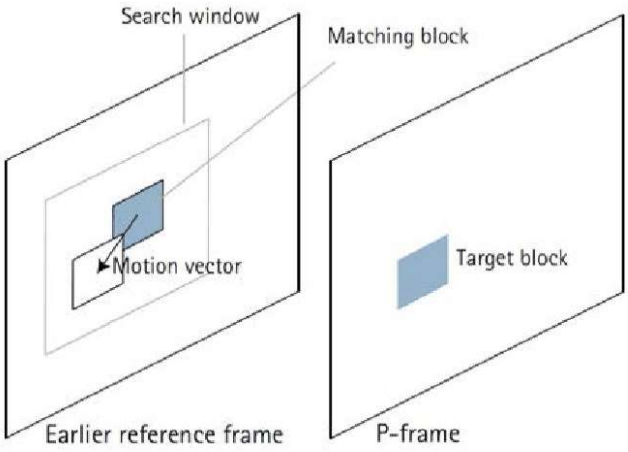


Рисунок 1.4 - Як працює компенсація руху

Крім алгоритмів стиснення зображень і алгоритмів стиснення відео, існують також алгоритми, які спеціалізуються на стисненні послідовності схожих зображень. Зазвичай вони будуються так само, як і алгоритми стиснення відео, але без етапу компенсації руху, оскільки об'єкти в послідовності зображень зазвичай сильно зміщені відносно один одного і неможливо застосувати існуючі алгоритми компенсації руху. Ця робота спрямована на написання більш оптимального алгоритму для стиснення послідовності зображень, для яких не існує можливості ефективно застосувати сучасні види компенсації руху.

Глава 2. РЕАЛІЗАЦІЯ ОСНОВНИХ АЛГОРИТМІВ СТИСНЕННЯ ЗОБРАЖЕНЬ. ПЕРЕДУМОВИ ДО ПОКРАЩЕННЯ ІСНУЮЧИХ АЛГОРИТМІВ

2.1 Дослідження різноманітних підходів задля вирішення задачі стиснення

2.1.1 Мета дослідження

Вивчивши принципи алгоритмів стиснення та знаючи, що рівень стиснення можна покращити, використовуючи контекст попередніх зображень, ми реалізували алгоритм, який дозволяє отримувати, зберігати та шукати контекст зображення на високому рівні, тобто знайдений на різних зображеннях Алгоритми для подібних регіонів. Цей алгоритм не підходить для послідовностей зображень, але він дуже підходить для стиснення ключових кадрів відео (рис. 2.1).

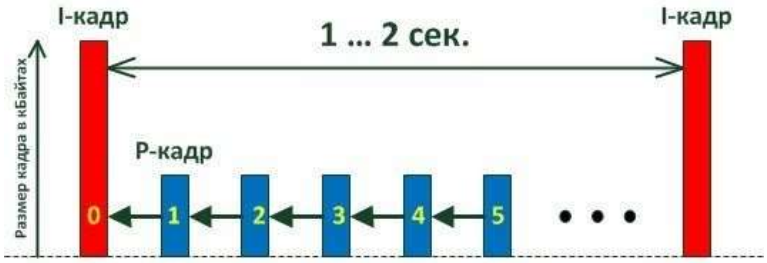


Рисунок 2.1 - Ключові (I) і дельта (P) кадри

Як правило, ключові кадри відео стискаються за допомогою звичайних алгоритмів стиснення зображення без урахування контексту попередньо стиснутих ключових кадрів. Це відбувається тому, що об’єкти часто змінюють своє положення на різних ключових кадрах, а стандартні алгоритми компенсації руху не можуть врахувати це відхилення. Тому в даному випадку необхідно розробити алгоритм, який зможе знаходити неточні збіги сегментів зображення незалежно від їх положення.

2.1.2 Алгоритми компенсації руху

Проблема пошуку схожих об'єктів на зображеннях виникає в тій чи іншій формі в багатьох областях програмування: від стиснення відео до розпізнавання об'єктів. Розглянемо найпопулярніші підходи до вирішення даної задачі і виберемо найбільш підходящий для проблеми, що розглядається. Більше уваги приділимо таким підходам: оцінка руху, оптичний потік, розпізнавання об’єктів.

Для стиснення відео використовуються алгоритми, засновані на тому, що об'єкти в сусідніх кадрах рухаються близько один до одного. Ці алгоритми розглядають прилеглу область навколо точки, що розглядається, і вибирають найбільш підходящу область, для якої різниця між розглянутим кадром і попереднім згідно з обраною метрикою буде найкращою. Ми не можемо гарантувати, що подібні об'єкти будуть поруч, тому такий підхід не підходить.

2.1.3 Алгоритми знаходження оптичного потоку

Для пошуку оптичного потоку та розпізнавання об’єктів використовується аналогічний процес [10]: спочатку ми вибираємо ключові (спеціальні) точки на зображенні, потім будуємо для них дескриптори (опис навколишнього простору) і те ж саме робимо для вони створюють два зображення, тоді ми шукаємо відповідність між дескрипторами. Чим кращий збіг між дескрипторами, тим більш схожими будуть області навколо розглянутих ключових точок. Розглянемо кілька найпопулярніших алгоритмів вибору ключових точок і пошуку дескрипторів.

Одними з найпопулярніших методів пошуку та опису ключових точок є ORB, SIFT, SURF [11] [12] [13]. У SIFT і SURF існує незмінність розміру та обертання дескрипторів у функціях. Однак наша мета полягає не в тому, щоб знайти опис ключових точок, які є інваріантними щодо обертання, і, що більш важливо, ключових точок, які є інваріантними щодо масштабу. Тому SIFT і SURF нам не відразу підходять. Давайте спробуємо ORB на простому прикладі. Давайте зробимо знімок екрана зображення Lenna.png, обріжемо частину та подивимось, як працює дескриптор ключових точок карти ORB. Ми використаємо BFMatch, щоб знайти однаковий дескриптор на двох фрагментах.

```python
import cv2

def orb(file1, file2, count=50):
    img1 = cv2.imread(file1, 0)
    img2 = cv2.imread(file2, 0)

    orb = cv2.ORB_create()
    kp1, des1 = orb.detectAndCompute(img1, None)
    kp2, des2 = orb.detectAndCompute(img2, None)

    if des1 is None or des2 is None:
        return

    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)
    match_img = cv2.drawMatches(img1, kp1, img2, kp2, matches[:count], None)
    cv2.imwrite('Matches.png',match_img)
    return kp1, des1, kp2, des2, matches
```

Рисунок 2.2 - Отримання ключових точок та дескрипторів через ORB

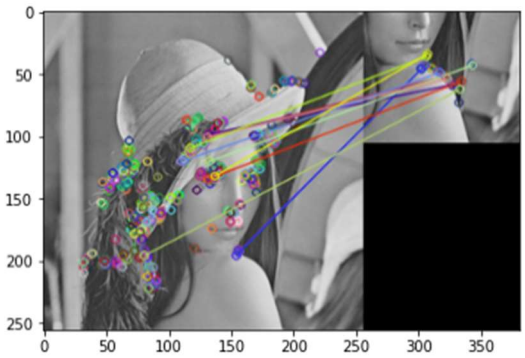


Рисунок 2.3 - Результат зіставлення точок та дескрипторів ORB на тестовому прикладі (пошук на цілому зображенні його частини)

Як бачите, навіть у такому простому прикладі ORB не може гарантувати необхідну точність. Спробуємо знайти відповідно алгоритм пошуку ключових точок і алгоритм їх опису. Алгоритм FAST використовується як частина ORB. Розглянемо їх окремо. Після вибору параметрів можна досягти наступних хороших ефектів (рис. 2.5).

```python
def keypoints(file):
    img = cv2.imread(file,0)

    fast = cv2.FastFeatureDetector_create(threshold=15)
    kp = fast.detect(img,None)

    img2 = cv2.drawKeypoints(img, kp, None,color=(255,0,0))
    cv2.imwrite('fast_true.png',img2)
    plt.imshow(imread("fast_true.png"))
```

Рисунок 2.4 - Отримання ключових точок через FAST

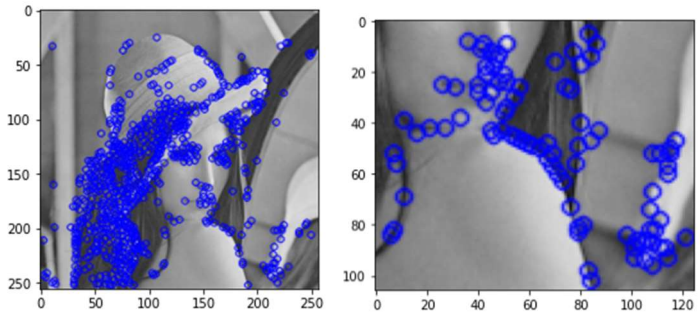


Рисунок 2.5 - Отримання ключових точок через FAST на тестових зображеннях

Зауважимо, що ключові точки обох фрагментів вибираються однаковим способом, що вказує на можливість знайти відповідність між ними при правильному опису.

