ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
Кафедра «Електроних обічислюваних машин»

Кваліфікаційна робота
на тему:
«Дослідження та порівняння генетичних алгоритмів з реставрування зображень»

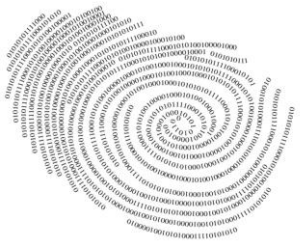
Виконав: ст. гр СПм-20-2 Торба О.О
Керівник: Каргін А.О

Постановка задачі

Мета даного дослідження полягає в об'єднанні генетичного алгоритму, що базується на теорії еволюції Дарвіна, та практичне завдання, а саме отримати реставроване зображення, яке буде схоже на оригінал.

Генетичні алгоритми

Один з найдивовижніших методів вирішення завдань, який запозичив ідею в теорії еволюції Чарльза Дарвіна, заслужено отримав назву "еволюційні обчислення". Найвідомішими представниками цієї родини є генетичні алгоритми. Генетичні алгоритми є одним із найдивовижніших методів вирішення задач пошуку, оптимізації та навчання. Вони можуть призвести до успіху там, де традиційні алгоритми не здатні дати адекватних результатів за прийнятний час.



Переваги та недоліки

- Глобальна оптимізація
- Застосування до складних задвань
- Застосування к задачам, що не мають математичного явлення
- Стійкість до шуму
- Розпаралелювання
- Безперервне навчання

- Спеціальні визначення
- Налаштування гіперпараметрів
- Великий обсяг лічильних операцій
- Передчасна збіжність
- Відсутність гарантованих рішень

Базова структура генетичного алгоритму

- Створення початкової популяції
- Вирахування пристосованості
- Використання операторів відбору, схрещування та мутації
- Перевірка умов зупинення

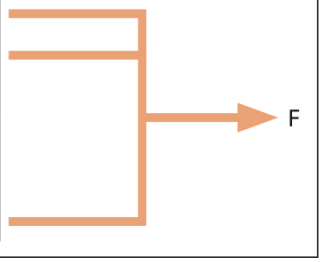
Уявлення рішення

- Хромосова виступає трикутников. Індивідуум – набір таких хромосом, тобто зображення з трикутників.
- Популяція – набір зображень-індивідуумів.
- Функція пристосованості – алгоритм порівняння зображень (MSE, SSIM)

Оператор відбору

- Турнірний оператор відбору

Індивідуум	Пристосованість
A	8
B	12
C	27
D	4
E	45
F	17



Оператор схрещування

- Імітований двійковий кросовер (SBX, або Simulated Binary Crossover)

$$\gamma = \begin{cases} (2u)^{\frac{1}{(\eta+1)}}, & if\ u \le 0.5 \\ \frac{1}{[2(1-u)]^{\frac{1}{(\eta+1)}}}, & otherwie \end{cases}$$

$$c_{1,k} = \frac{1}{2}[(1-\gamma) * p_{1,k} + (1+\gamma) * p_{2,k}]$$

$$c_{2,k} = \frac{1}{2}[(1+\gamma) * p_{1,k} + (1-\gamma) * p_{2,k}]$$

r_i	n_i	R_s	C_s
0.87682	3	0.99069	503.662

Parent-1

r_i	n_i	R_s	C_s
0.878591	2	0.92844	367.57

Child-1

r_i	n_i	R_s	C_s
0.822196	2	0.85161	197.597

Parent-2

r_i	n_i	R_s	C_s
0.820425	3	0.97138	272.59

Child-2

Fig. 15. Simulated Binary Crossover.

Parent

r_i	n_i	R_s	C_s
0.822196	3	0.97221	277.1122

Child

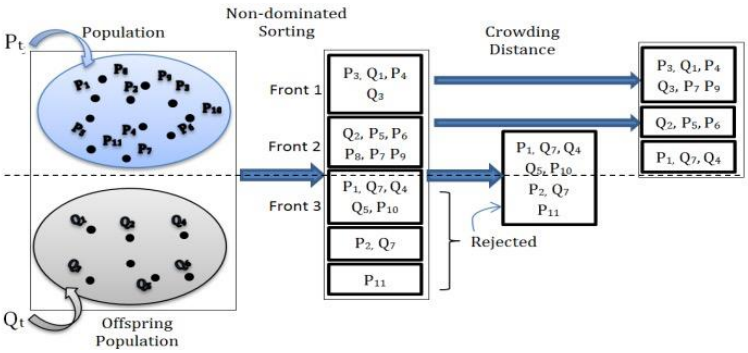
r_i	n_i	R_s	C_s
0.9000000	2	0.95099	500.5009

