Документација за проектот по курсот Дигитално процесирање на слика, со наслов:

„Алгоритми за креирање на дигитален мозаик“

Виктор Костадиноски[1],

226009, Компјутерски науки

Абстракт

Мозаик претставува вид на уметничка изработка која вклучува прецизно поставување на мали парчиња од некаков вид на камен или, пак, плочки од префарбани материјали, како што се стаклото или керамиката, со цел добивање на една крајна, јасна и препознатлива слика. Во овој проект, за разлика од физичките мозаици, ќе се фокусираме на изработката на дигитални мозаици и алгоритмите со помош на коишто се постигнува истото.

Дигиталниот мозаик, за разлика од физичкиот, претставува дигитална слика со каков било мотив – личност, предмет, некакво пејзаж, апстрактна појава и сл., којашто е добиена врз основа на различни математички пресметки со спојување на голем број на помали сликички добиени од едно конечно множество слики. Доколку би го споредиле со физичкиот мозаик, големиот број на помали сликички од дигиталниот мозаик, одговараат на плочките од камен коишто се употребуваат во рамките на физичкиот. Токму затоа, малите сликички од коишто ќе го формираме дигиталниот мозаик ќе ги нарекуваме со едно име – плочки.

Во овој проект, ќе објасниме, разработиме и подобриме алгоритам којшто се користи за креирање на дигитални мозаици врз основа на сличноста во боја. За таа цел, влезната слика, односно сликата којашто сакаме да ја добиеме со спојување на големиот број мали сликички (плочки), ќе ја поделиме на голем број региони, коишто ќе бидат со иста големина како и малите сликички (плочките). Потоа секој регионот од влезната слика ќе ги споредиме со малите сликички (плочките) и ќе ја најдеме онаа која според бојата е најблиска до него. Така добиените мали сликички (плочки) ќе ги споиме по правилен редослед и ќе добиеме еден крајна слика којашто е составена и изградена од голем број плочки, односно ќе добиеме дигитален мозаик.

Во рамките на проектот ќе разработиме, исто така, и повеќе начини и видови на подобрувања на првичниот алгоритам. Овие подобрување вклучуваат мали измени на првичниот алгоритам, како што: употреба на рачно напишани функции за пресметка на растојанија помеѓу средната вредност на бојата помеѓу две слики, употребата на ограничен случаен број плочки во процесот на избирање на најсоодветна (најблиска) плочка на даден регион во сликата (рандомизација), употреба на к-димензионални дрва и употреба на multithreading во процесот на избирање на најсоодветна (најблиска) плочка на даден регион во сликата.

За крај, ќе извршиме повеќе тестови со цел да ги определиме параметрите на алгоритамот коишто ќе доведат до најбрзо и најсоодветно креирање на дигитален мозаик.

1. Вовед

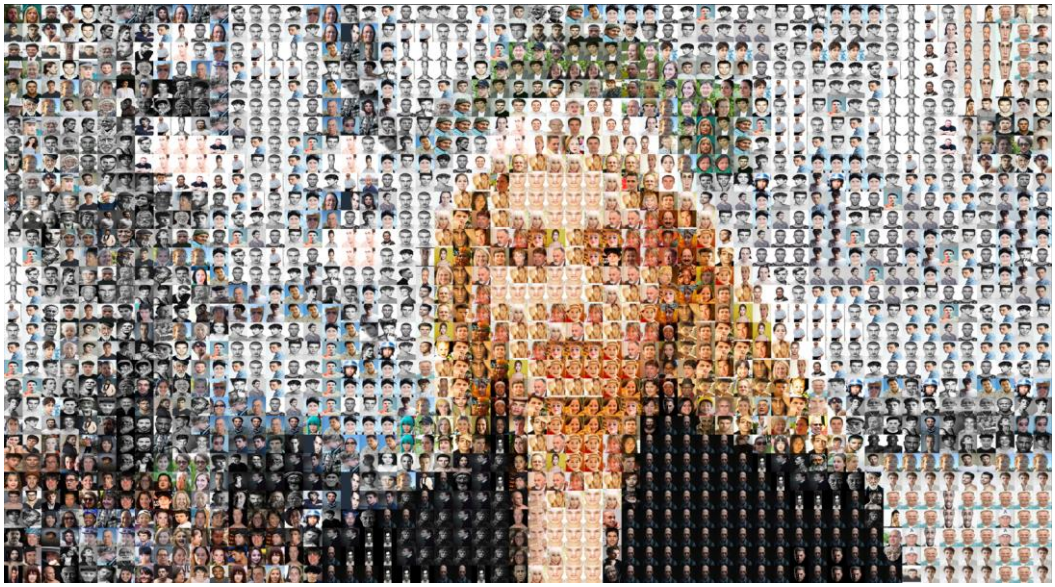
Мозаикот е уметничка форма која вклучува создавање слики со составување мали, обоени, парчиња материјали како камен, стакло или керамика. Овие мали парчиња, кои најчесто се нарекуваат плочки, се внимателно наредени за да формираат поголема, кохезивна и јасна слика. Историски гледано, мозаиците се користеле во различни култури, особено во античкиот Рим и византиската уметност. Како што можеме да видиме на Слика 1, мозаиците во римската уметност се користеле за украсување на подови, ѕидови и тавани на важни згради.

Во дигиталното процесирање на слика, концептот на мозаик е видоизменет, иако ја задржува основната суштина – креирање на слика од голем број на мали делови. Наместо физички материјали, дигитален мозаик ги заменува регионите на влезната сликата со помали, претходно дефинирани мали сликички – плочки. Целта е да се зачува целокупната структура, семантичко значење и визуелната содржина на оригиналната слика, со користење на множество помали плочки за прикажување на различните региони.

Слика 1: Римски таван украсен со мозаик на риби

1.1Начин на креирање на дигитални мозаици

Дигиталните мозаици се разликуваат во начинот на креирање од традиционалните мозаици. Додека во креирањето на традиционалните мозаици до израз доаѓа креативната моќ и способност на уметникот да го согледа, замисли и формира крајниот мозаик со мудро, рационално и креативно поставување на парчињата камен на правилните положби со цел добивање крајната слика, во формирањето на дигиталните мозаици се користат пресметковни алгоритми, кои врз основа на математички пресметки за блискот помеѓу две слики, најчесто во боја или текстура, ја формираат крајната слика – мозаикот. Во дигиталните мозаици, влезната слика се дели на региони кои се со иста големина како и плочките, а потоа со помош на алгоритамот за сличност помеѓу две слики, од конечно дефинирано множество со плочки се избира онаа која е најслична, односно најблиска, според бојата или текстурата, до дадениот регион во влезната слика. Вака избраните плочки се комбинираат и дигитално „залепуваат“ и се добива посакуваниот дигитален мозаик. Пример за дигитален мозаик можеме да видиме на Слика 2.

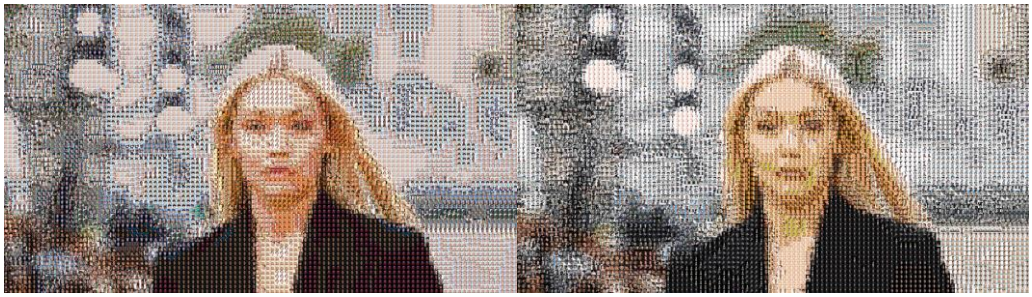


Слика 2: Дигитален мозаик на слика од девојка од плочки со лица на луѓе

1.2Уметнички и технички причини за креирање на дигиталните мозаици

Уметничките и техничките причини за создавање на дигитални мозаици варираат. Од уметничка гледна точка, дигиталните мозаици нудат уникатна и естетска претстава на влезната слика, комбинирајќи ја технологијата со креативниот израз коишто најмногу може да се забележи и издефинира преку изборот на уметникот во множеството на плочки, како и нивната големина, коишто ќе се употребуваат за формирање на крајната слика – мозаикот. Ваквиот креативен избор можеме да го согледаме на Слика 3. Дополнително, дигиталните мозаици често се користат за украсни цели, создавајќи стилизирани верзии на фотографии или уметнички дела.

Од техничка перспектива, дигиталните мозаици се користат во различни апликации со цел да се овозможи визуелизација на податоците, така што со секоја плочка ќе се претстави различен податок, или пак, компресија на слики, каде што кај големите слики, деловите коишто се повторуваат се сумирани и се претставуваат со помали елементи. Во рамките на овој проект, ние ќе се задржиме на техничките детели поврзани со алгоритмите за креирање на дигитални мозаици, начините на нивна временска оптимизација и избор на параметрите на овие алгоритми за најбрзо и естетски најсоодветно креирање на дигитални мозаици.



Слика 3: Дигитален мозаик на иста влезна слика добиен со две различни множества на плочки

2. Дефинирање на проблемот

Крајната и најважната цел на еден алгоритам за креирање на дигитални мозаици е токму тоа – создавање на дигитален мозаик од влезна слика. За да овозможиме решавање на оваа задача, мораме да ги дефинираме основните карактеристики на алгоритамот со цел побрзо и поедноставно постигнување на целта.

Основниот предизвик е генерирање на мозаик којшто ќе ја задржи целокупната структура, семантика и визуелни детали на оригиналната влезна слика и ќе го направи тоа преку користење на ефикасен пресметковен алгоритам. Во надминувањето на овој предизвик, ќе разгледаме повеќе карактеристики на самиот проблем и алгоритамот којшто ќе се користи за негово решавање.

2.1 Избор на влезната фотографија

Првата задача е изборот на влезна фотографија, којашто ќе се користи како извор врз основа на којшто алгоритамот ќе го формира крајниот мозаик. Се разбира, алгоритамот мора да работи за секаков тип на фотографии. Големината и самата семантика на влезната фотографија не смеат да претставуваат проблем за алгоритамот. Алгоритамот би морал да биде способен да создаде дигитален мозаик и од апстрактни фотографии со димензии од размерот на 200x200 пиксели, па се до најреалистични портрети со димензии од размерот на 5000x6000 пиксели. Во однос на самиот дигитален формат во којшто е зачувана влезната слика, во рамките на овој проект ќе разгледуваме само фотографии зачувани како .jpg или .png формат, иако за алгоритамот воопшто не би смеело да претставува проблем да обработи фотографии зачувани и во друг дигитален формат, со извршување на соодветни конверзии.

2.2Избор на множеството плочки

Откако ќе ја избереме влезната слика, следно што треба да направиме е да го избереме множеството на плочки коишто ќе се користат за замена на регионите од влезната слика. Како што кажавме претходно, во изборот на множеството на плочки доаѓа до израз креативниот избор на дигиталниот уметник кој го креира дигиталниот мозаик, но од тука па понатаму ние ќе ги разгледуваме само пресметковните бенефити и недостатоци од изборот на различни множества со плочки.

Идеално гледано, за нас и за нашиот алгоритам за создавање на дигитални мозаици, ни е да имаме такво множество со плочки, такво што за даден регион од влезната слика, во множеството плочки ќе имаме плочка чијашто разлика во боја со дадениот регион е еднаква на 0. За да го постигнеме ова, во множеството би требало да имаме онолку плочки колку што имаме бои во сите региони од сликата. Се разбира, иако ова е возможно и би можеле да го постигнеме со доволно големо множество на плочки, тоа нема да го направиме бидејќи во тој случај просторната и временската сложеност на алгоритамот би била неверојатно голема (A). Наместо тоа, ќе избереме множество такво што разликата во бојата помеѓу регионите од влезната слика и плочките од множеството ќе тежнее кон 0. Бидејќи секоја влезна слика е различна и во себе опфаќа региони со различни бои, оваа разлика во боја помеѓу регионите и плочките, за некои влезни слики би била поблиска до нулата, а за некои подалечна. Тоа ми можеле да го надминеме така што ќе користиме различни множества на плочки за различни влезни

фотографии, врз основа на боите присутни во регионите на влезната фотографија. Сепак, во рамките на предложениот алгоритам ќе користиме само едно множество со 2211 плочки кое дава повеќе од задоволителни резултати за најголемиот број влезни фотографии. Тематика на плочките од множеството е фотографии од лица на луѓе поставени пред позадини со различни бои.