Алгоритм BRIEF[14] не розглядається, оскільки ORB є комбінацією FAST і BRIEF. Було розглянуто алгоритм FREAK[15] для знаходження дескрипторів. Однак результати його виконання разом з FAST виявилися не кращими в порівнянні з результатами ORB. Було знайдено найкращого кандидата на роль дескриптора, і було вибрано HOG, оскільки він зберігає інформацію про палітру навколо заданої області та враховує положення пікселя, що розглядається, таким чином збігаючи той самий фрагмент краще, ніж інші (рис 2.6, 2.7).

```
def fast_hog(file1, file2, count=50, threshold=10):
    img1 = cv2.imread(file1, 0)
    img2 = cv2.imread(file2, 0)

    fast = cv2.FastFeatureDetector_create(threshold=threshold)
    winSize = (32,32)
    blockSize = (16,16)
    blockStride = (8,8)
    cellSize = (8,8)
    nbins = 9
    derivAperture = 1
    winSigma = 4.
    histogramNormType = 0
    L2HysThreshold = 2.0000000000000001e-01
    gammaCorrection = 0
    nlevels = 64
    hog = cv2.HOGDescriptor(winSize,blockSize,blockStride,cellSize,nbins,derivAperture,winSigma,
                            histogramNormType,L2HysThreshold,gammaCorrection,nlevels)

    winStride = (32,32)
    padding = (32,32)
    kp1 = fast.detect(img1, None)
    kp2 = fast.detect(img2, None)
    des1 = hog.compute(img1,winStride,padding,[k.pt for k in kp1])
    des2 = hog.compute(img2,winStride,padding,[k.pt for k in kp2])

    if des1 is None or des2 is None:
        return

    des1, des2 = np.array(des1).reshape((len(kp1),-1)), np.array(des2).reshape((len(kp2),-1))

    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)
    match_img = cv2.drawMatches(img1, kp1, img2, kp2, matches[:count], None)
    cv2.imwrite('Matches.png',match_img)
    return kp1, des1, kp2, des2, matches[:count]
```

Рисунок 2.6 - Методом FAST ми отримуємо ключові точки, а далі використовуємо HOG для отримання дескрипторів

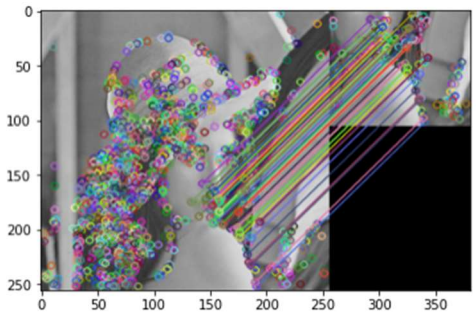


Рисунок 2.7 - Методом FAST ми отримуємо ключові точки, а далі використовуємо HOG на тестових зображеннях

Кількість коректних відповідностей майже досягає максимального рівня. Давайте розглянемо інший приклад для порівняння (рис. 2.8).

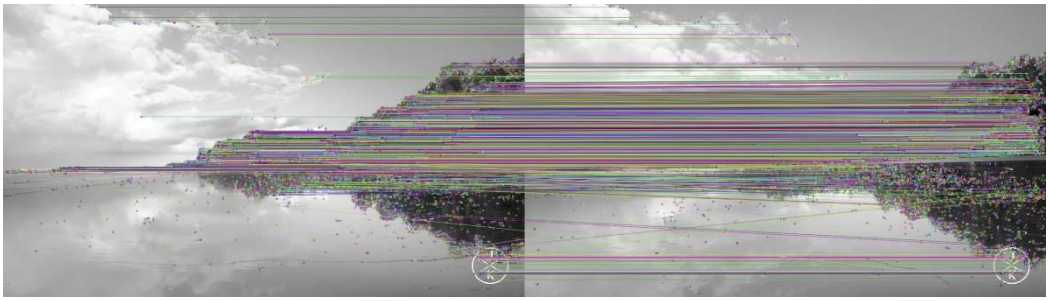


Рисунок 2.8 - Результати застосування комбінації FAST+HOG на тесті, де використовувалися два ключових кадри відеофрагменту.

Більшість відповідностей є правильними, і може здатися, що це два однакові кадри. Тому давайте наведемо нижче модуль попіксельної різниці між цими двома кадрами:

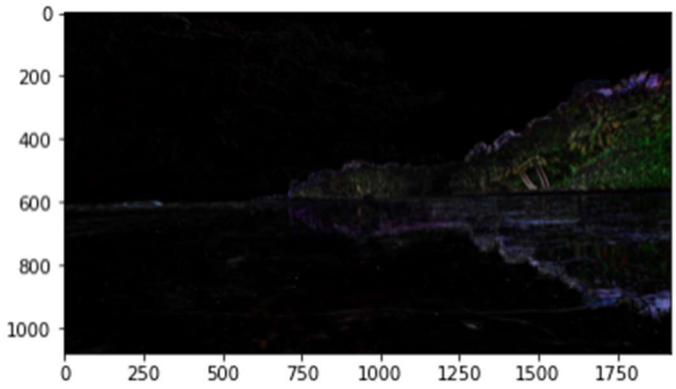


Рисунок 2.9 - Модуль різниці ключових кадрів тестового уривка з відео

Під час експерименту з пошуком найкращого алгоритму для виявлення ключових точок, ми також випробували алгоритм GoodFeaturesToTrack, який був запозичений з методів пошуку оптичного потоку. Нижче наведені результати його роботи:

```
def good_kp(frame):
    gray = frame
    if len(frame.shape) != 2:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    corners = cv2.goodFeaturesToTrack(gray, mask=None, maxCorners=10000, qualityLevel=0.00001, minDistance=1,
    corners = np.int0(corners)
    corners = [corner.ravel() for corner in corners]
    for corner in corners:
        cv2.circle(frame, (corner[0], corner[1]), 3, [0,255,0], -1)
    cv2.imwrite('Matches.png', frame)
    return corners
```

Рисунок 2.10 - Використання алгоритму пошуку ключових точок GoodFeaturesToTrack

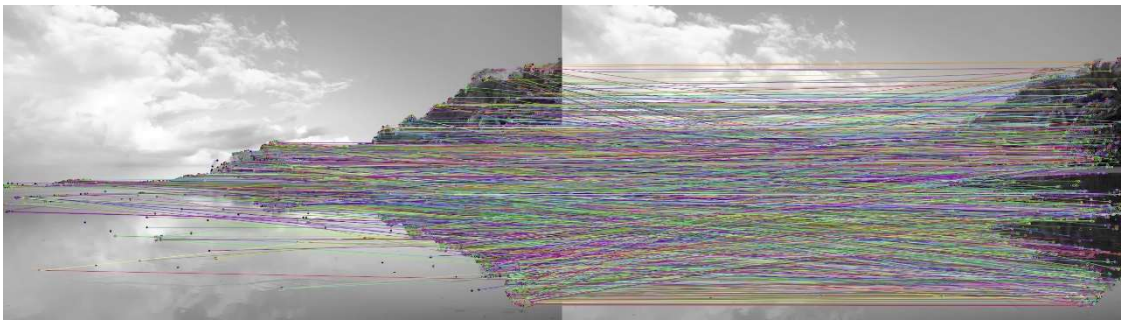


Рисунок 2.11 - Використання алгоритму пошуку ключових точок GoodFeaturesToTrack на тестових зображеннях

Кількість правильних зіставлень в значній мірі зменшується, якщо використовувати алгоритм GoodFeaturesToTrack порівняно з FAST. Таким чином, FAST залишається найкращим варіантом для пошуку ключових точок. У підсумку, найоптимальніша комбінація алгоритмів включає FAST для виявлення ключових точок, HOG для отримання дескрипторів та BFMatch для зіставлення.

2.1.4 Алгоритм JPEG

Складові алгоритму:

- Переклад з RGB до YCrCb.

```
def rgb2ycbcr(img):
    a = np.array([[0.299, 0.587, 0.114],
                  [-0.1687, -0.3313, 0.5],
                  [0.5, -0.4187, -0.0813]])
    b = np.array([0, 128, 128])
    return np.array([[(b + a.dot(x)).astype(int) for x in y] for y in img])
```

Рисунок 2.12 - Функція для перекладу RGB YCrCb

```
def ycbcr2rgb(img):
    a = np.array([[1, 0, 1.402],
                  [1, -0.34414, -0.71414],
                  [1, 1.77, 0]])
    b = np.array([0, 128, 128])

    return np.array([[a.dot(x - b).astype(int) for x in y] for y in img])
```

Рисунок 2.13 - Функція для перекладу YCrCb у RGB

Переконаємося, що видалення даних з Y каналу помітніше для людського ока, ніж з каналів Cr та Cb (рис 2.12).

```
def ycbcr2rgb(img):
    a = np.array([[1, 0, 1.402],
                  [1, -0.34414, -0.71414],
                  [1, 1.77, 0]])
    b = np.array([0, 128, 128])

    return np.array([[a.dot(x - b).astype(int) for x in y] for y in img])
```

Рисунок 2.14 - Функція для замутнення Cr та Cb коефіцієнтів

```
gauss = np.zeros_like(ycbcr_img)
gauss[:,:,0] = gaussian_filter(ycbcr_img[:,:,0], 2)
gauss[:,:,1] = ycbcr_img[:,:,1]
gauss[:,:,2] = ycbcr_img[:,:,2]
```

Рисунок 2.15 - Функція для замутнення коефіцієнтів Y

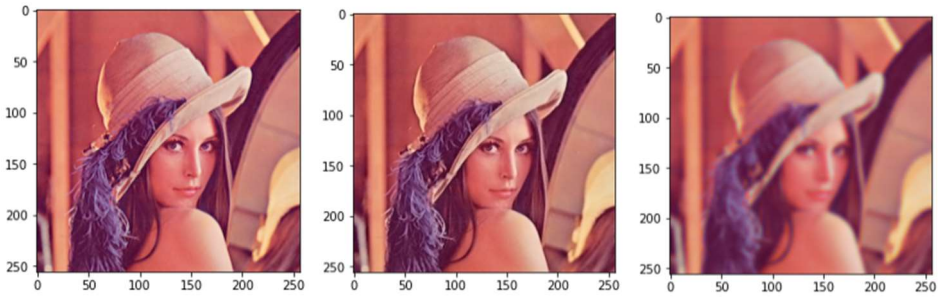


Рисунок 2.16 - Демонстрація впливу однакової розмитості різних каналів на людське око. Перше зображення - вихідне, друге - з розмитими Cb і Cr каналами, третє - з розмитим Y каналом.