Оператор мутації

- Оператор поліноміальної мутації

$$c_k = p_k + (p_k^u - p_k^l) * \delta_k$$

$$\delta_k = \begin{cases} (2u)^{\frac{1}{(\eta_m+1)}}, & if\ u < 0.5 \\ 1 - [2(1-u)]^{\frac{1}{(\eta_m+1)}}, & otherwie \end{cases}$$

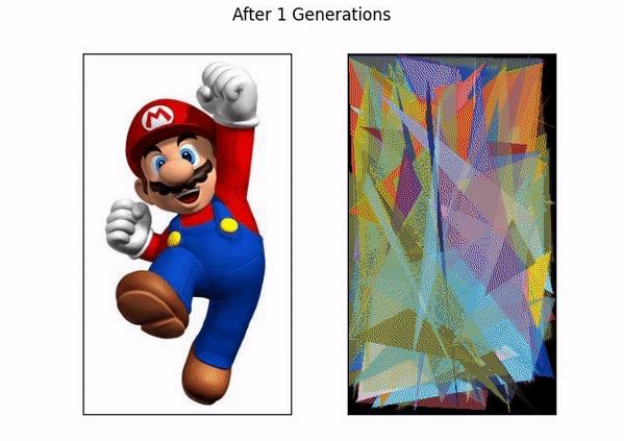


Демонстрація роботи

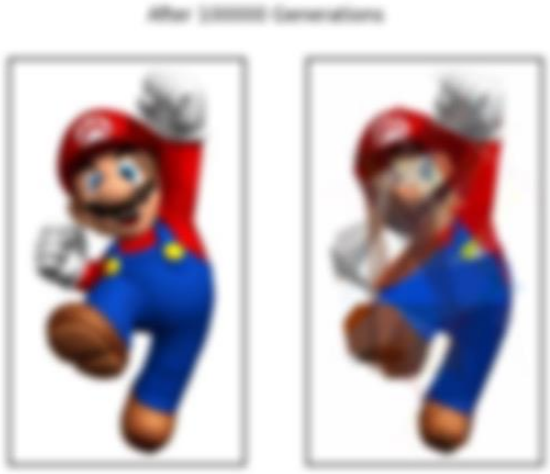
NUM_OF_POLYGONS = 100
POPULATION_SIZE = 200
MAX_GENERATIONS = 100000

Evaluate = MSE
Select = selTournament, tournsize=3
Mate = cxSimulatedBinaryBounded
Mutate = mutPolynomialBounded

elapsed time ~ 1.5 day



Висновки



ДОДАТОК Б

Лістинг програми

Б.1 Реалізація логіки обробки зображень

```
from PIL import Image, ImageDraw
import numpy as np
from skimage.metrics import structural_similarity
import cv2
import matplotlib.pyplot as plt

MAX_STEPS = 200
FLAG_LOCATION = 0.5


class ImageProcess:

    def __init__(self, imagePath, polygonSize):
        self.refImage = Image.open(imagePath)
        self.polygonSize = polygonSize

        self.width, self.height = self.refImage.size
        self.numPixels = self.width * self.height
        self.refImageCv2 = self.toCv2(self.refImage)

    def polygonDataToImage(self, polygonData):
        """
        accepts polygon data and creates an image containing
these polygons.
        :param polygonData: a list of polygon parameters. Each
item in the list
        represents the vertices locations, color and
transparency of the corresponding polygon
        :return: the image containing the polygons (Pillow
format)
        """

        # start with a new image:
        image = Image.new('RGB', (self.width, self.height))  #
TODO
        draw = ImageDraw.Draw(image, 'RGBA')

        # divide the polygonData to chunks, each containing the
data for a single polygon:
        chunkSize = self.polygonSize * 2 + 4  # (x,y) per vertex
+ (RGBA)
```

```
        polygons = self.list2Chunks(polygonData, chunkSize)

        # iterate over all polygons and draw each of them into
the image:
        for poly in polygons:
            index = 0

            # extract the vertices of the current polygon:
            vertices = []
            for vertex in range(self.polygonSize):
                vertices.append((int(poly[index] * self.width),
int(poly[index + 1] * self.height)))
                index += 2

            # extract the RGB and alpha values of the current
polygon:
            red = int(poly[index] * 255)
            green = int(poly[index + 1] * 255)
            blue = int(poly[index + 2] * 255)
            alpha = int(poly[index + 3] * 255)

            # draw the polygon into the image:
            draw.polygon(vertices, (red, green, blue, alpha))

        # cleanup:
        del draw

        return image

    def getDifference(self, polygonData, method="MSE"):
        # create the image containing the polygons:
        image = self.polygonDataToImage(polygonData)

        if method == "MSE":
            return self.getMse(image)
        else:
            return 1.0 - self.getSsim(image)

    def plotImages(self, image, header=None):
        fig = plt.figure("Image Comparison:")
        if header:
            plt.suptitle(header)

        # plot the reference image on the left:
        ax = fig.add_subplot(1, 2, 1)
        plt.imshow(self.refImage)
        self.ticksOff(plt)

        # plot the given image on the right:
        fig.add_subplot(1, 2, 2)
        plt.imshow(image)
        self.ticksOff(plt)
```

```
        return plt

    def saveImage(self, polygonData, imageFilePath,
header=None):
        """
        accepts polygon data, creates an image containing these
polygons,
        creates a 'side-by-side' plot of this image next to the
reference image,
        and saves the plot to a file
        :param polygonData: a list of polygon parameters. Each
item in the list
        represents the vertices locations, color and
transparency of the corresponding polygon
        :param imageFilePath: path of file to be used to save
the plot to
        :param header: text used as a header for the plot
        """

        # create an image from th epolygon data:
        image = self.polygonDataToImage(polygonData)

        # plot the image side-by-side with the reference image:
        self.plotImages(image, header)

        # save the plot to file:
        plt.savefig(imageFilePath)

    # utility methods:

    def toCv2(self, pil_image):
        """converts the given Pillow image to CV2 format"""
        return cv2.cvtColor(np.array(pil_image),
cv2.COLOR_RGB2BGR)

    def getMse(self, image):
        """calculates MSE of difference between the given image
and the reference image"""
        return np.sum((self.toCv2(image).astype("float") -
self.refImageCv2.astype("float")) ** 2) / float(
            self.numPixels)

    def getSsim(self, image):
        """calculates mean structural similarity index between
the given image and the reference image"""
        return structural_similarity(self.toCv2(image),
self.refImageCv2, multichannel=True)

    def list2Chunks(self, list, chunkSize):
        """divides a given list to fixed size chunks, returns a
generator iterator"""
        for chunk in range(0, len(list), chunkSize):
            yield (list[chunk:chunk + chunkSize])
```

```
    def ticksOff(self, plot):  # TODO
        """turns off ticks on both axes"""
        plt.tick_params(
            axis='both',
            which='both',
            bottom=False,
            left=False,
            top=False,
            right=False,
            labelbottom=False,
            labelleft=False,
        )
```