2.3Избор на големината на регионите и плочките

По изборот на влезната фотографија и множеството плочки, следниот избор што треба да го направиме е големината на регионите во сликата и големината на плочките од множеството со плочки. Иако овие големини не би морало да се еднакви, во рамките на предложениот алгоритам ќе разгледуваме случаеви само кога регионите од влезната фотографија се со иста големина како и плочките од множеството со плочки (Б).

Изборот на оваа големина зависи од повеќе фактори, и тоа: 1) изворната (максимална) големината на плочките, 2) јаснотијата во детали во рамките на добиениот мозаик, 3) брзината на извршување на алгоритамот и 4) големината на влезната фотографија,.

Пред да преминеме на понатамошно објаснување, би го дефинираме поимот регион во рамките на алгоритамот за креирање на дигитални мозаици. **Регион** (или област) претставува многу мал дел од влезната фотографија којшто ќе се замени со една и точно една плочка од множеството со плочки врз основна на нивната сличност во боја. Пример, доколку би кажале дека од влезна фотографија со големина од 1920x1080 пиксели е добиен дигитален мозаик со големина на регион (или големина на плочка, бидејќи во рамките на овој алгоритам тие се исти) од 8x8 пиксели, тоа би значело дека секоја област од 8x8 пиксели (вкупно 64 пиксели) од влезната фотографија е заменета со една и точно една плочка од множеството со плочки според нивната сличност во боја. Односно, во влезна фотографија со големина од 1920x1080 пиксели има точно 32 400 региони со големина од 8x8 пиксели. Тоа би значело дека мозаикот добиен со оваа поставеност на алгоритамот е составен од 32 400 плочки (мали сликички) секоја со големина од 8x8 пиксели. Крајната големината на добиениот мозаик нема да се промени, односно и тој ќе биде со големина од 1920x1080 пиксели. Оваа големина може сосема малку да се промени (во размер од неколку пиксели) доколку се користи различен начин на пополнување на краевите на мозаикот (3) .

Откако го објаснивме поимот регион можеме да дадеме осврт на различните фактори коишто влијаат во изборот на големината на плочките, односно регионите.

2.3.1 Изборната (максимална) големина на плочките

Изборот на големина на регион од горе е „фигуративно“ ограничен од изворната (максималната) големина на плочките. Па така на пример, доколку изборната големина на сите плочките од множеството со плочки е 128x128 пиксели, поставувањето на големината на регион на големина поголема од 128x128 пиксели би довело до непотребно растегнување на плочките, а со тоа и до непотребно губење на естетскиот квалитет на крајниот мозаик. Токму затоа велиме дека изборот на големина на регион е од горе ограничен (фигуративно) од изворната големина на плочките.

2.3.2 Јаснотија во детали во добиениот мозаик

Големината на регионот е еден од трите фактори (другите два се е големината на влезната фотографија и самото множество на плочки) коишто директно влијаат на јаснотијата во детали во добиениот мозаик. Имено, колку големината на регионот е помала, толку јаснотијата во деталите во добиениот мозаик е поголема. Бидејќи при мали големини на региони, бројот на региони во фотографијата е значително поголем, толку повеќе се зголемува бројот на плочки од коишто е составен мозаикот. Ова е директна компарација со бројот на пиксели на една фотографија: колку бројот на пиксели од коишто е составена фотографијата е поголем, толку јаснотијата во деталите од фотографијата е подобра. Всушност, со регионите во рамките на мозаикот, ние само правиме едно кластерирање (групирање) на пикселите од влезната фотографија и пронаоѓање на плочка којашто има средна вредност на боја најслична со средната вредност на боја на тој регион.

2.3.3 Брзината на извршување на алгоритамот

Брзината на извршување на алгоритамот е директно условена од големината на регион. Колку големината на регион е помала, толку бројот на региони во влезната слика е поголем. Бидејќи за секој регион треба да ја најдеме најсличната плочка според боја до него, тоа значи дека директно се зголемува бројот на споредби што треба да се направат. Така ако го земеме претходно споменатиот пример со влезна фотографија со големина од 1920x1080 пиксели и големина на регион од 8x8 пиксели, тоа би значело дека би имале 32 400 региони, односно 32 400 споредби. Сега доколку големината на регион ја намалиме дупло, односно имаме регион со големина од 4x4 пиксели, бројот на региони ќе се зголеми на 129 600. Тоа се 4 пати повеќе споредби коишто алгоритамот треба да ги направи. Токму затоа велиме дека брзината на извршување на алгоритамот е директно условена од големината на регион.

2.3.4 Големината на влезната фотографија

Големината на влезната фотографија, исто така, влијае на изборот на големина на регион. Доколку влезната фотографија е со премногу голема резолуција, а големина на регионот е премногу мала, тоа би довело до непотребно зголемување на бројот на региони, а со тоа и на времето на извршување на алгоритамот, без преголеми промени во јаснотијата во детали во добиениот мозаик. Од друга страна, пак, доколку големината на влезната фотографија е премногу мала, а големина на регионот е премногу голема тоа би довело до значително губење на деталите во добиениот мозаик, до степен мозаикот да ја загуби целокупната структура, семантика и визуелни детали на оригиналната влезна слика. Токму затоа, за правилниот избор на големина на регион врз основа на големина на влезната фотографија, ќе спроведеме неколку експерименти (4.3).

2.4Избор на параметрите на алгоритамот за креирање дигитални мозаици

Дигиталното процесирање на фотографии со големи димензии е пресметковно скапа операција. Токму затоа правилниот избор на параметри на алгоритамот за креирање на дигитални мозаици е особено важно. Правилниот избор на параметри може да доведе

до намалување на времето потребно да се создаде еден дигитален мозаик дури и за 120 пати. Во рамките на предложениот алгоритам ќе разгледаме параметри коишто ќе овозможат поставување на најразлични опции на алгоритамот и коишто ќе доведат до менување на времето на извршување (зголемување и намалување) како и менување на естетската убавина (подобрување и влошување) на крајно добиениот мозаик. Овие параметри вклучуваат опции за: употребата на ограничен случаен број плочки со цел намалување на просторот на пребарувања во процесот на избирање на најблиска плочка на даден регион во сликата (рандомизација), употреба на к-димензионални дрва со цел намалување на времето потребно за избирање на најсоодветна плочка и употреба на multithreading со цел паралелизација на процесот на избирање најблиска плочка.

3. Детален опис на основниот алгоритамот за креирање дигитални мозаици

Во овој дел ќе дадеме детален опис и објаснување на предложениот алгоритам за креирање на дигитален мозаик. Ќе започнеме со основната имплементација, а потоа истата ќе ја надоградиме и подобриме со додавање на повеќе опции, за на крајот да добиеме ефикасен и подобрен алгоритам којшто нуди поставување на повеќе различни параметри со цел добивање брзи и естетски прифатливи резултати

Предложениот алгоритам е напишан во програмскиот јазик Python. Во рамките на алгоритамот ќе користиме повеќе надворешни помошни библиотеки и една помошна класа и тоа: библиотеката за работа со слики во Python – OpenCV, библиотеката за брзи нумерички пресметки - Numpy, библиотека која нуди готова имплементација на К-димензионални дрва – SciPy, модулот os – нуди функции за работа со директориуми и фајлови, модулот random – нуди функции за случајно генерирање на броеви и елементи од некаква колекција, модулот time – овозможува мерење на времето на извршување на назначен код, модулот concurrent – нуди функции за имплементација на multrithreading и мојата помошна класа – ImageProcessor. За секоја од функциите коишто ги нудат готовите библиотеки, а се употребува во рамките на алгоритамот за креирање на мозаик ќе биде дадено детално објаснување како ја извршува својата задача.

Откако овие детали ни се познати, можеме да почнеме со објаснување на имплементацијата на алгоритамот. За почеток ќе креираме една единствена класа со име MosaicProcessor во чии рамки ќе ги апстрахираме сите функционалности на алгоритамот. Конструкторот на оваа класа ќе прима за почеток 2 параметри: image_processor - инстанца од класата ImageProcessor и tiles_path – стринг којшто во себе ја содржи патеката до директориумот каде се зачувани плочките коишто ќе ги користи алгоритамот (default вредност "./tiles"). Во рамките на класата ќе креираме еден метод create_mosaic којшто ќе прима два параметри image_filepath – стринг којшто ја содржи патеката до влезната слика и tiles_size – торка која што ја содржи информацијата за големината на плочките, односно регионите (default вредност (16,16)). Методот create_mosaic ќе врати една вредност – крајниот креиран мозаик како numpy низа која потоа со користење на функцијата imshow од библиотеката OpenCV ќе можеме да ја прикажеме на екран или, пак, да ја зачуваме со користење на функцијата imwrite, исто од библиотеката OpenCV. Почетниот код изгледа вака:

```python
import concurrent.futures
import os.path
import random
import time
import cv2
import numpy as np
from scipy.spatial import KDTree
from ImageProcessor import ImageProcessor


class MosaicProcessor:  1 usage  new *
    __DEFAULT_TILE_SIZE = (16, 16)
    __DEFAULT_TILES_FOLDER = f"{os.path.abspath(os.getcwd())}\\tiles"

    def __init__(self, image_processor: ImageProcessor, tiles_path=__DEFAULT_TILES_FOLDER):  new *
        self.image_processor = image_processor
        self.tiles_path = tiles_path

    def create_mosaic(self, image_filepath, tiles_size=__DEFAULT_TILE_SIZE):  4 usages (3 dynamic)  new *
        return mosaic


if __name__ == '__main__':
    image_processor = ImageProcessor()
    mosaic_processor = MosaicProcessor(image_processor)
    tiles_size = (33, 33)
    img_filepath = "./test_images/both/girl.jpg"
    mosaic_image = mosaic_processor.create_mosaic(img_filepath, tiles_size)
    image_processor.show_image(mosaic_image,  win_name: "Mosaic Picture")
```

Притоа, функцијата os.path.abspath(os.getcwd()) ја враќа назад апсолутната патека до моменталниот директориум, а функцијата image_processor.show_image(mosaic_image, "Mosaic Picture") го прикажува добиениот мозаик на екран во прозорец со наслов „Mosaic Picture“.