Тайлінг - для локалізації по пам'яті та можливості подальшого розпаралелювання, ми використовуємо техніку тайлінгу, яка полягає у розбитті всього зображення на квадрати розміром 8x8. Кожен квадрат можна обробляти незалежно один від одного.

```
...
ret = [[] for i in range(3)]
for i in range(img.shape[0] // 8):
    for j in range(img.shape[1] // 8):
        block = y_component[np.ix_(range(i*8, (i+1)*8), range(j*8, (j+1)*8))]
```

Рисунок 2.17 - Розбиття на тайли на прикладі Y компоненти - Дискретне косинусне перетворення

```
def dct(block):
    alpha = lambda a: 1 / np.sqrt(2) if a == 0 else 1
    cs = lambda a, b: np.cos((2 * a + 1) * b * np.pi/ 16)
    f = lambda x, y, u, v: block[x][y] * cs(x, u) * cs(y, v)
    guv = lambda u, v: 1 / 4 * alpha(u) * alpha(v) * sum([f(x, y, u, v) for y in range(8) for x in range(8)])
    g = np.array([[guv(u, v) for v in range(8)] for u in range(8)])

    return g
```

Рисунок 2.18 - Реалізація ДКП - Квантування

```
y_quantization_matrix = np.array([
    [16, 11, 10, 16, 24, 40, 51, 61],
    [12, 12, 14, 19, 26, 58, 60, 55],
    [14, 13, 16, 24, 40, 57, 69, 56],
    [14, 17, 22, 29, 51, 87, 80, 62],
    [18, 22, 37, 56, 68, 109, 103, 77],
    [24, 35, 55, 64, 81, 104, 113, 92],
    [49, 64, 78, 87, 103, 121, 120, 101],
    [72, 92, 95, 98, 112, 100, 103, 99]
])
color_quantization_matrix = np.array([
    [17, 18, 24, 47, 99, 99, 99, 99],
    [18, 21, 26, 66, 99, 99, 99, 99],
    [24, 26, 56, 99, 99, 99, 99, 99],
    [47, 66, 99, 99, 99, 99, 99, 99],
    [99, 99, 99, 99, 99, 99, 99, 99],
    [99, 99, 99, 99, 99, 99, 99, 99],
    [99, 99, 99, 99, 99, 99, 99, 99],
    [99, 99, 99, 99, 99, 99, 99, 99]
])
```

Рисунок 2.19 - Матриці квантування для Y та Cr/Cb компонент відповідно

Лінеарізація - це процес перетворення матриці 8x8 у рядок з 64 коефіцієнтів в порядку зиг-загу. Це означає, що ми збираємо елементи матриці у послідовність, використовуючи особливості Дискретного косинусного перетворення (DCT). У термінах формату JPEG, перший коефіцієнт називається DC, а решта - AC.

Після лінеарізації матриці, ми можемо застосувати RLE (Run-Length Encoding) кодування для AC коефіцієнтів. Це означає, що ми кодуємо послідовності нульових значень AC коефіцієнтів, замінюючи їх на пари чисел, що вказують кількість послідовних нулів та наступне ненульове значення.

DC коефіцієнти можуть бути кодовані за допомогою дельта кодування. Це означає, що ми кодуємо різницю між поточним DC коефіцієнтом та попереднім DC коефіцієнтом, замість самого значення.

Нарешті, отримані дані можуть бути кодовані за допомогою кодування Хаффмана. Кодування Хаффмана використовується для ефективного подання даних шляхом призначення коротких бітових кодів частішим значенням та довших кодів рідшим значенням.

Таким чином, послідовність кроків, яку потрібно виконати для стиснення даних після зиг-заг перетворення в термінах JPEG, включає лінеарізацію, RLE кодування для AC коефіцієнтів, дельта кодування для DC коефіцієнтів та кодування Хаффмана для отриманих даних.

```
ret = []
n0 = 0
for x in zigzag_list:
    if x == 0:
        if n0 == 0:
            ret.append(0)
        n0 += 1
        continue
    if n0 != 0:
        ret.append(n0)
        n0 = 0
    ret.append(x)

if n0 != 0:
    ret.append(n0)

return ret
```

Рисунок 2.20 - Стиснення RLE після зиг-заг перетворення

2.1.5 Сингулярний розклад

Одним з підходів до стиснення зображення є сингулярне розкладання матриці, складеної з векторів, відповідних окремим зображенням записаної послідовності. Як еталонні вектори використовується набір лівих сингулярних векторів, яким відповідають ненульові значення сингулярних чисел.[13]

Обчислення основних компонент може бути зведено до обчислення сингулярного розкладання матриці даних або обчислення власних векторів і власних значень матриці коварійної вихідних даних.

Сингулярне розкладання виконує декомпозицію дійсної матриці, і ця декомпозиція єдина. Сингулярне розкладання показує геометричну структуру матриці і дозволяє наочно уявити наявні дані. Сингулярне розкладання матриці було розроблено у 1873 році.

Нехай є матриця змінних X розмірністю $(I \times J)$, де I - число рядків, а J - число незалежних змінних (стовпців), яких, як правило, багато $(J >> 1)$. У методі головних компонентів використовуються нові, формальні змінні $tt_a (a = 1...A)$, що є лінійною комбінацією вихідних змінних $x_j (j = 1...J)$.

$$t_a = p_{a1}x_1 + ... + p_{aJ}x_J \qquad (2.1)$$

За допомогою цих нових змінних матриця X розкладається у добуток двох матриць T і P –

$$X = TP^t + E = \sum_{a=1}^{A} t_a p_a^t + E \qquad (2.2)$$

Матриця T називається матрицею рахунків. Її розмірність – (I×A).

Матриця P називається матрицею навантажень. Її розмірність (A×J).

E – це матриця залишків, розмірністю (I×J).

Нові змінні t_a називаються головними компонентами, тому сам метод називається методом головних компонентів. Число стовпців - t_a в матриці T, і p_a в матриці P - дорівнює A, яке називається числом основних компонентів. Ця величина свідомо менша від числа змінних J і числа зразків I.

Важливим властивістю способу є лінійна незалежність основних компонент. Матриця рахунків T не перебудовується зі збільшенням числа компонент, а до неї просто додається ще один стовпець - відповідний новому напрямку. Теж відбувається і з матрицею навантажень P [12].

Розкладання матриці за методом основних компонентів не єдине можливе, тому скористаємося сингулярним розкладанням для декомпозиції матриці за цим методом. У цьому випадку вихідна матриця X розкладається у добуток трьох матриць:

$$X = USV^t \qquad (2.3)$$

S - позитивно визначена діагональна матриця, елементами якої є сингулярні числа, а матриці U,V-унітарні.

Зв'язок між методом головних компонентів та сингулярним розкладанням визначається такими співвідношеннями:

$$T = US,\ P = V. \qquad (2.4)$$

2.2 Розрахунок ефективності стиснення без втрат

Розглянемо, що таке будь-який алгоритм стиснення без втрат з точки зору теорії множин. Стиснення без втрат передбачає однозначне відображення вихідних даних у стислі дані. Розглянемо як множину вихідних даних множину всіх можливих зображень розміру $N \times M$ з числом кроків яскравості D. Нехай множиною початкових даних є X, множиною стиснутих даних є Y. З огляду на бієкцію $|X| = |Y|$.

Тепер припустимо, що на вхід для стиснення надходить випадкове зображення з вибраними параметрами, причому кожне зображення надходить з однаковою ймовірністю. Основна відмінність наведених нижче міркувань від основних принципів стиснення, описаних у першому розділі, полягає в тому, що ми розглядатимемо все зображення як один випадковий об’єкт (випадкову подію). Зауважте, що тоді за формулою Шеннона, припускаючи, що зображення походить від рівномірного розподілу, кількість інформації, необхідної для опису цього зображення, становить:

$$H(x) = -\sum_{|Y|} \frac{1}{|Y|} \log_2 \frac{1}{|Y|} = \log_2 |Y| = \log_2 |X| = NM \log_2 D$$

Легко помітити, що це значення є не що інше, як розмір зображення у форматі BMP без стиснення. Таким чином, незалежно від алгоритму стиснення, якщо вхідне зображення повністю випадкове, в середньому воно займе не менше місця, ніж у нестиснутому вигляді. З цього випливають такі наслідки.

2.3 Твердження про ефективність алгоритмів стиснення зображень та відео

1. Стиснення даних не існує без припущення про нерівномірний розподіл стиснутих даних.