Следно што ќе имплементираме е вчитување на плочките во програмата, така што ќе креираме речник со следните парови клуч и вредност: {tile_filepath: tile}, каде што tile_filepath е стринг којшто ја содржи целосната патека до плочката, а tile претставува самата вчитана плочка во програмата како numpy низа. На овој начин сите плочки ќе бидат вчитани во меморијата на програмата и ќе имаме брз случаен пристап до нив. Плочките ќе бидат вчитани во соодветната наведена големина (tiles_size). Дополнително, вака пополнетиот речник ќе го зачуваме и на диск како датотека со име {tiles_size[0]}x{tiles_size[1]}-tiles-database.pickle Следен пат кога ќе се обидеме да извршиме вчитување на плочките во меморија, најпрво ќе провериме дали постои датотеката {tiles_size[0]}x{tiles_size[1]}-tiles-database.pickle. Притоа, доколку постои ќе извршиме директно вчитување на речникот од диск во меморија, а доколку не постои ќе го креираме, пополниме и запишеме на диск за следно користење. За оваа цел ќе го напишеме методот __populate_and_get_tiles_database којшто ќе прими два параметри tiles_path - стринг којшто во себе ја содржи патеката до директориумот каде се зачувани плочките и tiles_size – големината на плочките како торка, а ќе го врати претходно објаснетиот речник. Методот ќе го повикаме во create_mosaic, а резултатот ќе го сместиме во нова променлива на ниво на класа: self.__tiles

```python
def __populate_and_get_tiles_database(self, tiles_path, tiles_size):  2 usages (1 dynamic)  new *
    database_filepath = f"{os.path.abspath(os.getcwd())}\\databases\\{tiles_size[0]}x{tiles_size[1]}-tiles-database.pickle"

    if self.image_processor.file_exists(database_filepath):
        return self.image_processor.load_from_file(database_filepath)

    print("Calculating and populating tiles database...")
    tiles_names = self.image_processor.read_image_names(tiles_path)
    tiles = dict()
    new_tile_size = tiles_size[1], tiles_size[0]
    for tile_name in tiles_names:
        tile = cv2.imread(tile_name, 1)
        tile = cv2.resize(tile, new_tile_size)
        tiles[tile_name] = tile

    self.image_processor.save_to_file(tiles, database_filepath)
    return tiles
```

Притоа, методот file_exists од класата image_processor проверува дали датотеката со патека database_filepath постои. Ако постои, ќе изврши вчитување на датотеката преку повик на методот load_from_file. Методот read_image_names од класата image_processor ги вчитува имињата на сите плочи од наведената патека. Методот save_to_file од класата image_processor го запишува наведениот речник tiles на локација database_filepath како .pickle датотека со користење на модулот pickle којшто овозможува лесна, брза и едноставна серијализација и десеријализација на Python објекти. Функциите imread и resize од библиотеката OpenCV, овозможуваат вчитување на слика во меморија како numpy матрица и промена на нејзината големина, соодветно.

За да ја објасниме следната функција што ќе ја имплементираме мораме прво да разјасниме како ќе ја пресметуваме бојата на регион во влезната слика и бојата на плочката.

Секоја слика вчитана со функцијата imread е претставена како повеќе димензионална матрица, т.е. како низа која е составена од други низи – редови. Секој ред, исто така, во себе содржи низи. Овие низи, пак, содржат 3 целобројни броеви (се со должина 3) и, всушност, секоја од овие низи претставува еден пиксел. Па, така редот има онолку низи во себе (пиксели) колку што има пиксели во еден ред од сликата. Трите целобројни броеви со коишто е претставен секој пиксел, пак, се всушност трите вредности на синиот, зелениот и црвениот канал (BlueGreenReed-BGR).

Регион, кажавме дека претставува групација (област) од повеќе пиксели во влезната слика. Бидејќи е составен од повеќе пиксели, не можеме да избереме еден пиксел и да кажеме дека тоа ќе биде бојата на тој регион од фотографијата. Токму затоа ќе примениме друга техника: ќе ја пресметаме средна вредност од сите пиксели во регионот, на секоја од трите канали и така добиената тројка од средни вредности ќе ја сметаме за **боја на регионот**. Пример да замислиме дека имаме 2x2 регион (вкупно 4 пиксели) со следните вредности:

[[[255, 0, 0], [255, 0, 0]],
[[0, 255, 0], [0, 255, 0]]]

Во овој случај, средната вредност на секој од каналите ќе ја пресметаме како:
Синиот канал: (255 + 255 + 0 + 0) / 4 = 127.5
Зелениот канал: (0 + 0 + 255 + 255) / 4 = 127.5
Црвениот канал: (0 + 0 + 0 + 0) / 4 = 0

За крај бојата на дадениот регион би била: [127.5, 127.5, 0]. Истата техника ќе ја искористиме и за пресметување на боја на секоја од плочките. **Всушност кога ќе кажеме боја на регион и боја на сликичка, мислиме на средната вредност од сите пиксели во регионот/плочката, на секој од трите канали.**

Откако знаеме како ќе ја пресметуваме бојата на регионите и на плочките, ќе го имплементираме метод __populate_and_get_average_database. Овој метод ќе прима два параметри tiles – речник од тип {tile_filepath: tile} и tiles_size – големина на плочките, а ќе врати речник од тип {tile_average_color: tile_filepath}, каде tile_average_color е бојата на плочката, а tile_filepath е стринг со целосната патека до плочката. На овој начин во меморија ќе ја имаме бојата на секоја од плочките коишто ќе ги користиме.

Дополнително, вака пополнетиот речник ќе го зачуваме и на диск како датотека со име {tiles_size[0]}x{tiles_size[1]}-average-database.pickle Следен пат кога ќе се обидеме, повторно да ја пресметаме бојата на секоја од плочките, најпрво ќе провериме дали постои фајлот {tiles_size[0]}x{tiles_size[1]}-average-database.pickle и притоа доколку постои ќе извршиме директно вчитување на речникот од диск во меморија, со тоа енормно убрзувајќи го нашиот алгоритам, бидејќи ја отстрануваме потребата за повторно и повторно пресметување на бојата на плочките. Доколку датотеката не постои, бидејќи користиме големина на плочки за којашто претходно ги немаме веќе еднаш направено потребните пресметки, речникот ќе го креираме, пополниме и запишеме на диск за следно користење. Методот ќе го повикаме во create_mosaic, а резултатот ќе го сместиме во нова променлива на ниво на класа: self.__index_table.

```
def __populate_and_get_average_database(self, tiles, tiles_size):  2 usages (2 dynamic)  new *
    database_filepath = f"{os.path.abspath(os.getcwd())}\\databases\\{tiles_size[0]}x{tiles_size[1]}-average-database.pickle"

    if self.image_processor.file_exists(database_filepath):
        return self.image_processor.load_from_file(database_filepath)

    print("Calculating and populating average database...")
    index_table = dict()
    for tile_filepath, tile in tiles.items():
        tile_average_color = self.image_processor.average_color(tile)
        index_table[tuple(tile_average_color)] = tile_filepath

    self.image_processor.save_to_file(index_table, database_filepath)
    return index_table
```

Притоа, методот average_color од класата ImageProcessor ја пресметува бојата на плочката според претходно опишаниот начин преку повик на готовата функција mean од библиотеката numpy. Рачната имплементација на функција која е значително поспора во извршување можеме да ја видиме подолу со име average_color_slow.

```
# This approach is more efficient because it uses NumPy's optimized array operations.
def average_color(self, image):  8 usages (3 dynamic)  ltonkdong
    return np.mean(image, axis=(0, 1))

def average_color_slow(self, image):  new *
    image_copy = image.astype(np.float64)
    blue_sum = 0
    green_sum = 0
    red_sum = 0
    pixel_count = 0

    for row in image_copy:
        for pixel in row:
            blue_sum += pixel[0]
            green_sum += pixel[1]
            red_sum += pixel[2]
            pixel_count += 1

    return blue_sum / pixel_count, green_sum / pixel_count, red_sum / pixel_count
```

Следно ќе додадеме нова променлива на ниво на класа: self.__override_database којашто ќе се предава како параметар на конструкторот (со default вредност False) и којашто ќе му овозможи на корисникот лесно повторно ре-пополнување и зачувување на двете претходно споменати датотеки {tiles_size[0]}x{tiles_size[1]}-average-database.pickle и {tiles_size[0]}x{tiles_size[1]}-tiles-database.pickle (пример во случај на користење на друго множество со плочки). Соодветните промени ќе направиме и во методите __populate_and_get_tiles_database и __populate_and_get_average_database.

Уште само 3 методи ни се потребни за почетно функционирање на алгоритамот за креирање на дигитални мозаици, а тоа се методите: __get_regions, get_mosaic_size и find_best_match_for_region_base.

Методот __get_regions ќе прима 2 параметри: image – влезната слика и tiles_size – големината на плочките (т.е. големина на регионот, бидејќи во оваа имплементација, како што кажавме, регионите и плочките се со иста големина), а ќе врати листа со сите региони на сликата, пополнета со региони движејќи се од врвот на сликата, кон дното, а хоризонтално од лево кон десно.

Методот __get_mosaic_size ќе прима 3 параметри: image_size – големината на влезната слика, tiles_size – големината на плочките/регионите и option – опција на пополнување на рабовите на мозаикот, а ќе ја врати големината на мозаикот што треба да го изградиме. Параметарот option прима една од две константни вредности: KEEP_SAME_SIZE (или 1) и FILL_BORDERS (или 2) .

Пред да објасниме што овие опции значат, треба да напоменеме дека нашиот алгоритам, при креирање на мозаик со зададена големина на плочки секогаш ќе се обиде да користи плочки со зададената големина, односно никогаш нема да се обиде да ја промени назначената големина на плочката.

Да го разгледаме сега следниот случај: Да претпоставиме дека имаме влезна слика со големина 5x2 пиксели и користиме плочки со големина од 2x2 пиксели. Доколку ја користиме опцијата KEEP_SAME_SIZE, мозаикот којшто би го добиле, симболично поедноставен би изгледал вака:

1 1 2 2 X
1 1 2 2 X

каде што 1 и 2 се имињата на некои плочки, а со X се означени црните (празни) пиксели. На овој начин мозаикот ја задржал оригиналната големина како и влезната слика, но бидејќи користиме плочки со големина 2x2, за последниот регион од првиот и вториот ред недостасува уште еден пиксел во којшто би била вметната соодветната 2x2 плочка. Како што напоменавме, алгоритамот нема да се обиде да ја промени големина на плочката во 1x2 со цел да ја вметни во достапниот просто. Доколку, пак, ја користиме опцијата FILL_BORDERS мозаикот којшто би го добиле, симболично поедноставен би изгледал вака:

1 1 2 2 3 3
1 1 2 2 3 3

каде што 1, 2 и 3 се имињата на некои плочки, а со X се означени црните (празни) пиксели. Со оваа опција мозаикот ќе има различна големина во споредба со влезната

слика (големина на мозаикот сега е 6x2), но на краевите од мозаикот нема да добиеме црни (непополнети) пиксели кои произлегле од нецелобројното делење на големина на влезната слика со големината на регионот и строгото неменување на големината на плочката, дури и во случаеви кога тоа е потребно.

Методот __find_best_match_for_region_base ќе прима еден параметар: region – регионот за којшто ќе ја бараме најблиската плочка, а ќе ја врати најблиската (според боја) плочка од множеството плочки.

Тука за првпат ќе дефинираме како ќе пресметуваме блискоста (сличноста, растојанието) помеѓу две слики (во случајов регион и плочка). Да претпоставиме дека ги имаме регионот R чијашто боја е изнесува [B1, G1, R1]=[100, 150, 200] и плочката P чијашто боја изнесува [B2, G2, R2]=[120, 130, 180]. Растојанието, односно оддалеченоста во бојата помеѓу овие две слики ќе ја дефинираме на следниот начин:

$$\text{Оддалеченост } = \sqrt{(B2 - B1)^2 + (G2 - G1)^2 + (R2 - R1)^2}$$

Колку вредноста на оддалеченоста е помала, толку двете слики R и P се поблиски (послични).

Откако ја дефиниравме оддалеченоста (односно сличноста) помеѓу две слики, можеме да ја дадеме имплементацијата на методот __find_best_match_for_region_base. Сè што ќе направи овој метод е линеарно изминување на сите плочки вчитани во меморија и враќање на онаа плочка чијашто оддалеченост со проследениот регионот е најмала.

Конкретните имплементациите на методите __get_regions, get_mosaic_size и find_best_match_for_region_base се следните:

```
def __get_regions(self, image, tiles_size):  1 usage (1 dynamic)  new *
    height, width, depth = image.shape
    tile_height, tile_width = tiles_size
    regions = [image[y:y + tile_height, x:x + tile_width] for y in range(0, height, tile_height) for x in
               range(0, width, tile_width)]
    return regions
```

```
def get_mosaic_size(self, image_size, tiles_size, option):  1 usage (1 dynamic)  new *
    height, width, depth = image_size

    match option:
        case MosaicProcessor.KEEP_SAME_SIZE:
            return height, width, depth
        case MosaicProcessor.FILL_BORDERS:
            tile_height, tile_width = tiles_size
            if height % tile_height != 0:
                mosaic_height = ((height // tile_height) * tile_height) + tile_height
            else:
                mosaic_height = ((height // tile_height) * tile_height)

            if width % tile_width != 0:
                mosaic_width = ((width // tile_width) * tile_width) + tile_width
            else:
                mosaic_width = ((width // tile_width) * tile_width)

            return mosaic_height, mosaic_width, depth
```

```python
def find_best_match_for_region_base(self, region):  3 usages (3 dynamic)  new *
    region_average = self.image_processor.average_color(region)
    closest_tile = None
    min_distance = float("inf")
    for tile_average, tile_name in self.__index_table.items():
        distance = self.image_processor.calculate_distance(region_average, tile_average)
        if distance < min_distance:
            min_distance = distance
            closest_tile = self.__tiles[tile_name]

    return closest_tile
```

Притоа, методот calculate_distance ја пресметува оддалеченоста помеѓу двете проследени бои region_average и tile_average на претходно опишаниот начин.

```python
# This approach is around 3.5 times faster
def calculate_distance(self, image_avg1, image_avg2):  2 usages  ltonkdong
    return ((image_avg1[0] - image_avg2[0]) ** 2 + (image_avg1[1] - image_avg2[1]) ** 2 + (image_avg1[2] - image_avg2[2]) ** 2) ** 0.5

def calculate_distance_slow(self, image_avg1, image_avg2):  new *
    return np.linalg.norm(image_avg1 - image_avg2)
```

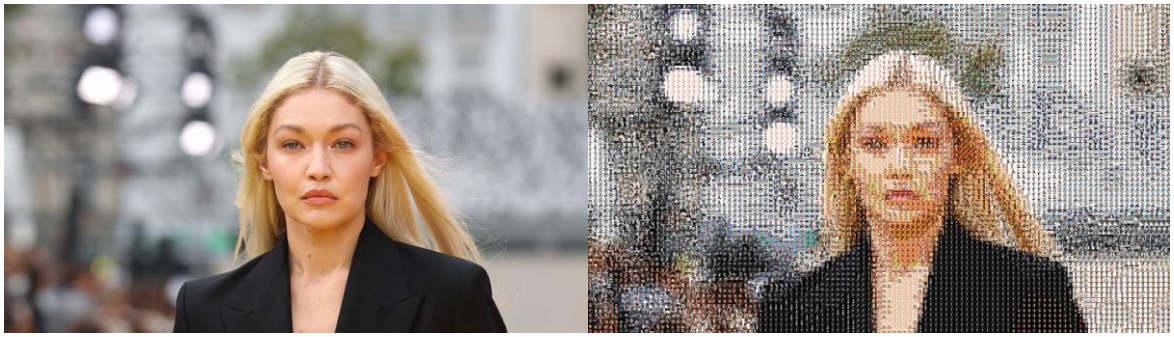
За истата цел може да се користи и функцијата calculate_distance_slow којашто прави повик до готовата функција norm од пакетот linalg од библиотеката numpy, но моите експерименти покажаа дека оваа функција е за околу 3.5 пати поспора од рачно имплементираната. За таа цел се користи функцијата calculate_distance, а не calculate_distance_slow.

Откако ги дадовме описот, дефинициите и имплементациите на сите претходно споменати методи, можеме да ја дадеме конечната почетна имплементација на алгоритамот за креирање на дигитални мозаици преку целосната почетна имплементацијата на методот create_mosaic.

```python
def create_mosaic(self, image_filepath, tiles_size=__DEFAULT_TILE_SIZE, fill_option=KEEP_SAME_SIZE,):  1 usage  new *
    if tiles_size[0] > 128 or tiles_size[1] > 128:
        raise Exception("Tiles size cannot be greater than 128")

    self.__tiles = self.__populate_and_get_tiles_database(self.tiles_path, tiles_size)
    self.__index_table = self.__populate_and_get_average_database(self.__tiles, tiles_size)

    print("Creating a mosaic...")
    image = cv2.imread(image_filepath, 1)

    start_time = time.time()
    image_regions = self.__get_regions(image, tiles_size)

    mosaic_shape = self.get_mosaic_size(image.shape, tiles_size, fill_option)
    mosaic = np.zeros(mosaic_shape, dtype=np.uint8)
    mosaic_height, mosaic_width, depth = mosaic_shape
    tile_height, tile_width = tiles_size

    i = 0
    for y in range(0, mosaic_height, tile_height):
        for x in range(0, mosaic_width, tile_width):

            if fill_option == MosaicProcessor.KEEP_SAME_SIZE:
                if image_regions[i].shape[1] != tile_width or image_regions[i].shape[0] != tile_height:
                    i += 1
                    continue

            best_tile = self.find_best_match_for_region_base(image_regions[i])
            mosaic[y:y + tile_height, x:x + tile_width] = best_tile
            i += 1

    self.last_execution_time = time.time() - start_time

    return mosaic
```

Притоа класната променлива self.__last_execution_time чува во себе информации за времето на последното извршување на алгоритамот.

Почетниот алгоритам можеме да го истестираме така што ќе се обидеме да добиеме мозаик на дадена влезна слика. Резултатот од алгоритамот и влезната слика можеме да ги видиме на Слика 4. Големината на влезната слика изнесува 3608x2030 пиксели, а големината на регионот е 32x32 пиксели. Времето на извршување на алгоритамот изнесува околу 7.65 секунди. Притоа датотеките 32x32-average-database.pickle и 32x32 -tiles-database.pickle веќе постоеа и беа само вчитани од диск во меморија. Времето за оваа операција (како и времето да се креираат овие датотеки доколку не постојат) во овој пример и во сите следни нема да се смета во времето на извршување на алгоритамот.



Слика 4: Влезната слика и добиениот мозаик со големина на регион од 32x32 пиксели и начин на пополнување=FILL_BORDERS

3.1 Подобрување на претходниот алгоритам со воведување на рандомизација

Пред да продолжиме понатаму, само ќе напоменеме дека во сите понатамошни испитувања, тестирања и експерименти доколку не е поинаку нагласено ќе се употребува влезната слика girl.jpg со големина од 3608x2030 зачувана во директориумот /test_images/large/ во рамките на проектот. Како истата изгледа можеме да видиме лево на Слика 4.

Времето на извршување на алгоритамот од 7.65 секунди можеби и не изгледа многу, но тоа време драстично се зголемува при креирање на мозаици со помали региони. Така, времето на извршување се зголемува на 30.39 секунди во случаеви кога користиме регион со големина од 16x16 пиксели. Во случаеви, пак, кога користиме регион со големина од 8x8 пиксели, тоа време изнесува 134.01 секунди. Токму затоа потребни ни се одредени подобрувања на почетниот предложен алгоритамот за креирање на дигитални мозаици.

Првичното подобрување на алгоритамот е со воведување на рандомизација. Имено, во основниот алгоритам за да ја пронајдеме најблиската плочка на даден регион, потребно е да ја изминеме секоја можна плочка, да ја пресметаме нивната оддалеченост и да ја земеме онаа чијашто оддалеченост е најмала. Па така, доколку во една фотографија имаме N региони и M плочки, временската сложеност на основниот алгоритам изнесува O(NM). Оваа временска сложеност е доста лоша и токму затоа сакаме да ја подобриме – со воведување на рандомизација.

Рандомизацијата значи дека при определувањето на најблиска плочка на даден регион, место да го изминуваме целото множество плочки, ќе изминеме само R случајно избрани плочки. R-те случајно избрани плочки ќе бидат различни за секој регион со цел да му дадеме на алгоритамот што е можно поголема шанса за избор на посоодветна плочка. Што ќе се случи ако рандомизацијата ја правиме на ниво на влезна слика, место на ниво на регион?

Да претпоставиме дека имаме два региони R1 и R2 чијашто боја е B. Нека најблиска плочка до регионите R1 и R2 е плочката P1 со боја B1. Нека втората најблиска плочка до регионите R1 и R2 е плочката P2 со боја B2. Со рандомизацијата ние ѝ даваме подеднаква шанса и на плочката P1 и да плочката P2 да биде избрана во новото подмножеството со плочки.

Да претпоставиме дека имаме рандомизација на ниво на регион. Нека при рандомизацијата за регионот R1 е избрана плочката P2, а при рандомизацијата за регионот R1 е избрана плочката P1. Тоа значи дека подобрувањето со рандомизација на ниво на регион, алгоритамот направил неоптимален избор само во 1 од 2 случаеви (при изборот на плочката за регионот R1).

Нека сега имаме рандомизација на ниво на влезна слика и нека при рандомизацијата е избрана плочката P2. Тоа значи дека при подобрувањето со рандомизација на ниво на влезна слика, алгоритамот направил неоптимален избор и во двата случаеви (и при изборот на плочка за регионот R1 и при изборот за регион R2). Токму затоа ќе користиме оптимизација со рандомизација на ниво на регион со цел да ги намалиме неоптималните избори коишто алгоритамот би можел да ги направи.

Оптимизација со рандомизација лесно можеме да ја имплементираме на следниот начин:

```
def create_mosaic(self, image_filepath, tiles_size=__DEFAULT_TILE_SIZE, fill_option=KEEP_SAME_SIZE,  1 usage
                  randomization_improvement=False,
                  randomization_value=__DEFAULT_RANDOMIZATION_VALUE,
                  ):
    if tiles_size[0] > 128 or tiles_size[1] > 128:
        raise Exception("Tiles size cannot be greater than 128")

    self.__tiles = self.__populate_and_get_tiles_database(self.tiles_path, tiles_size)
    self.__index_table = self.__populate_and_get_average_database(self.__tiles, tiles_size)

    if randomization_value > len(self.__index_table.keys()) or randomization_value < 0:
        randomization_improvement = False

    if randomization_improvement:
        self.__save_index_table = self.__populate_and_get_average_database(self.__tiles, tiles_size)

    print("Creating a mosaic...")
    image = cv2.imread(image_filepath, 1)

    start_time = time.time()
    image_regions = self.__get_regions(image, tiles_size)

    mosaic_shape = self.get_mosaic_size(image.shape, tiles_size, fill_option)
    mosaic = np.zeros(mosaic_shape, dtype=np.uint8)
    mosaic_height, mosaic_width, depth = mosaic_shape
    tile_height, tile_width = tiles_size
```

```
i = 0
for y in range(0, mosaic_height, tile_height):
    for x in range(0, mosaic_width, tile_width):

        if randomization_value == 0:
            break

        if fill_option == MosaicProcessor.KEEP_SAME_SIZE:
            if image_regions[i].shape[1] != tile_width or image_regions[i].shape[0] != tile_height:
                i += 1
                continue

        if randomization_improvement:
            random_index_keys = random.sample(list(self.__save_index_table.keys()), randomization_value)
            self.__index_table = {key: self.__save_index_table[key] for key in random_index_keys}

        best_tile = self.find_best_match_for_region_base(image_regions[i])
        mosaic[y:y + tile_height, x:x + tile_width] = best_tile
        i += 1

self.last_execution_time = time.time() - start_time

return mosaic
```

Притоа, променливата self.__save_index_table чува копија на self.__index_table, а функцијата sample од модулот random избира randomization_value – случајно избрани плочки од целосното множество на плочки, при пронаоѓањето на најблиска плочка за секој регион.

Времињата на извршување на алгоритамот со подобрување со рандомизација со randomization_value = 30, при користење на плочки со големина од 32x32, 16x16 и 8x8 пиксели изнесуваат: 0.42 секунди, 1.47 секунди и 5.31 секунди, соодветно. Како што може да се забележи, имаме драстични подобрување на времињата на извршување на алгоритамот и тоа: од 7.65 на 0.42 секунди, од 30.39 од на 1.47 секунди и од 134.01 на 5.31 секунди.

Сепак, иако имаме подобрување на времето на извршување на алгоритамот имаме намалување на естетскиот квалитет на добиениот мозаик. Мозаиците добиени со алгоритамот со подобрување со рандомизација со randomization_value = 30 и randomization_value = 230, со големина на регион од 32x32 пиксели, можеме да ги видиме на Слика 5.