2. Для будь-якого алгоритму стиснення вірно твердження: для будь-якого набору вхідних даних X сума бітів, витрачена на збереження всіх цих даних у стислому вигляді, не менша за суму бітів, витрачена на збереження цих даних у нестисненому , «необроблена» форма, яка є $|X|\log_2|X|$.

3. У свою чергу, з параграфа вище випливає, що для будь-якого алгоритму стиснення частина даних після «стиснення» зменшується в розмірі, а частина збільшується, і чим більше даних вимагає менше біт для зберігання, тим більше біт витрачається на зберігання решта збільшується. стислі дані.

4. З другого абзацу також випливає, що кожному алгоритму можна призначити для кожного набору даних коефіцієнт надлишковості алгоритму стиснення A на множині X, який дорівнює:

$$\frac{\sum_{x\in X} A(x) - |X|\log_2|X|}{|X|}$$

де $A(x)$ – кількість бітів, необхідних для зберігання вхідних даних x, стиснутих алгоритмом A. Зауважимо, що цей коефіцієнт завжди невід’ємний і чим він більший, тим гірше розглянутий алгоритм поводиться з рівномірно розподіленими даними.

5. Немає поняття «ефективність алгоритму стиснення» або «якість алгоритму стиснення» без зазначення множини X, а також інформації про розподіл ймовірностей елементів цієї множини.

6. Не існує і не може існувати універсального алгоритму стиснення, кращого за всі інші алгоритми на будь-яких наборах вхідних даних і будь-яких розподілах вхідних даних.

Таким чином, по-перше, ефективність стиснення досягається лише завдяки тому факту, що на практиці розподіл, якому підлягають вхідні дані, не є рівномірним, а по-друге, алгоритм стиснення, який адаптується до розподілу, до якого належать дані, які він стискає, теоретично може досягти значно кращої якості стиснення, ніж статичний алгоритм.

2.4 Теоретичне досягнення найвищої ефективності стиснення зображень та відео

Припустимо тепер, що вхідні дані підкоряються деяким відомий нам розподіл $p(x)$. Тоді за формулою Шеннона маємо середню кількість біт для збереження стиснених даних:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

Крім того, відзначимо, що в цьому випадку якість стиснення безпосередньо залежить від функції ймовірності. До речі, якщо ви розробите такий алгоритм, який буде вгадувати наступні дані з єдиною ймовірністю, кількість бітів, необхідних для передачі цієї інформації, буде 0.

Таким чином, нам потрібно знайти щільність ймовірності $P(x)$ з наявної інформації про характер розподілу вхідних даних. У найпростішому випадку доступною інформацією є попередні зображення, що стискаються, у заданому контексті. Іншими словами, потрібно знайти умовну ймовірність $P(x \mid X)$, де X – множина попередніх стислих зображень.

На цьому етапі може виникнути кілька помилкових уявлень:

1. Ви можете поступово стискати зображення та змінювати функцію ймовірності залежно від перших частин прочитаного зображення, і це буде ефективніше, ніж якщо ви не змінюєте функцію ймовірності та стискаєте все зображення. Відповідь: Функції ймовірності розраховуються для всіх можливих зображень, тому вони вже враховують те, що якщо, скажімо, одноколірні

зображення часто випадають, а зараз випав синій, то зображення майже напевно буде синім.

2. Існує нескінченна кількість різних зображень однакового розміру, навіть якщо розглядати певну область, і функція ймовірності майже завжди буде рівномірною. Відповідь: по-перше, якби це було так, то не було б більш оптимального алгоритму стиснення, ніж BMP, а по-друге, насправді різних зображень шалено мало в порівнянні з кількістю можливих.

Зауважте, що якщо ми розглядаємо лише стиснення зображень, які зараз зберігаються у світі на серверах, і не беремо до уваги генерацію та оновлення цих зображень, таких зображень точно не більше, ніж атомів у Всесвіті, яких приблизно 1082. У той же час кількість різних напівтонових чорно-білих зображень розміром 32x32 становить близько 102466. До речі, неважко помітити, що за відсутності оновлення зображень, навіть якщо вони рівномірно розподілені, кількість Біти Шеннона, необхідні для передачі будь-якого з цих зображень, мають порядок $log2$ 1082 = 266 (біт) = 33 (байти)

3. Можна розрахувати функцію ймовірності для певної області, в якій вона буде застосовуватися, і отриманий алгоритм стиснення буде теоретично кращим з усіх існуючих для цієї області. Відповідь: і так, і ні; теоретично це справді вірне твердження, але проблема практичної реалізації тут та ж, згідно з якою ми все ще не обмінюємося популярними зображеннями, стиснутими до рівня 33 байтів - необхідного обсягу пам'яті для зберігання результуючої функції ймовірності. Як було сказано вище, вже для чорно-білих зображень розміром 32x32 кількість різних зображень, для яких потрібно знати ймовірність випадання, становить близько 102466, зберегти таку кількість інформації на комп'ютері просто неможливо.

В результаті маємо наступні вимоги до розробленого алгоритму стиснення:

1. Адаптованість до контексту або здатність до навчання (алгоритм має бути динамічним, доступним для навчання, повинен враховувати попередні зображення під час стиснення наступних)

2. Ефективне зберігання ймовірностей (для великої кількості варіантів використання функцій обчислення його ймовірності по зображенню, об’єднання кількох зображень у групи з однаковою ймовірністю).

Глава 3. РОЗРОБКА АЛГОРИТМУ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СТИСНЕННЯ ЗОБРАЖЕНЬ ТА ВІДЕО

3.1 Реалізація алгоритму стиснення зображень та відео

В основі лежить клас, який відповідає за стиснення зображень. Процес стиснення у найпростішому випадку відбувається на дві дії — взяти значення ймовірності за ключом зображення зі словника і закодувати його з допомогою будь-якого статистичного алгоритму (арифметичне кодування, кодування Хаффмана).

Як ми вже з'ясували раніше, стиснення можливе тільки якщо ми припускаємо про те, що вхідні дані мають нерівномірний розподіл. Також ми вже затвердили загальну схему стиснення: перетворювач, квантувальник, кодер символів.

Ми спробуємо реалізувати один із найпопулярніших алгоритмів стиснення без втрат, JPEG2000. Як і JPEG, стандарт стиснення зображень JPEG 2000 складається з чотирьох основних етапів алгоритму - попередньої обробки, перетворення, квантування та кодування. На відміну від JPEG, крок квантування необов’язковий, якщо користувач бажає виконати стиснення без втрат. JPEG використовує розширену версію кодування Хаффмана, тоді як JPEG 2000 використовує новий метод кодування під назвою Вбудоване блочне кодування з оптимізованою обрізкою (EBCOT).

Крок 1 - Попередня обробка

Єдиним етапом попередньої обробки, який ми будемо використовувати, є центрування значень інтенсивності градацій сірого. Для цього ми віднімаємо 127 від кожного значення інтенсивності в матриці зображення. Якщо ми стискаємо кольорове зображення, ми спочатку перетворюємо його в простір YCbCr, а потім центруємо канали Y, Cb і Cr.

Крок 2 - Перетворення

Однією з головних змін у стандарті JPEG2000 є використання DWT (дискретного вейвлетного перетворення) замість DCT. Якщо ми виконуємо стиснення з втратами, ми використовуємо DWT з фільтром CDF97. Для стиснення

без втрат ми використовуємо DWT з фільтром LeGall53, обчисленим за допомогою методу підйому, розробленого Вімом Свелденсом. В обох випадках ми обчислюємо 2-3 ітерації DWT. У нашому прикладі ми рахуємо дві ітерації кожного перетворення.

Крок 3 - Кількісна оцінка

Квантувати DWT. Повнорозмірна версія. Якщо ми виконуємо стиснення без втрат, DWT кодується, і алгоритм завершує роботу. Для стиснення із втратами JPEG2000 використовує схему квантування, частково подібну до тієї, що використовується для блоків 8 x 8 у JPEG.

За дві ітерації DWT створює сім блоків, і кожен із цих блоків квантується окремо. Значення в кожному блоці або зсуваються в бік нуля, або перетворюються на нуль, а потім перетворюються в ціле число за допомогою функції стану. Дивіться підрозділ «Квантування» для отримання додаткової інформації про процес квантування. Результати для нашого поточного прикладу показано праворуч.

Крок 4 - Кодування

На останньому етапі стандарту стиснення ми використовуємо вбудоване блочне кодування з оптимізованим скороченням. Ми не розглядаємо тут EBCOT - метод вбудованого блочного кодування з оптимальним скороченням.

Щоб здійснити стиснення без втрат, ми можемо використовувати EBCOT (Embedded Block Coding with Optimized Truncation) для кодування елементів вейвлет-перетворення, який був згенерований фільтром LeGall. У цьому випадку ми можемо використовувати 215 544 біти для зберігання зображення. Оригінальне необроблене зображення потребує 307 200 біт пам’яті, тому підхід без втрат економить близько 30%. Ступінь нашого стиснення досягає аж до-5,6 біт на піксель.

Для стиснення з втратами дозволяє використовувати лише 84 504 біти замість початкового розміру $160 \times 240 \times 8 = 307200 \text{біт}$. Стиснення становить приблизно 2,2 біта на піксель. Якщо ми стиснемо те саме зображення за допомогою JPEG, нам знадобиться 103 944 біти, щоб досягти коефіцієнта стиснення 2,7 біта на піксель.

Необроблені зображення, зображення JPEG 2000 і зображення JPEG показані на рисунку 3.1.