Слика 5 Мозаик добиен со алгоритам со подобрување со рандомизација. randomization_value = 30 (лево), randomization_value = 230 (десно)

За крај на овој дел, треба да напоменеме дека колку вредноста на randomization_value се зголемува, толку времето на извршување на алгоритамот се влошува, но крајниот мозаик е со подобрен естетски резултат. Доколку randomization_value се изедначи или, пак, стани поголем од големината на оригиналното множество со плочки, времето на извршување на алгоритамот ќе се изедначи со она кое се добиваше пред подобрувањето со рандомизација. Дури, пак, ова време може да е и полошо и од времето на извршување на основниот алгоритам поради дополнителниот overhead којшто овој подобрен алгоритам го има во случајниот избор на плочки од оригиналното множество. Токму затоа методот create_mosaic е имплементиран на тој начин што ако забележи вакво пречекорување на вредноста на randomization_value, ќе ја исклучи целосно опцијата за подобрување со рандомизација.

3.2Подобрување на алгоритамот со употреба на к-димензионални дрва

Воведувањето на рандомизација значително го подобри времето на извршување на алгоритамот, но како цена за тоа се влоши квалитетот и естетскиот изглед на крајниот мозаик. За да го надминеме овој недостаток на подобрувањето со рандомизација, ќе го измениме предложениот алгоритам така што место линеарна споредба на секој регион со секоја плочка од множеството со плочки, ќе употребиме к-димензионално (во случајов 3-димензионално) дрво. Тоа ќе ја намали сложеноста на алгоритамот од O(NM), на O(N$\log_2$M), каде N е бројот на региони на сликата, а M е бројот на плочки во целосното множество со плочки.

Пред да поминеме на имплементацијата на подобрувањето со к-димензионални дрва, најпрво ќе ги објасниме основните карактеристики на к-димензионалните дрва.

3.2.1 Основни карактеристики на к-димензионалните (КД) дрва

КД-дрвата се тип на податочна структура – бинарно дрво што се користи за организирање точки од к-димензионален простор. Секој јазол од дрвото претставува точка од к-димензионален просторот. Секој јазол (односно секоја к-димензионална точка) го дели дрвото (и поддрвата) на две половини, користејќи една од к-те димензиите (или карактеристиките) како основа за поделбата. Овие димензии можат да претставуваат различни карактеристики на податоците. Во контекст на нашиот алгоритам за креирање на дигитални мозаици, к-те димензиите ги претставуваат каналите на бојата на плочките (BluRedGren, BGR), па токму затоа к = 3.

Примарната придобивка од користењето на КД-дрво е тоа што ни овозможува ефикасно пребарување на најсличната плочка за даден регион од влезната слика. Наместо да го споредуваме регион од влезната слика со сите сите плочки од множеството со плочки, КД-дрвото ќе го намали бројот на споредби со фокусирање само на „важната обласат од 3 димензионалниот простор“, т.е. со фокусирање само на еден од трите канали, B, G или R.

3.2.2 Конструирање на КД-дрво

Во нашиот случај, КД-дрвото ќе го изградиме со вредностите за боја на плочките од множеството со плочки . Притоа за да го изградиме КД-дрво, треба да ги следиме

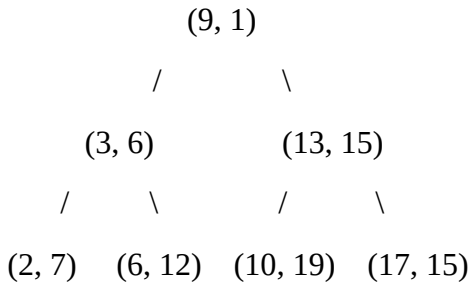
следните чекори:

- Избор на димензија:
  На секое ниво од дрвото, најпрво треба да избереме канал (димензија) врз основа на којшто ќе ги поделиме останатите бои на плочки. Вообичаено, каналот се избира со користење на round-robin.
- Подредување на точките и избор на медијана:
  Откако ќе избереме канал, треба да ги сортираме боите на плочките во растечки редослед според вредноста на избраниот канал. Од така сортираните бои на плочките ја избираме медијаната (бојата што се наоѓа на средина од сортираните бои) и таа боја станува корен на моменталното ниво. Сите бои коишто имаат вредност за избраниот канал помала од вредноста на бојата на медијана одат лево од неа, а сите поголеми одат десно од неа, обезбедувајќи балансираност на дрвото.
- Рекурзивна поделба:
  Претходните два чекори рекурзивно се применуваат и на боите од левото и на боите од десното поддрво со цел конструирање на целосното КД-дрво.

Да го разгледаме следниот пример на конструирање на 2-димензионално дрво. Ги имаме следните 2-димензионални точки:

{(3,6),(17,15),(13,15),(6,12),(9,1),(2,7),(10,19)}

Доколку го примениме претходно опишаниот алгоритам за конструирање на к-димензионално дрво, ќе го добиеме следното 2-димензионално дрво:



3.2.3 Пронаоѓање на најблизок сосед во КД-дрво

Алгоритамот за наоѓање на најблизок сосед во КД-дрво е сличен на оној на бинарно пребарување. Разликата произлегува во тоа што на секое ниво од дрвото ја менуваме димензијата врз основа на којашто ќе го вршиме гранењето. Исто така, како резултат на ова менување на димензии доаѓа до нарушување на карактеристиката на бинарно пребарувачко дрво – лево од даден јазол да се наоѓаат сите вредности помали од вредноста на тој јазол, а десно од него сите вредности поголеми од неговата вредност. Пример да ги погледнеме точките (10, 19), (17, 15) од претходното дрво. На нивото над овие точки, поделбата се врши врз основа на x-координатата и точно (10, 19) се наоѓа лево од (13, 15), а (17, 15) десно од (13, 15). Но, доколку ги погледнеме нивните y-координати, ќе забележиме дека ова правило не важи - (10, 19) е лево од (13, 15), иако 19 > 15. И покрај овој недостаток на КД-дрвата, сепак е можно точно пронаоѓање

на најблискиот сосед на дадена точка, така што на патот наназад ќе ги провериме и оние области (гранки) од КД-дрвото коишто можат да го содржат најблискиот сосед. Ова сепак не значи дека постои шанса да се изминат сите јазли на КД-дрвото и со тоа да добиеме линеарно пребарување. Во најлош случај, доколку КД-дрвото има висина H, ќе ни требаат максимално 2H споредби за да го најдеме најблискиот сосед. Доказот на ова тврдење е надвор од границите и целите на оваа документација.

Како подобро би го разбрале алгоритамот за пронаоѓање на најблизок сосед (или во нашиот конкретен случај пронаоѓање на најблиска плочка на даден регион од влезната слика) со употреба на КД-дрво, ќе разгледаме еден пример засновaт на претходно конструираното КД-дрво.

Пребарување на најблизок сосед за точката (10, 10)

1. **Започни од коренот: (9, 1)**
   - Спореди ја целната точка (10, 10) со точката (9, 1) по x-оската (димензија 0).
     Бидејќи (10 > 9), се движиме кон десното поддрво: (13, 15).
2. **Спореди со (13, 15)**
   - Спореди ја точката (10, 10) со (13, 15) по y-оската (димензија 1). Бидејќи (10 < 15), се движиме кон лево поддрво: (10, 19).
3. **Спореди со (10, 19)**
   - Спореди ја точката (10, 10) со (10, 19) по x-оската (димензија 0). Бидејќи (10 == 10), се движиме кон лево поддрво (кое е празно).
   - Моменталното најдобро растојание е до (10, 19) и изнесува $\sqrt{(10-10)^2 + (10-19)^2}$=9.
4. **Врати се назад и провери ги другите поддрва**
   - Врати се назад до (13, 15). Провери дали треба да го пребаруваме другото поддрво:
   - Растојанието од (10, 10) до рамнината на поделба (y = 15) е $|10 - 15| = 5$.
   - Бидејќи 5 < 9, пребаруваме во десното поддрво: (17, 15).
5. **Спореди со (17, 15)**
   - Евклидовото растојание до (10, 10) е $\sqrt{(17-10)^2 + (15-10)^2}\approx$ 8.6, што е помало од 9.
   - Ажурирај го најблискиот сосед да е (17, 15) со растојание од приближно 8.6.
6. **Врати се назад до (9, 1) и провери друго поддрво**
   - Хоризонталното растојание од (10, 10) до x-оската на (9, 1) е **|10 - 9| = 1**, што е помало од моменталното најдобро растојание (8.6), па затоа проверувај го левото поддрво на (9, 1)
7. **Премини кон левото дете (3, 6)**
   - Евклидовото растојание до (10, 10) е $\sqrt{(17-3)^2 + (10-6)^2}\approx$ 8.1
   - Ажурирај го најблискиот сосед да е (3, 6) со растојание од приближно 8.1.
8. **Провери ги левото и десното дете на (3, 6)**
   - Растојанијата од (10, 10) до (2, 7) и (6, 12) се поголеми од 8.1, па нема потреба од понатамошно ажурирање.
9. **Резултат**

- Најблискиот сосед на точката (10, 10) во даденото КД-дрво е (3, 6) со растојание од приближно 8.1.

На аналоген начин ќе се формира и пребарува и КД-дрвото од боите на плочките во рамките на нашиот алгоритам. Единствената разлика е тоа што ќе се употреби 3-димензионално дрво, место 2-димензионално.

3.2.4 Имплементација на КД-дрво во рамките на алгоритамот за креирање на дигитален мозаик

За да го имплементираме пребарувањето на најблиска плочка на даден регион од влезната слика со употреба на КД-дрва, ќе го употребиме модулот spatial од библиотеката scipy којшто ни нуди готова имплементација на КД-дрва. За да изградиме КД-дрво ќе го повикаме конструкторот на класата KDTree со еден аргумент: листа од боите на плочките врз основа на коишто ќе го изградиме КД-дрвото. Инстанцата од оваа класа ќе ја сместиме во променлива на ниво на класа, self.__kd_tree.

За да ја најдеме, пак, најблиската плочка на даден регион од влезната слика, ќе го повикаме методот query од KDTree инстанцата. Овој метод ќе ни го врати индексот на којшто се наоѓа најблиската плочка во листата што ја искористивме за креирање на објектот self.__kd_tree. Овој индекс лесно можеме да го искористиме, во комбинација со двата речници self.__tiles и self.__index_table за да ја добиеме самата плочка. За таа цел ќе го имплементираме методот __find_best_match_for_region_improved којашто ќе прими еден параметар: region – регионот од влезната слика за којшто треба да ја најдеме најблиската плочка, а ќе ја врати најблиската плочка.

Подобрената верзија на алгоритамот со употреба на КД-дрва можеме да ја согледаме преку изменетиот метод create_mosaic и методот __find_best_match_for_region_improved:

```
def create_mosaic(self, image_filepath, tiles_size=__DEFAULT_TILE_SIZE, fill_option=KEEP_SAME_SIZE,
                  randomization_improvement=False,
                  randomization_value=__DEFAULT_RANDOMIZATION_VALUE,
                  improved=False
                  ):
    if tiles_size[0] > 128 or tiles_size[1] > 128:
        raise Exception("Tiles size cannot be greater than 128")

    self.__tiles = self.__populate_and_get_tiles_database(self.tiles_path, tiles_size)
    self.__index_table = self.__populate_and_get_average_database(self.__tiles, tiles_size)

    if randomization_value > len(self.__index_table.keys()) or randomization_value < 0:
        randomization_improvement = False

    if randomization_improvement:
        self.__save_index_table = self.__populate_and_get_average_database(self.__tiles, tiles_size)

    if improved:
        self.__kd_tree = KDTree(list(self.__index_table.keys()))

    print("Creating a mosaic...")
    image = cv2.imread(image_filepath, 1)

    start_time = time.time()
    image_regions = self.__get_regions(image, tiles_size)

    mosaic_shape = self.get_mosaic_size(image.shape, tiles_size, fill_option)
    mosaic = np.zeros(mosaic_shape, dtype=np.uint8)
    mosaic_height, mosaic_width, depth = mosaic_shape
    tile_height, tile_width = tiles_size
```

```
i = 0
for y in range(0, mosaic_height, tile_height):
    for x in range(0, mosaic_width, tile_width):

        if randomization_value == 0:
            break

        if fill_option == MosaicProcessor.KEEP_SAME_SIZE:
            if image_regions[i].shape[1] != tile_width or image_regions[i].shape[0] != tile_height:
                i += 1
                continue

        if randomization_improvement:
            random_index_keys = random.sample(list(self.__save_index_table.keys()), randomization_value)
            self.__index_table = {key: self.__save_index_table[key] for key in random_index_keys}

        if randomization_improvement and improved:
            self.__kd_tree = KDTree(list(self.__index_table.keys()))

        if improved:
            best_tile = self.find_best_match_for_region_improved(image_regions[i])
        else:
            best_tile = self.find_best_match_for_region_base(image_regions[i])

        mosaic[y:y + tile_height, x:x + tile_width] = best_tile
        i += 1

self.last_execution_time = time.time() - start_time

return mosaic
```

```
def find_best_match_for_region_improved(self, region):  3 usages (2 dynamic)
    region_average = self.image_processor.average_color(region)
    distance, i = self.__kd_tree.query(region_average)
    tile_name = list(self.__index_table.values())[i]
    closest_tile = self.__tiles[tile_name]
    return closest_tile
```

Времињата на завршување на алгоритамот со подобрување со КД-дрва, при користење на плочки со големина од 32x32, 16x16 и 8x8 пиксели изнесуваат: 0.30 секунди, 1.02 секунди и 3.38 секунди, соодветно. Како што може да се забележи, имаме подобрување на времињата на завршување на алгоритамот и тоа: од 0.42 на 0.30 секунди, од 1.47 на 1.02 секунди и од 5.31 на 3.38 секунди.

Овие временски подобрувања не се запрепастувачки, но не смее да се заборави фактот дека при подобрувањето на алгоритамот со употреба на КД-дрва, овозможивме користење на целосното множество со плочки при споредбата за секој регион. Како резултат на тоа крајно добиените мозаици се со значително подобар квалитет на детали и со поголема естетска убавина. Разликата во деталите на мозаикот добиен од алгоритам со подобрување со рандомизација (randomization_value = 90) и мозаикот добиен од алгоритам со подобрување со употреба на КД-дрва, при употреба на иста големина на регион од 16x16 пиксели, можеме да ги видиме на Слика 6.

Слика 6 Мозаик добиен од алгоритам со подобрување со употреба на КД-дрва (горе) и мозаик добиен од алгоритам со подобрување со рандомизација, randomization_value = 90 (долу)

3.3Подобрување на алгоритамот со употреба паралелизам (multithreading)

Последното подобрување на предложениот алгоритам за креирање на дигитални мозаици можеме да го направиме со воведувањето на паралелизам (multithreading). Имено, зошто секој следен регион да го чека претходниот, во процесот на пронаоѓање на најблиска плочка, кога овие операции се независни една од друга и може да се одвиваат паралелно. Па затоа, токму тука ќе го воведеме паралелизмот во рамките на алгоритамот.

Паралелизмот, како парадигма во компјутерските науки е, така да речеме, критична и често стравувана област. Тоа е така бидејќи човековиот мозок не е интуитивно приспособен да ги открива проблеми коишто можат да се појават при воведувањето на паралелизам во алгоритмите, како што се проблемите со синхронизација, критични области и состојби на трка.

За справување со проблемите на синхронизација ќе ја искористиме помошта од модулот concurrent, а ние само ќе креираме максимален број на нишки (thread-ови) и ќе ја повикаме функцијата map, којашто прима два параметри: iterables – листа којашто ќе се итерира и map_function – мапирачка функција којашто ќе се примени на секој елемент од листата iterables. Функцијата ќе врати совршено синхронизирана листа, според исиот редослед како и елементите од влезната листа iterables.

Во нашиот случај како аргумент за параметарот iterables ќе ја испратиме листата со региони, а во зависност од тоа дали ќе ги користиме опциите на алгоритамот за подобрување со рандомизација или пак подобрување со КД-дрва, или пак двете, или пак ниту една, како аргумент за мапирачката функција ќе го испратиме соодветниот find_best_match_for_region методот. Во овој случај, map функцијата ќе ни ги врати најблиските плочки за секој од регионите на влезната слика, подредени по правилен редослед, исто како што биле подредени и регионите во влезната листа.

Единствени критични региони во рамките на нашиот алгоритам се self.__index_table и self.__tiles речниците и инстанцата од KDTree, self.__kd_tree. Меѓутоа од сите овие објекти во рамките на методите __find_best_match_for_region__base и __find_best_match_for_region__improved се врши само операција на читање, така што не постојат ризици поврзани со состојба на трка. Со синхронизацијата ќе се справи, како што кажавме, модулот concurrent.

Единствените измени коишто остануваат да се направат се во методот create_mosaic, со цел да овозможиме функционирање на алгоритамот истовремено со којашто сакаме опција за подобрување: рандомизација, КД-дрва и/или паралелизација. Тоа ќе го постигнеме со неколку параметри од типот bool коишто ќе ја имаат улогата на опции на методот create_mosaic. Исто така, ќе направиме и поделба на два дополнителни методи: __create_mosaic_sequential и __create_mosaic_parallel , со цел поголема прегледност, разбирливост и одржливост на нашиот код.

За крај остануваат да го спомнеме помошниот методите __prepare_index_randomization_base којшто врши подготовка на речникот self.__index_table во случаеви кога се користи алгоритамот со опциите за подобрувањето со рандомизација и паралелизација, и методот __prepare_index_randomization_improved којшто врши подготовка на речникот self.__index_table во случаеви кога се користи алгоритамот со сите три опции за подобрување: рандомизација, КД-дрва и паралелизација.

Конечната имплементација на create_mosaic и останатите претходно споменати методи е следна:

```
def find_best_match_for_region_improved_parallel(self, region, kd_tree, index_table):  1 usage  new *
    region_average = self.image_processor.average_color(region)
    distance, i = kd_tree.query(region_average)
    tile_name = list(index_table.values())[i]
    closest_tile = self.__tiles[tile_name]
    return closest_tile

def __prepare_index_randomization_improved(self, image_region):  1 usage  new *
    random_index_keys = random.sample(list(self.__save_index_table.keys()), self.__save_randomization_value)
    index_table = {key: self.__save_index_table[key] for key in random_index_keys}
    kd_tree = KDTree(list(index_table.keys()))
    return self.find_best_match_for_region_improved_parallel(image_region, kd_tree, index_table)

def __prepare_index_randomization_base(self, image_region):  1 usage  new *
    random_index_keys = random.sample(list(self.__save_index_table.keys()), self.__save_randomization_value)
    self.__index_table = {key: self.__save_index_table[key] for key in random_index_keys}
    return self.find_best_match_for_region_base(image_region)
```

```python
def __create_mosaic_parallel(self, mosaic_shape, tiles_size, image_regions, fill_option, improved,  2 usages (1 dynamic)  new
                             randomization_improvement, randomization_value, max_workers):
    mosaic = np.zeros(mosaic_shape, dtype=np.uint8)
    mosaic_height, mosaic_width, depth = mosaic_shape
    tile_height, tile_width = tiles_size

    if improved:
        if randomization_improvement and randomization_value > 0:
            self.__save_randomization_value = randomization_value
            with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
                matched_tiles = list(executor.map(self.__prepare_index_randomization_improved,  *iterables: image_regions))
        elif not randomization_improvement:
            with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
                matched_tiles = list(executor.map(self.find_best_match_for_region_improved,  *iterables: image_regions))

    else:
        if randomization_improvement and randomization_value > 0:
            self.__save_randomization_value = randomization_value
            with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
                matched_tiles = list(executor.map(self.__prepare_index_randomization_base,  *iterables: image_regions))
        elif not randomization_improvement:
            with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
                matched_tiles = list(executor.map(self.find_best_match_for_region_base,  *iterables: image_regions))

    i = 0
    for y in range(0, mosaic_height, tile_height):
        for x in range(0, mosaic_width, tile_width):
            if randomization_value == 0:
                break

            if fill_option == MosaicProcessor.KEEP_SAME_SIZE:
                if image_regions[i].shape[1] != tile_width or image_regions[i].shape[0] != tile_height:
                    i += 1
                    continue

            mosaic[y:y + tile_height, x:x + tile_width] = matched_tiles[i]
            i += 1

    return mosaic
```

```python
def __create_mosaic_sequential(self, mosaic_shape, tiles_size, image_regions, fill_option, improved,  2 usages
                               randomization_improvement, randomization_value, best_match_region_times):
    mosaic = np.zeros(mosaic_shape, dtype=np.uint8)
    mosaic_height, mosaic_width, depth = mosaic_shape
    tile_height, tile_width = tiles_size

    i = 0
    for y in range(0, mosaic_height, tile_height):
        for x in range(0, mosaic_width, tile_width):

            if randomization_value == 0:
                break

            if fill_option == MosaicProcessor.KEEP_SAME_SIZE:
                if image_regions[i].shape[1] != tile_width or image_regions[i].shape[0] != tile_height:
                    i += 1
                    continue

            per_region_start_time = time.time()

            if randomization_improvement:
                random_index_keys = random.sample(list(self.__save_index_table.keys()), randomization_value)
                self.__index_table = {key: self.__save_index_table[key] for key in random_index_keys}
                self.__kd_tree = KDTree(list(self.__index_table.keys()))

            if improved:
                best_tile = self.find_best_match_for_region_improved(image_regions[i])
            else:
                best_tile = self.find_best_match_for_region_base(image_regions[i])

            best_match_region_times.append(time.time() - per_region_start_time)

            mosaic[y:y + tile_height, x:x + tile_width] = best_tile
            i += 1

    return mosaic
```

```python
def create_mosaic(self, image_filepath, tiles_size=__DEFAULT_TILE_SIZE, fill_option=KEEP_SAME_SIZE,  1 usage  new *
                  randomization_improvement=False,
                  randomization_value=__DEFAULT_RANDOMIZATION_VALUE,
                  improved=False,
                  parallel_processing=False,
                  max_workers=__MAX_WORKERS,
                  ):

    if tiles_size[0] > 128 or tiles_size[1] > 128:
        raise Exception("Tile size cannot be greater than 128")

    self.__tiles = self.__populate_and_get_tiles_database(self.tiles_path, tiles_size)
    self.__index_table = self.__populate_and_get_average_database(self.__tiles, tiles_size)

    if randomization_value > len(self.__index_table.keys()) or randomization_value < 0:
        randomization_improvement = False

    if randomization_improvement:
        self.__save_index_table = self.__populate_and_get_average_database(self.__tiles, tiles_size)

    print("Creating a mosaic...")
    image = cv2.imread(image_filepath, 1)

    start_time = time.time()
    image_regions = self.__get_regions(image, tiles_size)

    if improved:
        self.__kd_tree = KDTree(list(self.__index_table.keys()))

    mosaic_shape = self.get_mosaic_size(image.shape, tiles_size, fill_option)

    best_match_region_times = []

    if parallel_processing:
        mosaic = self.__create_mosaic_parallel(mosaic_shape, tiles_size, image_regions, fill_option, improved,
                                               randomization_improvement, randomization_value, max_workers)

    else:
        mosaic = self.__create_mosaic_sequential(mosaic_shape, tiles_size, image_regions, fill_option, improved,
                                                 randomization_improvement, randomization_value,
                                                 best_match_region_times)

    self.last_execution_time = time.time() - start_time

    return mosaic
```

За крај на овој дел, ќе се осврнеме на „подобрувањето“ постигнато со воведувањето на опцијата за паралелизација во рамките на алгоритамот за креирање на дигитални мозаици. Доколку ги претпоставиме времињата на извршување на алгоритамот со опцијата за подобрување со КД-дрва и истовремена паралелизација, би очекувале значително подобрување на времето на извршување. Меѓутоа резултатите се различни од очекуваното. Времињата на извршување на алгоритамот со опциите за подобрување со КД-дрва и истовремена паралелизација, при користење на плочки со големини од 32x32, 16x16 и 8x8 пиксели изнесуваат: 0.66 секунди, 2.22 секунди и 6.43 секунди. Споредбено со времињата на извршување на алгоритамот кога се користи опцијата за подобрување само со КД-дрва (0.30 секунди, 1.02 секунди и 3.38 секунди) имаме забележителни влошувања.