Кодування Хаффмана - це метод стиснення даних, який можна використовувати для будь-якого типу даних. Цей алгоритм є ентропійним і базується на аналізі частоти входження символів у набірі даних.

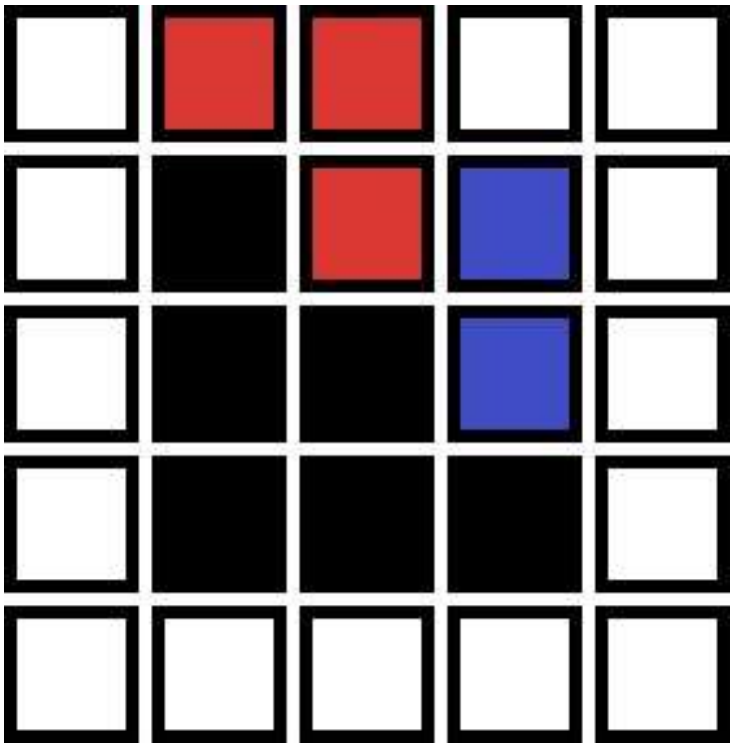


Рисунок 3.1 – Схематичне зображення частини зображення (у пікселях) [14]

Для спрощення припустимо, що ми працюємо з одновимірними даними (двовимірні дані можна зводити до одновимірних) і використовуємо лінійну функцію як перетворювач. В якості квантувача розглянемо округлення значень до найближчого цілого числа. Тоді процес стиснення можна представити у вигляді лінійного рівняння:

$$Ax = b$$

$$c = \mathrm{Coder}([b])$$

де x - вхідні дані, A - матриця перетворення, b - вектор значень, отриманий після застосування перетворення. [b] позначає округлення кожного значення вектора b до найближчого цілого числа. Coder - це функція кодування символів.

Спочатку ми підраховуємо, скільки разів кожен колір зустрічається на зображенні. Потім ми сортуємо кольори в порядку зменшення частоти. В результаті отримуємо рядок, схожий на рис. 3.2:

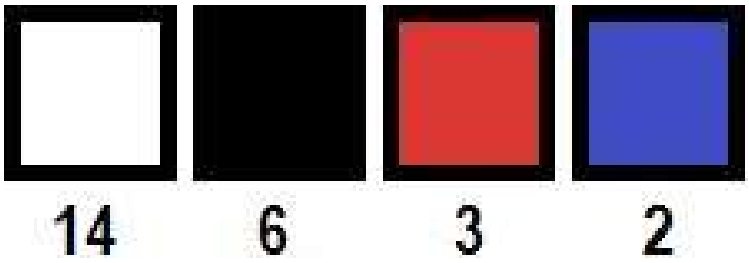


Рисунок 3.2 – Схема результатів, відсортована за кольором [15]

Важливою властивістю алгоритму стиснення є можливість декодування. Це означає, що кожне з перетворень, які застосовуються, повинно бути оборотним. Тобто ми повинні мати здатність відновлювати початкові дані зі стисненого вигляду.

Окрім того, перетворювачі повинні бути стійкими до невеликих змін вхідних даних. Якщо вхідні дані зазнають незначних змін, наприклад, змінюється вектор b в рівнянні Ax=b, то результат перетворення також повинен змінюватися незначно. Ця характеристика стійкості до невеликих змін описується числом обумовленості матриці, яке можна обчислити за допомогою певної формули.:

$$\mu(\mathrm{A}) = \|\mathrm{A}\| * \left\|\mathrm{A}^{-1}\right\|$$

Де ||*A*||- будь-яка операторна норма матриці. Значення кількості обумовленості потрібно мінімізувати.

Тепер ми з’єднуємо кольори, будуючи дерево так, щоб найдальший від кореня колір зустрічався рідше. Кольори з'єднані попарно, а з'єднання утворюють вузол. Вузол можна з’єднати з іншим вузлом або кольором. У нашому прикладі дерево може виглядати так:

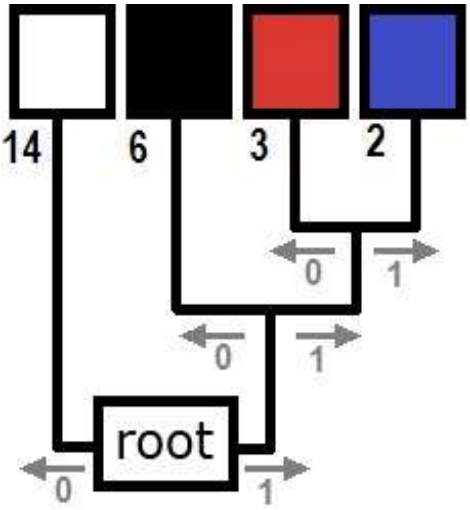


Рисунок 3.3- Дерево Хаффмана [16]

Наш результат називається деревом Хаффмана на рисунку 3.3. Його можна використовувати для кодування та декодування. Кожен колір закодований нижче. Ми генеруємо коди, розглядаючи кожен колір як лист дерева та визначаючи код для кожного з них. Якщо ми обертаємо праворуч у вузлі, ми пишемо 1, якщо ми обертаємо ліворуч -0. Цей процес створює таблицю кодів Хаффмана, у якій кожному символу присвоюється бітовий код, так що найпоширеніші символи мають найкоротші коди, а найменш поширені символи отримують найдовші коди, як показано на рисунку 3.4.

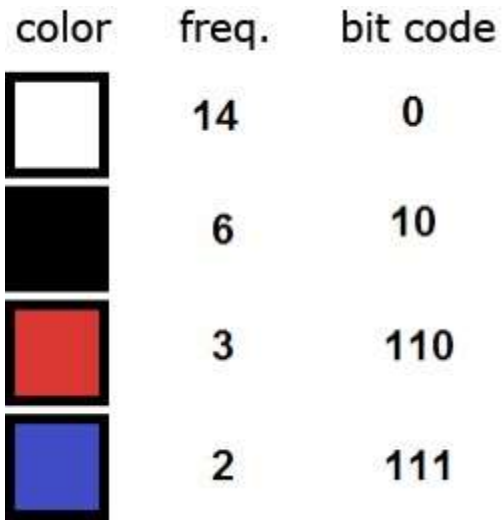


Рисунок 3.4 - Таблиця кодування[17]

Перетворення A необхідне забезпечення обнулення більшості останніх елементів вектора, інакше кажучи — необхідне “ущільнення енергії”.

Ми можемо розглядати будь-який кодувальник символів, який використовує відомі ймовірності символів. У простому випадку, можна використовувати кодувальник, який перед записом числа обчислює, скільки біт потрібно для збереження цього числа без зайвих нулів. Таким чином, для набору чисел від 0 до n, загальна кількість біт, які потрібні для збереження найбільшого числа з цього набору, можна обчислити за формулою:

$$[\log_2(\log_2(n))]+[\log_2(n)]$$

Розглянемо загалом процес декодування:

$$b+e=\text{Decoder}(c)$$
$$x_{dec}=A^{-1}(b+e)$$

Створені нами дерева Хаффмана та кодові таблиці не є єдиними можливими. Для нашого зображення ви можете створити альтернативне дерево Хаффмана, як показано на рисунку 3.5:

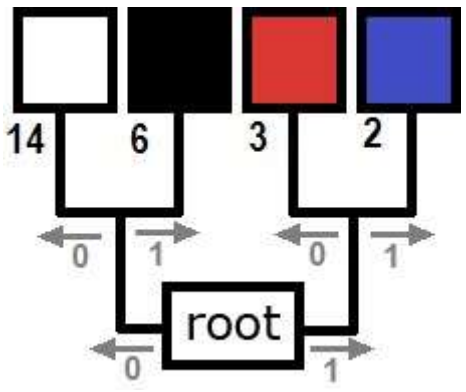


Рисунок 3.5 - Другий варіант таблиці Хаффмана [18]

Тоді відповідна кодова таблиця показана на рисунку 3.6:

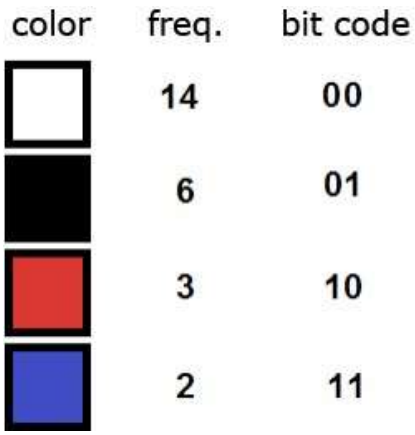


Рисунок 3.6 – Таблиця кодування другого видання [19]

У нашому прикладі краще використовувати перший варіант. Це тому, що це забезпечує краще стиснення для нашого конкретного зображення.