Овие резултати се невообичаени, но не неочекувани. Имено, причината за ова влошување се должи на еден механизам којшто се нарекува GIL. GIL, односно The Global Intrepreter Lock е тип на mutex брава којашто се користи од страна на Cpython интрепретерот за да се осигура дека точно само една нишка може да извршува Python bytecode во исто време, дури и во програми со повеќе нишки. Ваквата функција на GIL – от овозможува поедноставување на менаџирањето со меморија, или подетално со бројачот на референци (користен од страна на garbage collectorot на Python) којшто не е thread-safe.

Сега, знаејќи го ова и знаејќи дека и покрај тоа што нашиот алгоритам навистина користи повеќе нишки во позадини, но само една нишка, во еден момент е активна и знаејќи дека при промена на нишка има соодветен overhead добиен од соодветната промена на контекст на нишка, причината за добиените времиња на извршување на алгоритамот со опциите за подобрување со КД-дрва и истовремена паралелизација, станува јасна.

Еден начин да го заобиколиме GIL – от во процесот на паралелизација на предложениот алгоритам е да употребиме паралелизација на ниво на процеси, место на ниво на нишки. Имено, место во процесот на пронаоѓање на најблиска плочка на даден регион од влезната слика, да употребуваме нишки, со овој поразличен начин на паралелизација, би употребиле процеси. Односно, ќе креираме нов процес за секој регион од влезната слика. Тоа лесно можеме да го постигнеме на ниво на код. Сè што треба да направиме е да ја промениме класата од ThreadPoolExecutor во ProcessPoolExecutor.

Меѓутоа времињата коишто ги добиваме со овој пристап се драстично влошени. Времето на извршување на алгоритамот за креирање дигитални мозаици со опцијата за подобрување со употреба на КД-дрва и паралелизација (на ниво на процес) и големина на плочка од 32x32 пиксели изнесува: 327.55 секунди. Тоа е влошување за повеќе од 490 пати. Ова е само доказ дека overhead – от којшто е присутен за креирањето на процеси и нивната промена на контекст, е далеку поголем од бенефитот што се добива со воведување на паралелизам на ниво на процес. Токму затоа оваа паралелизација на ниво на процес ќе ја отстраниме од предложениот алгоритам.

Единствениот друг начин да се заобиколи GIL – от во процесот на паралелизација е да се употребат некои други интрепретери, како што се Jython или IronPython. Но, и со овој начин имаме голем број на ограничувања, бидејќи голем дел библиотеките коишто ги употребивме во рамките на предложениот алгоритам, немаат поддршка токму за овие интрепретери.

4. Експерименти и анализи

На почетокот од овој дел, ќе направиме една крајна промена на предложениот алгоритам за креирање на дигитални мозаици. Во рамките на методот create_mosaic ќе додадеме опција за запишување на лог и опција за автоматско зачувување на креираниот мозаик на диск. Овие дополнувања ќе ни овозможат полесно да ги спроведеме експериментите и анализите од овој дел. Овие промените се видливи во крајниот и целосен код на предложениот алгоритам.

Логот којшто алгоритамот го формира е детален и ги содржи следните информации:

- NAME – името на влезната фотографија
- SIZE – големина на влезната фотографија изразена во пиксели
- T_SIZE – големината на плочката/регионот изразена во пиксели
- REG – вкупниот број на региони добиени од влезната слика
- REG_AVG – просечното време за пронаоѓање на најблиска плочка за еден регион (вредност 0 доколку е употребена паралелизација)

- ALG – дали е употребено подобрување со КД-дрва (име вредност Improved доколку е, инаку вредност Base)
- RAND – дали е употребено подобрување со рандомизација (вредност True или False)
- RAND_V – вредноста на randomization_value (-1 доколку не е употребено подобрување со рандомизација)
- PARA – дали е употребено подобрување со паралелизација на ниво на нишка (вредност True или False)
- THREADS – бројот на употребени нишки (-1 доколку не е употребено подобрување со паралелизација)
- EXEC_TIME – вкупното време на извршување на алгоритамот

4.1Најкратко време на извршување

Првиот експеримент којшто ќе го спроведеме е да утврдиме која комбинација на подобрување овозможува најкратко време на извршување на алгоритамот. За таа цел ќе ја употребиме фотографијата bear.jpg (која се наоѓа во директориумот test_images/small/ во рамките на проектот) и големина на регион од 16x16 пиксели.

Логот изгледа вака:

```
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0 s  ALG: Improved  RAND: True  RAND_V: 60  PARA: True  THREADS: 12  EXEC_TIME: 0.3506948947906494 s
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0.00011726251 s  ALG: Improved  RAND: True  RAND_V: 60  PARA: False  THREADS: -1  EXEC_TIME: 0.25411638941955566 s
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0 s  ALG: Improved  RAND: False  RAND_V: -1  PARA: True  THREADS: 12  EXEC_TIME: 0.18764877319335938 s
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0.0000390896 s  ALG: Improved  RAND: False  RAND_V: -1  PARA: False  THREADS: -1  EXEC_TIME: 0.0887155532836914 s
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0 s  ALG: Base  RAND: True  RAND_V: 60  PARA: True  THREADS: 12  EXEC_TIME: 0.25811290740966797 s
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0.0001246200 s  ALG: Base  RAND: True  RAND_V: 60  PARA: False  THREADS: -1  EXEC_TIME: 0.27268886566162211 s
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0 s  ALG: Base  RAND: False  RAND_V: -1  PARA: True  THREADS: 12  EXEC_TIME: 2.586602210998535 s
NAME: bear.jpg  SIZE: 740x740  T_SIZE: 16x16  REG: 2209  REG_AVG: 0.0012244753 s  ALG: Base  RAND: False  RAND_V: -1  PARA: False  THREADS: -1  EXEC_TIME: 2.6035463809967041 s
```

Како што може да се забележи, најкратко време на извршување се постигнува кога се користи опцијата за подобрување само со употреба на КД-дрва. Овој резултат е сосема очекуван поради фактот што овој пристап има најмал overhead (потребно е само еднаш да се изгради КД-дрвото, а потоа само да се изминува) и најдобра временска сложеност од O(Nlog2M), каде што N е бројот на региони, а M бројот на плочки.

За да се увериме во претходниот заклучок експериментот ќе го спроведеме и на сликата girl.jpg со големина на плочка од 32x32 пиксели.

Логот изгледа вака:

```
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0 s  ALG: Improved  RAND: True  RAND_V: 60  PARA: True  THREADS: 12  EXEC_TIME: 4.982667446136475 s
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0.0000950348 s  ALG: Improved  RAND: True  RAND_V: 60  PARA: False  THREADS: -1  EXEC_TIME: 2.7492923736572266 s
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0 s  ALG: Improved  RAND: False  RAND_V: -1  PARA: True  THREADS: 12  EXEC_TIME: 2.483238935470581 s
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0.0000368005 s  ALG: Improved  RAND: False  RAND_V: -1  PARA: False  THREADS: -1  EXEC_TIME: 1.087876796722412 s
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0 s  ALG: Base  RAND: True  RAND_V: 60  PARA: True  THREADS: 12  EXEC_TIME: 2.9903135299682617 s
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0.0000798665 s  ALG: Base  RAND: True  RAND_V: 60  PARA: False  THREADS: -1  EXEC_TIME: 2.3156015872955322 s
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0 s  ALG: Base  RAND: False  RAND_V: -1  PARA: True  THREADS: 12  EXEC_TIME: 38.711421966552734 s
NAME: girl.jpg  SIZE: 2030x3608  T_SIZE: 16x16  REG: 28702  REG_AVG: 0.0013066237 s  ALG: Base  RAND: False  RAND_V: -1  PARA: False  THREADS: -1  EXEC_TIME: 37.15798616409302 s
```

Повторно, најкратко време на извршување се постигнува кога се користи опцијата за подобрување само со употреба на КД-дрва.

4.2Влијанието на големината на регионот и големината на влезната слика врз времето на извршување на алгоритамот

Во овој експеримент, ќе воочиме како големината на регионот и големината на влезната слика влијаат на времето на извршување на алгоритамот. За таа цел ќе ја употребиме фотографијата underground.jpg (која се наоѓа во директориумот /test_images/large/ во рамките на проектот) со големина од 7912x5275 пиксели. Ќе го

употребиме алгоритамот со опцијата за подобрување со употреба на КД-дрва. Големините на регионите за којшто ќе го спроведеме експериментот се: 8x8, 16x16, 32x32 и 64x64 пиксели.

Логот изгледа вака:

```
NAME: underground.jpg   SIZE: 5275x7912  T_SIZE: 8x8     REG: 652740   REG_AVG: 0.0000309801 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 21.851035404205322 s
NAME: underground.jpg   SIZE: 5275x7912  T_SIZE: 16x16   REG: 163350   REG_AVG: 0.0000369796 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 6.320996284484863 s
NAME: underground.jpg   SIZE: 5275x7912  T_SIZE: 32x32   REG: 40920    REG_AVG: 0.0000497487 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 2.1160964965820312 s
NAME: underground.jpg   SIZE: 5275x7912  T_SIZE: 64x64   REG: 10292    REG_AVG: 0.0000937475 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 1.0194780826568604 s
```

Очекувано, времето на извршување на алгоритамот се зголемува со намалувањето на големината на регионот. Сега истиот експеримент ќе го спроведеме на некоја многу помала фотографија, пример како што е фотографијата bear.jpg.

Логот изгледа вака:

```
NAME: bear.jpg          SIZE: 740x740    T_SIZE: 8x8     REG: 8649     REG_AVG: 0.0000306904 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 0.2717878818511963 s
NAME: bear.jpg          SIZE: 740x740    T_SIZE: 16x16   REG: 2209     REG_AVG: 0.0000320051 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 0.07372260093688965 s
NAME: bear.jpg          SIZE: 740x740    T_SIZE: 32x32   REG: 576      REG_AVG: 0.0000415421 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 0.024005174636840820 s
NAME: bear.jpg          SIZE: 740x740    T_SIZE: 64x64   REG: 144      REG_AVG: 0.0000743570 s   ALG: Improved   RAND: False   RAND_V: -1   PARA: False   THREADS: -1   EXEC_TIME: 0.010995388031005860 s
```

Повторно очекувано, времињата на извршување на алгоритамот се зголемија со намалувањето на големината на регионот. Меѓутоа времињата на извршување на алгоритамот при влезната слика bear.jpg се драстично помали во споредба со времињата на извршување на алгоритамот при влезната слика underground.jpg. Ова е директен пример и доказ на тврдењето коешто го дадовме во 2.3.3.