Кодування Хаффмана найбільш наочно демонструється стисненням растрового зображення. Припустимо, у нас є растрове зображення розміром 5×5 із 8-бітним кольором, тобто є 256 різних кольорів. Нестисне зображення займає $5 \times 5 \times 8 = 200$біт.

Варто також зауважити, що в базовій версії для зменшення кількості збережених ймовірностей ми будемо зберігати лише ймовірності для найбільш ймовірних зображень, тоді як для інших зображень ми будемо вважати ймовірності однаковими.

3.2 Розробка програмного забезпечення для стиснення зображень та відео

Як інструмент розв’язування задачі і реалізації алгоритму була обрана мова програмування Python.

Python - це універсальна, високорівнева, інтерпретована мова програмування, яка може використовуватися для різноманітних задач. Вона була створена Гвідо ван Россумом і вперше випущена у 1991 році. Python прославився своєю простотою, елегантністю та зручним синтаксисом, що дозволяє розробникам писати чистий і зрозумілий код. Він підтримує багато парадигм програмування,

включаючи процедурний, об'єктно-орієнтований та функціональний стиль програмування. Python має широке співтовариство розробників, активну підтримку та велику кількість бібліотек і модулів, що робить його однією з найпопулярніших мов програмування.

Приклад програми на мові програмування Python наведено нижче

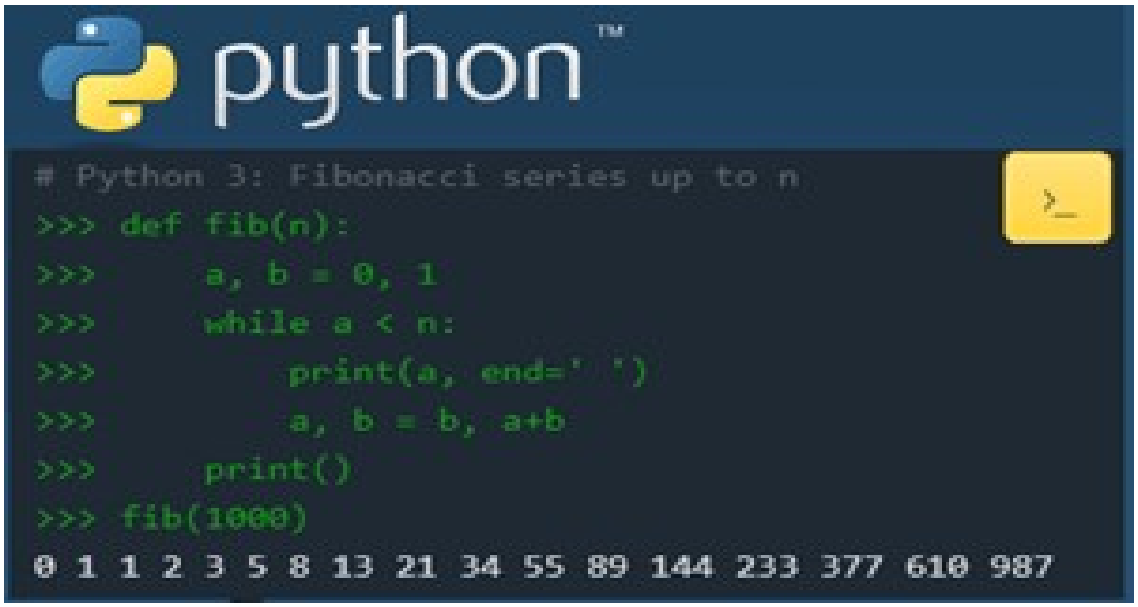


Рисунок 3.7 – Приклад коду на Python

Для розробки коду було обрано IDE PyCharm. PyCharm - це інтегроване середовище розробки (IDE) для мови програмування Python, створене компанією JetBrains. Воно надає розширений набір інструментів та функцій для розробки Python-програм, що допомагає збільшити продуктивність розробника. PyCharm має інтуїтивний інтерфейс, можливості автодоповнення коду, перевірки синтаксису, рефакторингу коду, налаштування середовища відповідно до ваших потреб та багато інших корисних функцій. Варіанти PyCharm включають Community Edition (безкоштовна версія) та Professional Edition (платна версія з додатковими можливостями).

Рисунок 3.8 — PyCharm

Для початку роботи були підключені наступні бібліотеки

```
import os

from datetime import timedelta
from pathlib import Path
import cv2
import _pickle as cPickle
import numpy as np
import pytesseract as tess
from PyQt6.uic.uiparser import QtGui

from PyQt6 import uic
from PyQt6.QtGui import QPixmap
from PyQt6.QtWidgets import QApplication, QFileDialog
from imutils import contours
import json
import sqlite3

from PIL import Image
import glob
```

Для завантаження файлу було створено окрема функція.

```
def choose():
    global image_file
    image_file = QFileDialog.getOpenFileName()
    image_file = image_file[0]
    form.label.setPixmap(QPixmap(image_file))
    form.label.setScaledContents(True)
```

Після обробки зображень, для виводу інформації було написана функція для стиснення зображення, код якої наведено й додатку А.

На кожну окрему кнопку була створена відповідна подія

```
form.pushButton.clicked.connect(choose);
form.pushButton_2.clicked.connect(pressure);
```

3.3 Опис функціоналу програмного забезпечення для стиснення зображень та відео

Для запуск програми необхідно відкрити файл, та запусти у PyCharm. Перед вами з’явиться наступна картина (рис. 3.10)

Рисунок 3.9 – початок програми

Далі натиснувши на кнопку «ЗАВАНТАЖИТИ», та вибрати необхідне зображення, пройде завантаження файлу (рис. 3.10)

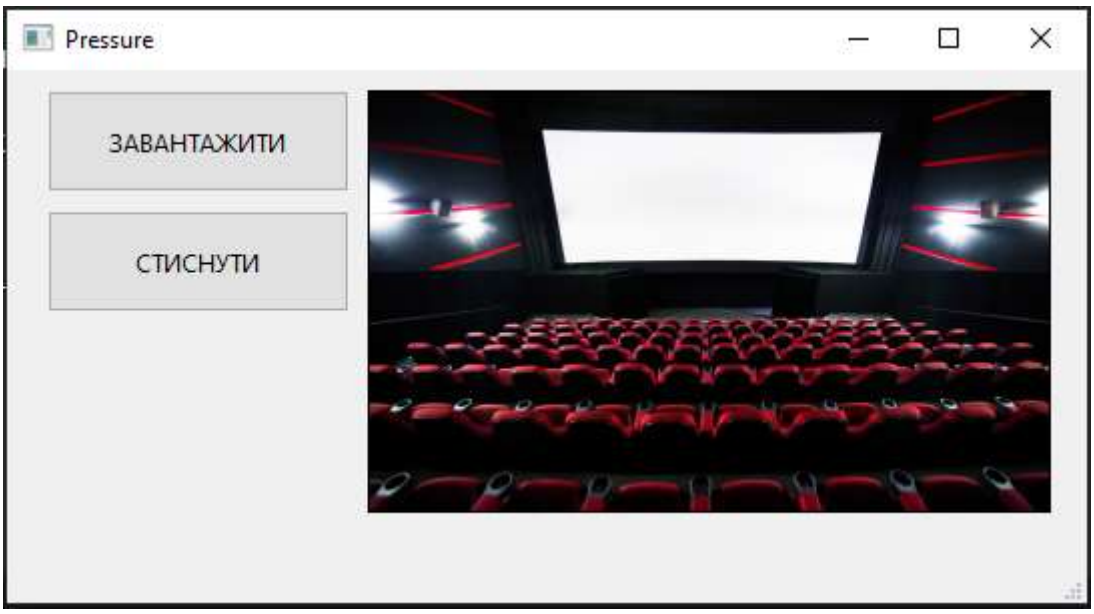


Рисунок 3.10 – Вибір файлу

Після чого можна зайнятися стисненням. Для цього натисніть на кнопку «СТИСНУТИ». На рисунку 3.11 зображено файл до стиснення та на рисунку 3.12 файл після стиснення

Рисунок 3.11 – Файл до стиснення

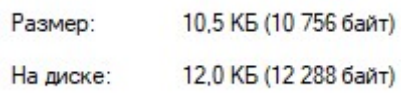


Рисунок 3.12 – Файл після стиснення

3.4 Результати тестування роботи програми

Під час тестування, усі знайдені помилки одразу ж виправлялися. Нижче наведено декілька прикладів тестування роботи програми.