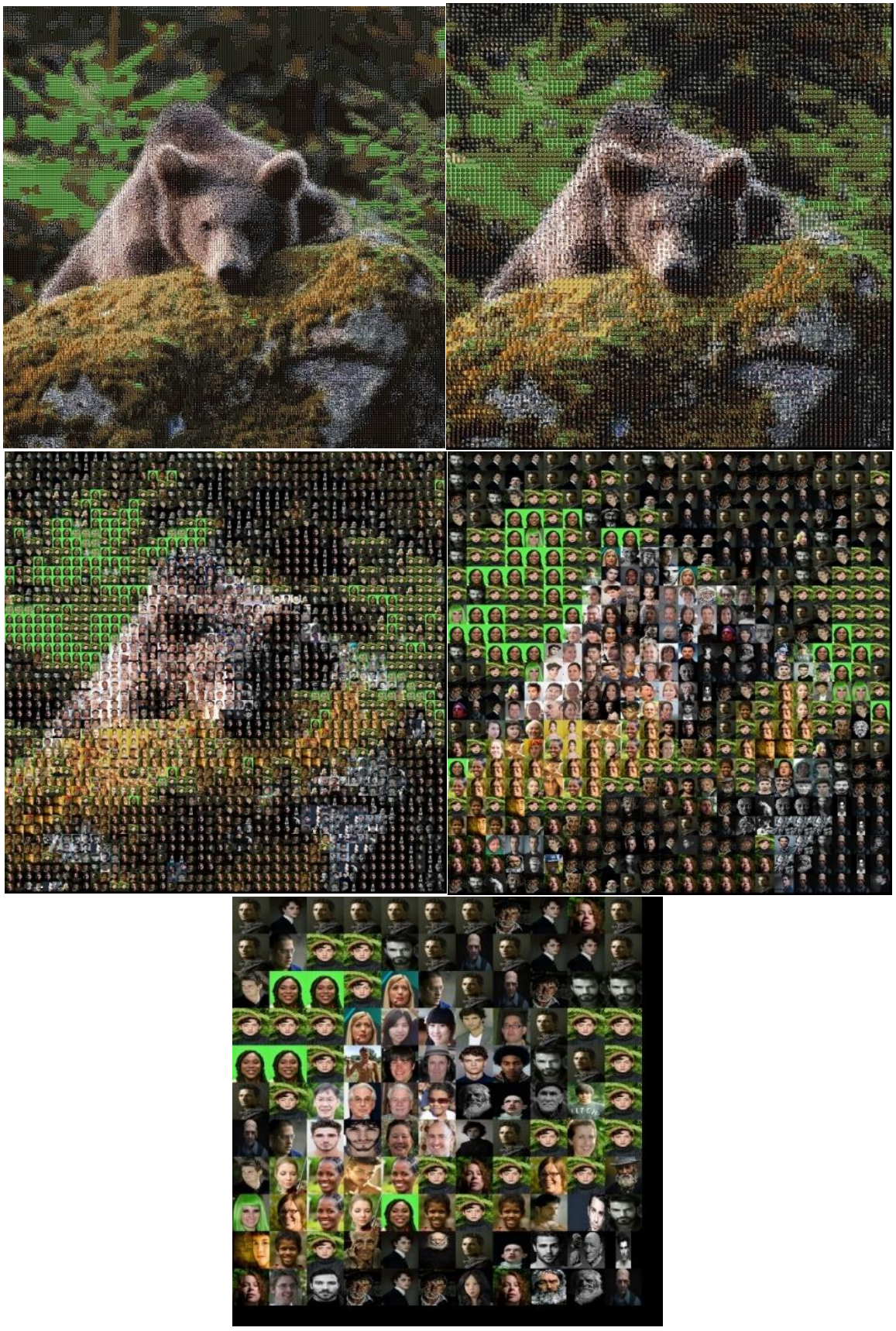
4.3Влијанието на големината на регионот врз квалитет на добиениот мозаик

Последниот експеримент којшто ќе го извршиме во рамките на оваа документација се однесува на естетскиот квалитет на добиениот мозаик. Метриката „естетски квалитет“ е од субјективен карактер и за неа не можеме да кажеме дека е потврдена со научни докази. Целта на овој експеримент е да ги согледаме разликите во добиените мозаици при користење на региони и фотографии со различни големини.

За оваа цел ќе употребиме две фотографии: фотографијата girl.jpg со големина од 3608x2030 пиксели и фотографијата bear.jpg со големина од 720x720 пиксели. За двете фотографии ќе креираме мозаици со големини на регион од 4x4, 8x8, 16x16, 32x32 и 64x64 пиксели. Ќе го употребиме алгоритамот со опцијата за подобрување со КД-дрва бидејќи дава најбрзи и најдобри резултати.

За овој експеримент нема да ни биде потребен логот којшто го формира алгоритамот, ами самите добиени мозаици. Резултатите можеме да ги видиме на Слика 7 и Слика 8.

Очекувано, големината на самата влезна слика директно ја условува големина на плочката којашто ќе ја користиме, со што го потврдиме тврдењето од 2.3.4.



Слика 7 Мозаици на сликата bear.jpg со големина на плочки од 4x4, 8x8, 16x16, 32x32 и 64x64

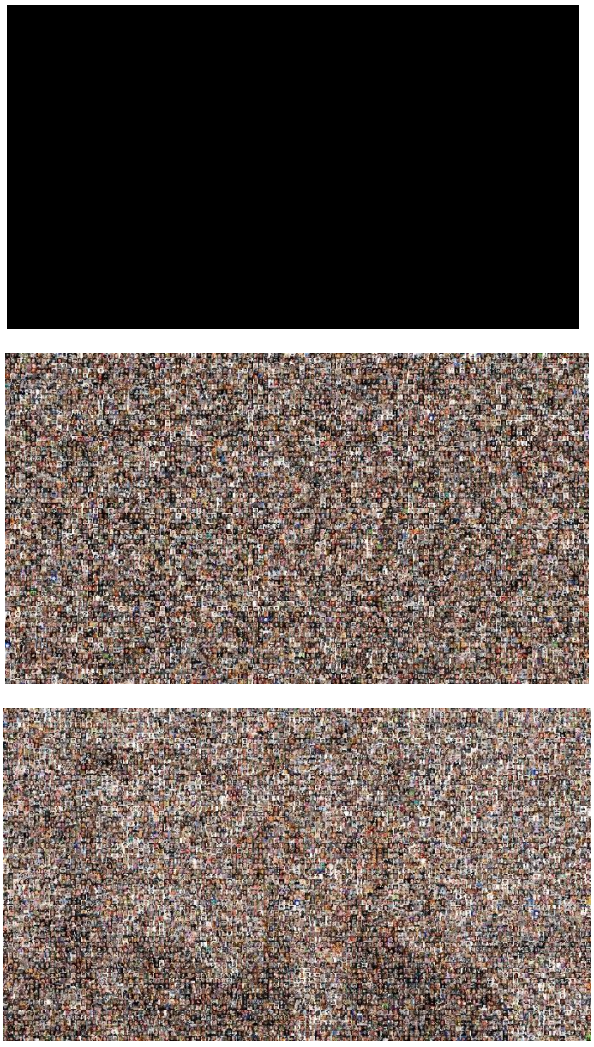
Слика 8 Мозаици на сликата girl.jpg со големина на плочки од 4x4, 8x8, 16x16, 32x32 и 64x64

4.4Неколку експерименти од забавен карактер

За сосема самиот крај, ќе направиме еден експерименти од информативен и забавен карактер.

Да погледнеме што се добива кога го употребуваме алгоритамот за креирање на дигитални мозаици со опцијата за подобрување со рандомизација со randomization_value =0, randomization_value = 1 и randomization_value = 2. Резултатите се видливи на Слика 9.

Очекувано, при randomization_value = 0 добиваме „празен“ мозаик бидејќи алгоритамот нема на располагање плочки од коишто може да избира. При randomization_value = 1 се добива целосен шум бидејќи алгоритамот секогаш ја користи единствената плочка којашто му е на располагање, а е случајно избрана. И за крај, при randomization_value = 2, се добива мозаик со голем шум, којшто наликува на некаков „gray-scale“ мозаик, бидејќи при секој избор имаа само две случајно избрани плочки, па изборот се сведува на подобрата од нив.



Слика 9 Мозаици на фотографијата girl.jpg при користење на алгоритам со опцијата за подобрување со рандомизација со randomization_value =0 (горе), randomization_value = 1 (средина) и randomization_value = 2 (долу)

5. Заклучок

Во рамките на овој проект беше опфатен развојот на алгоритамот за креирање на дигитални мозаици од влезни фотографии. Преку истражувањето и имплементацијата на различните опции, како што се: опцијата за употреба на рандомизација, опцијата за употреба на КД-дрва и опцијата за паралелизација со употреба на нишки, алгоритамот, исто така, беше успешно оптимизиран и временски и во однос на визуелниот квалитет.

Преку експериментите спроведени на крајот, пак, практично ги потврдивме тврдењата изнесени на почетокот од документација и се уверивме во оптимизациите постигнати со различните опции за оптимизација.

За крај може да се каже дека проектот во себе сè уште има потенцијал за понатамошни усовршувања и надоградувања. Овие надоградувања вклучуваат: имплементирање на опциите за динамично менување на големината на плочка во случаеви кога тоа е потребно, имплементирање на опција за креирање на мозаик врз основа на текстурата и употреба на различни видови на интрепретери со цел тестирање на подобрувањата постигнати со воведувањето на опцијата за паралелизација на ниво на нишки.

А. Недостатокот од совршено множество на плочки

Да претпоставиме дека имаме влезна фотографија таква што имаме по точно еден региони (со големина од 1x1 пиксел) за секоја можна боја од RGB (BGR) просторот со бои. Во тој случај во влезната фотографија би имале вкупно 255 * 255 * 255 = 16 581 375 региони.

Како би имале совршена имплементација на алгоритамот за пронаоѓање на најблиска плочка на даден регион од влезната слика (нивната оддалеченост секогаш да е 0), во множеството би требало да имаме исто толку многу плочки, односно 16 581 375.

Да претпоставиме дека го користиме алгоритамот за креирање на дигитални мозаици со опцијата за подобрување со употреба на КД-дрва. Вкупниот број на споредби коишто ќе треба алгоритамот да ги направи (во најлош случај) со цел да ги определи најблиските плочки изнесува $16\ 581\ 375 * \log_2(16\ 581\ 375) \approx 3{,}97672 * 10^8 \approx$ 397 672 000 ≈ 400 милиони споредби. Доколку претпоставиме дека времето за една споредба изнесува околу 0.0001 секунда (што е преголема и преоптимистична претпоставка, но доволно добра за примерот), тогаш потребно време за извршување на алгоритамот би било ≈ 40 000 секунди ≈ 11.1 часа.

Б. Различна големина на регион и плочка

Големината на регионите од влезната слика, како што кажавме, не мора да е иста со големина на плочката. Да разгледаме еден таков пример.

Да претпоставиме дека имаме фотографија со големина од 4x2 пиксели. Притоа употребуваме регион со големина од 2x2 пиксели и плочка со големина од 3x2 пиксели. Добиениот мозаик во овој случај би бил со големина од 6x2 пиксели и симболично поедноставен би изгледал вака:

1 1 1 2 2 2
1 1 1 2 2 2

каде што 1 и 2 се имињата на некои плочки. Во овој случај секои 4 пиксели (2x2) од влезната слика (секој регион) се заменети со плочка со плоштина од 6 пиксели (3x2).

Можен, исто така е и обратниот случај кога регион со поголема големина се заменува со плочка со помала големина. Пример да претпоставиме дека имаме влезна фотографија со големина од 6x2 пиксели. Притоа користиме големина на регион со големина од 3x2 пиксели и големина на плочка од 2x2 пиксели. Добиениот мозаик во овој случај би бил со големина од 4x2 пиксели и симболично поедноставен би изгледал вака:

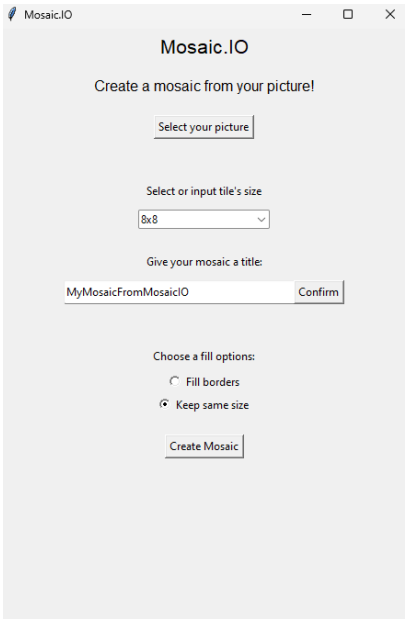
1 1 2 2
1 1 2 2

каде што 1 и 2 се имињата на некои плочки. Во овој случај секои 6 пиксели (3x2) од влезната слика (секој регион) се заменети со плочка со плоштина од 4 пиксели (2x2).

B. Графички кориснички интерфејс

Со цел полесно и поинтуитивно користење на алгоритамот за креирање на дигитални мозаици, во рамките на проектот, исто така, е вклучена и верзија којашто нуди интуитивен кориснички графички интерфејс (GUI). Кодот користен за изработка на интерфејсот нема да се објаснува во рамките на овој проект бидејќи е надвор од неговите цели и граници. Изгледот на графичкиот интерфејс можеме да го видиме на Слика 10.

Целосниот код употребен во рамките на овој проект е достапен на:
https://github.com/Itonkdong/Mosaic-Project



Слика 10 Графичкиот кориснички интерфејс којшто го имплементира алгоритамот за креирање на дигитални мозаици