Тест на стиснення зображення

Ідентифікатор тест-варіанту	Користувач стиснув файл
Набір вхідних даних	Вхідне зображення
Очікувані результати	Користувач успішно стискає зображення
Виконувані дії	Користувач додає зображення до програми і нажимає стиснути

Результат тесту наведено на рис. 3.14-3.16

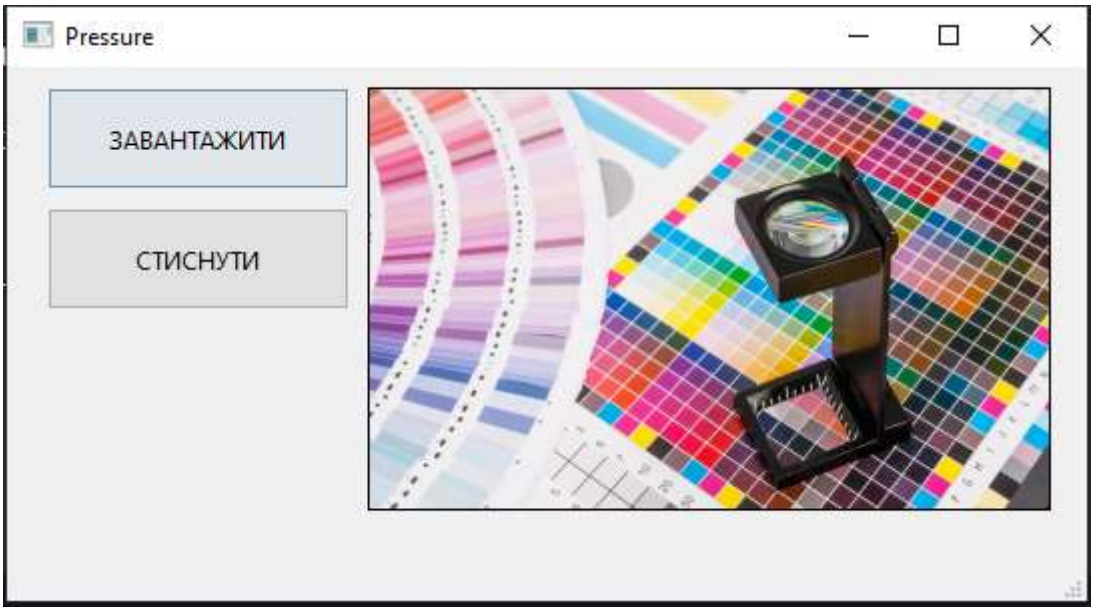


Рисунок 3.14 – результати тестування

Рисунок 3.15 – результати тестування

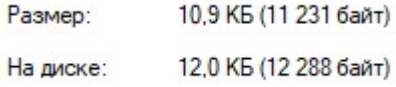


Рисунок 3.16 – результати тестування

Тест пройдено успішно.

За результатами наших тестів програмне забезпечення показало свою працездатність. Розроблений додаток пройшов всі тестування та показав відмінний результат.

ВИСНОВОК

Було досліджено базові принципи стиснення даних без втрат і з втратами. На основі цих принципів було сформовано концепцію універсального алгоритму стиснення даних без втрат. Ця концепція була обґрунтована з погляду теорії інформації та теорії множин, а також сингулярного розкладу. Крім того, було проведено інтенсивне дослідження у галузі стиснення зображень і відео, а також у ключових аспектах, пов'язаних з виявленням оптичного потоку, компенсацією руху відео та розпізнаванням об'єктів. За результатами цього дослідження була розроблена програма, спрямована на стиснення зображень.

В рамках цих досліджень було проведено ретельний аналіз і вивчено основні підходи до стиснення зображень і відео, а також розглянуто різні методи пошуку оптичного потоку, компенсації руху відео та розпізнавання об'єктів. Ці знання були використані для розробки програмного забезпечення, яке спроможне стиснути зображення з високою ефективністю. На основі отриманих знань і результатів досліджень була розроблена програма, яка спеціалізується на стисненні зображень. З метою узагальнення існуючих підходів до стиснення зображень був розроблений універсальний алгоритм, здатний навчатися вхідним даним, що робить його потенційно набагато більш ефективним, ніж існуючі стандартні алгоритми стиснення. Таким чином, даний алгоритм не гірший за перераховані вище, що є найбільш популярними алгоритмами стиснення зображень на сьогоднішній день.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Р. Вудс. Цифрова обробка зображень/Р. Вудс. - Київ: Техносфера, 2015;
2. Д.Селомон. Стиснення даних, зображень та звуку / Д.Селомон. - Київ: Техносфера, 2014;
3. Стиснення на основі передбачення значень пікселів. [Електронний ресурс]. - Режим доступу: https://www.csd.uwo.ca/~melsakka/publications/journals/pdfs/2007_jvcir_Nathan ael.pdf. - Дата доступу: 25.04.2023;
4. Стиснення на основі припущення схожості значень прилеглих у двовимірному значенні пікселів зображення. [Електронний ресурс]. - Режим доступу: https://naun.org/main/NAUN/computers/17-679.pdf. - Дата доступу: 25.04.2023;
5. Вейвлет перетворення стосовно зображень. [Електронний ресурс]. - Режим доступу: https://www.researchgate.net/publication/266018963_Wavelet_image_compressi on. - Дата доступу: 25.04.2023;
6. Стандарт JPEG. [Електронний ресурс]. - Режим доступу: https://web.stanford.edu/class/ee398a/handouts/lectures/08-JPEG.pdf. - Дата доступу: 25.04.2023;
7. Стандарт JPEG2000. [Електронний ресурс]. - Режим доступу: https://www.imaging.org/site/PDFS/Papers/2000/PICS-0-81/1645.pdf. - Дата доступу: 25.04.2023;
8. Детальний опис JPEG. [Електронний ресурс]. - Режим доступу: https://www.researchgate.net/publication/268523100_THE_JPEG_IMAGE_COMPRESSION_ALGORITHM. - Дата доступу: 25.04.2023;
9. Опис SIFT, SURF, ORB. [Електронний ресурс]. - Режим доступу: https://arxiv.org/pdf/1710.02726.pdf. - Дата доступу: 25.04.2023;

10. Опис дескриптора FREAK [Електронний ресурс]. - Режим доступу: https://www.researchgate.net/publication/258848394_FREAK_Fast_retina_keypo int . - Дата доступу: 25.04.2023;
11. База даних медичних знімків, що використовується для тестів. [Електронний ресурс]. - Режим доступу: http://www.aylward.org/notes/open-access-medical-image-repositories. - Дата доступу: 25.04.2023;
12. CIFAR-10 [Електронний ресурс]. - Режим доступу: https://www.cs.toronto.edu/~kriz/cifar.html. - Дата доступу: 25.04.2023;
13. Semantic Perceptual Image Compression using Deep Convolution Networks [Електронний ресурс]. - Режим доступу: https://arxiv.org/pdf/1612.08712v2.pdf. - Дата доступу: 25.04.2023;
14. An End-to-End Compression Framework Based on Convolutional Neural Networks [Електронний ресурс]. - Режим доступу: https://arxiv.org/pdf/1708.00838v1.pdf - Дата доступу: 25.04.2023;
15. Practical Full Resolution Learned Lossless Image Compression [Електронний ресурс]. — Режим доступу: https://openaccess.thecvf.com/content_CVPR_2019/papers/Mentzer_Practical_Full_Resolution_Learned_Lossless_Image_Compression_CVPR_2019_paper.pdf. - Дата доступу: 25.04.2023.

ДОДАТКИ

Додаток А(main.py)

```
from pathlib import Path
import cv2
import scipy
import numpy as np
import sys
np.set_printoptions(threshold=sys.maxsize)

from PIL import Image

from numpy import r_
from scipy import fftpack
import math
np.set_printoptions(precision=3)
np.set_printoptions(suppress=True)

from PyQt6 import uic
from PyQt6.QtGui import QPixmap
from PyQt6.QtWidgets import QApplication, QFileDialog

Form, Window = uic.loadUiType("pressure.ui")

app = QApplication([])
window = Window()
form = Form()
form.setupUi(window)
window.show()

image_file = ''


def choose():
    global image_file
    image_file = QFileDialog.getOpenFileName()
    image_file = image_file[0]
    form.label.setPixmap(QPixmap(image_file))
    form.label.setScaledContents(True)


np.set_printoptions(precision=3)
np.set_printoptions(suppress=True)

DEBUG = False
im = cv2.imread("2.jpg", 1)

b, g, r = cv2.split(im)
```

```
def dct2(a):
    x = scipy.fftpack.dct(scipy.fftpack.dct(a, axis=0, norm='ortho'), axis=1, norm='ortho')
    return x


imsize = r.shape
dct_r = np.zeros(imsize)
dct_g = np.zeros(imsize)
dct_b = np.zeros(imsize)

for i in r_[:imsize[0]:8]:
    for j in r_[:imsize[1]:8]:
        dct_r[i:(i + 8), j:(j + 8)] = dct2(r[i:(i + 8), j:(j + 8)])
        dct_g[i:(i + 8), j:(j + 8)] = dct2(g[i:(i + 8), j:(j + 8)])
        dct_b[i:(i + 8), j:(j + 8)] = dct2(b[i:(i + 8), j:(j + 8)])

thresh = 0.04


def thresholding(x):
    x = x * (abs(x) > (thresh * np.max(x)))
    return x


dct_r = thresholding(dct_r)
dct_b = thresholding(dct_b)
dct_g = thresholding(dct_g)


def run_len():
    i, j = 0, 0
    up = True
    while True:
        yield (i, j)
        if i == j == 7:
            break
        if up:
            if i == 0 and j != 7:
                j += 1
                up = False
            elif j == 7 and i != 7:
                i += 1
                up = False
            elif (j != 7 and i != 0):
                i -= 1
                j += 1
        else:
            if j == 0 and i != 7:
                i += 1
                up = True
            elif i == 7 and j != 7:
                j += 1
                up = True
            elif (j != 0 and i != 7):
                j -= 1
                i += 1
    yield 0
```

```
def compress_block(x):
    gen = run_len()
    old = None
    cnt = 1
    y = np.zeros((8, 8))
    y[:len(x), :len(x[0])] = x
    x = y
    fin = []
    for i in range(8):
        for j in range(8):
            a, b = next(gen)
            num = int(x[a][b])
            if (old != num):
                fin.append((old, cnt))
                cnt = 1
                old = num
            else:
                cnt += 1
    fin.append((old, cnt))
    return fin[1:]


def print_sample():
    x = r[204:212, 204:212]
    x = dct2(x)
    x = thresholding(x)
    x = compress_block(x)



# Huffman encoding for channel compression
def compress_channel(channel_data, file_name):
    # Huffman encoding start
    symbol_dict = {}

    class node:
        def __init__(self, freq, symbol, left=None, right=None):
            self.freq = freq
            self.symbol = symbol
            self.left = left
            self.right = right
            self.huff = ''

    def printNodes(node, val=''):
        newVal = val + str(node.huff)
        if (node.left):
            printNodes(node.left, newVal)
        if (node.right):
            printNodes(node.right, newVal)
        if (not node.left and not node.right):
            symbol_dict[node.symbol] = newVal
```

```
unique_char = set(channel_data)
    counter = {}
    for char in unique_char:
        counter[char] = channel_data.count(char)

    chars = [i for i in counter]
    freq = [counter[i] for i in counter]
    nodes = []
    total = sum(freq)
    for x in range(len(chars)):
        nodes.append(node(freq[x], chars[x]))

    while len(nodes) > 1:
        nodes = sorted(nodes, key=lambda x: x.freq)
        left = nodes[0]
        right = nodes[1]
        left.huff = 0
        right.huff = 1
        newNode = node(left.freq + right.freq, left.symbol + right.symbol, left, right)
        nodes.remove(left)
        nodes.remove(right)
        nodes.append(newNode)

    printNodes(nodes[0])
    temp = [[counter[i], i] for i in counter]
    temp.sort(reverse=True)
    Rx = sum([len(symbol_dict[i[1]]) * (i[0] / total) for ind, i in enumerate(temp)])

    Hx = sum([(-i[0] / total) * (math.log(i[0] / total, 2)) for ind, i in enumerate(temp)])

    bin_str = ""
    for i in channel_data:
        bin_str += symbol_dict[i]


    x = len(bin_str) % 8
    if x != 0:
        bin_str += "0" * (8 - x)

    ch_code = bytes()
    for i in range(0, len(bin_str) - 1, 8):
        num = int(bin_str[i: i + 8], base=2)
        _chr = num.to_bytes(1, 'little')
        ch_code += _chr
    metadata[file_name] = [symbol_dict, len(ch_code)]
    if DEBUG:
        with open(file_name, "wb") as f:
            f.write(ch_code)
    return ch_code
```

```
def pressure():
    global path
    image = Image.open(image_file)
    path = Path(image_file)

    global thresh, metadata, red, green, blue, imsize, dct_r, dct_g, dct_b, im, b, g, r, DEBUG

    np.set_printoptions(precision=3)
    np.set_printoptions(suppress=True)

    DEBUG = False
    im = cv2.imread(path.name, 1)

    b, g, r = cv2.split(im)

    imsize = r.shape
    dct_r = np.zeros(imsize)
    dct_g = np.zeros(imsize)
    dct_b = np.zeros(imsize)

    for i in r_[:imsize[0]:8]:
        for j in r_[:imsize[1]:8]:
            dct_r[i:(i + 8), j:(j + 8)] = dct2(r[i:(i + 8), j:(j + 8)])
            dct_g[i:(i + 8), j:(j + 8)] = dct2(g[i:(i + 8), j:(j + 8)])
            dct_b[i:(i + 8), j:(j + 8)] = dct2(b[i:(i + 8), j:(j + 8)])

    thresh = 0.04

    dct_r = thresholding(dct_r)
    dct_b = thresholding(dct_b)
    dct_g = thresholding(dct_g)

    if DEBUG:
        print_sample()

    red = []
    green = []
    blue = []
    for i in r_[:imsize[0]:8]:
        r_block = []
        g_block = []
        b_block = []
        for j in r_[:imsize[1]:8]:
            r_block.append(compress_block(dct_r[i:(i + 8), j:(j + 8)]))
            g_block.append(compress_block(dct_g[i:(i + 8), j:(j + 8)]))
            b_block.append(compress_block(dct_b[i:(i + 8), j:(j + 8)]))
        red.append(r_block)
        green.append(g_block)
        blue.append(b_block)

    if DEBUG:
        with open("r.txt", "w") as f:
            f.write(str(red).replace("(", "").replace(")", "").replace(" ", ""))

    red = str(red).replace("(", "").replace(")", "").replace(" ", "")
    green = str(green).replace("(", "").replace(")", "").replace(" ", "")
    blue = str(blue).replace("(", "").replace(")", "").replace(" ", "")
```

```
bytedata = bytes()
    metadata = {}
    metadata['imsize'] = imsize
    metadata['thresh'] = thresh
    bytedata += compress_channel(red, 'red')
    bytedata += compress_channel(green, 'green')
    bytedata += compress_channel(blue, 'blue')

    bytedata = bytes(str(metadata).replace(" ", ""), "ascii") + b'\x00' + bytedata


    image_data=bytedata

    for index, char in enumerate(image_data):
        if char == 0:
            break

    metadata = image_data[:index]
    metadata = metadata.decode("ascii")
    data = eval(metadata)
    red, green, blue = data['red'], data['green'], data['blue']
    imsize = data['imsize']
    thresh = data['thresh']

    r_data = image_data[index + 1: index + 1 + red[1]]
    g_data = image_data[index + 1 + red[1]: index + 1 + red[1] + green[1]]
    b_data = image_data[index + 1 + red[1] + green[1]: index + 1 + red[1] + green[1] + blue[1]]


    r_matrix = huffman_decode(r_data, red[0])
    g_matrix = huffman_decode(g_data, green[0])
    b_matrix = huffman_decode(b_data, blue[0])

    r_matrix = eval(r_matrix.split("]]]")[0] + "]]]")
    g_matrix = eval(g_matrix.split("]]]")[0] + "]]]")
    b_matrix = eval(b_matrix.split("]]]")[0] + "]]]")


    r_channel = decode_channel(r_matrix)
    g_channel = decode_channel(g_matrix)
    b_channel = decode_channel(b_matrix)

    for i in r_[:imsize[0]:8]:
        for j in r_[:imsize[1]:8]:
            r_channel[i:(i + 8), j:(j + 8)] = idct2(r_channel[i:(i + 8), j:(j + 8)])
            g_channel[i:(i + 8), j:(j + 8)] = idct2(g_channel[i:(i + 8), j:(j + 8)])
            b_channel[i:(i + 8), j:(j + 8)] = idct2(b_channel[i:(i + 8), j:(j + 8)])

    x = cv2.merge((np.array(np.clip(b_channel, 0, 255), dtype=np.uint8),
                   np.array(np.clip(g_channel, 0, 255), dtype=np.uint8),
                   np.array(np.clip(r_channel, 0, 255), dtype=np.uint8)))

    cv2.waitKey(0)
    cv2.destroyAllWindows()
    cv2.imwrite(f'nature_compressed_{thresh}.jpg', x)
```

```
def huffman_decode(ch_data, symbol_dict):
    binary = ""
    symbol_dict = {symbol_dict[i]: i for i in symbol_dict}

    for i in ch_data:
        binary_num = bin(i)[2:]
        binary_num = "0" * (8 - len(binary_num)) + binary_num
        binary += binary_num
    data = ""
    curr = ""
    for i in binary:
        curr += i
        x = symbol_dict.get(curr, None)
        if x:
            curr = ""
            data += x
    return data

def run_len():
    i, j = 0, 0
    up = True
    while True:
        yield (i, j)
        if i == j == 7:
            break
        if up:
            if i == 0 and j != 7:
                j += 1
                up = False
            elif j == 7 and i != 7:
                i += 1
                up = False
            elif (j != 7 and i != 0):
                i -= 1
                j += 1
        else:
            if j == 0 and i != 7:
                i += 1
                up = True
            elif i == 7 and j != 7:
                j += 1
                up = True
            elif (j != 0 and i != 7):
                j -= 1
                i += 1
    yield 0
```

```
def decompress_block(x):
    gen = run_len()
    index = 0
    num = x[0]
    max_cnt = x[1]
    cnt = 0
    y = np.zeros((8, 8))

    for i in range(8):
        for j in range(8):
            a, b = next(gen)
            y[a][b] = num
            cnt += 1
            if (cnt == max_cnt and ((i, j) != (7, 7))):
                cnt = 0
                index += 2
                num = x[index]
                max_cnt = x[index + 1]
    return y


def decode_channel(channel):
    fin = []
    for ind1, i in enumerate(channel):
        x = []
        for ind2, j in enumerate(i):
            x.append(decompress_block(j))
        x = np.concatenate(x, axis=1)
        fin.append(x)
    fin = np.concatenate(fin, axis=0)
    fin = fin[:imsize[0], :imsize[1]]
    return fin

def idct2(a):
    return fftpack.idct(fftpack.idct(a, axis=0, norm='ortho'), axis=1, norm='ortho')

form.pushButton.clicked.connect(choose);
form.pushButton_2.clicked.connect(pressure);

app.exec()
```