Б.2 Реалізація генетичного алгоритма з елітизмом

```
from deap import tools
from deap import algorithms

def eaSimpleWithElitismAndCallback(population, toolbox, cxpb,
mutpb, ngen, callback=None, stats=None,
             halloffame=None, verbose=__debug__):
    """This algorithm is similar to DEAP eaSimple() algorithm,
with two additions:
    1. halloffame is used to implement an elitism mechanism. The
individuals contained in the
       halloffame are directly injected into the next generation
and are not subject to the
       genetic operators of selection, crossover and mutation.
    2. a callback argument was added. It represents an external
function that will be called after
       each iteration, passing the current generation number and
the current best individual as arguments
    """
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats
else [])

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not
ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    if halloffame is None:
        raise ValueError("halloffame parameter must not be
empty!")

    halloffame.update(population)
    hof_size = len(halloffame.items) if halloffame.items else 0
```

```
    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
    if verbose:
        print(logbook.stream)

    # Begin the generational process
    for gen in range(1, ngen + 1):

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population) -
hof_size)

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb,
mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not
ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # add the best back to population:
        offspring.extend(halloffame.items)

        # Update the hall of fame with the generated individuals
        halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring

        # Append the current generation statistics to the
logbook
        record = stats.compile(population) if stats else {}
        logbook.record(gen=gen, nevals=len(invalid_ind),
**record)
        if verbose:
            print(logbook.stream)

        if callback:
            callback(gen, halloffame.items[0])

    return population, logbook
```

Б.3 Реалізація головної функції main

```
import os
import random
from datetime import timedelta
```

```
from timeit import default_timer as timer

import matplotlib.pyplot as plt
import numpy
import seaborn as sns
from deap import base
from deap import creator
from deap import tools

# problem related constants
import ElitismCallback
from ImageProcess import ImageProcess

POLYGON_SIZE = 3
NUM_OF_POLYGONS = 100

# calculate total number of params in chromosome:
# For each polygon we have:
# two coordinates per vertex, 3 color values, one alpha value
NUM_OF_PARAMS = NUM_OF_POLYGONS * (POLYGON_SIZE * 2 + 4)

# Genetic Algorithm constants:
POPULATION_SIZE = 200
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.5  # probability for mutating an individual
MAX_GENERATIONS = 150000
HALL_OF_FAME_SIZE = 20
CROWDING_FACTOR = 10.0  # crowding factor for crossover and
mutation

# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# create the image test class instance:
imageTest = ImageProcess.ImageProcess("images/mario.png",
POLYGON_SIZE)

# calculate total number of params in chromosome:
# For each polygon we have:
# two coordinates per vertex, 3 color values, one alpha value
NUM_OF_PARAMS = NUM_OF_POLYGONS * (POLYGON_SIZE * 2 + 4)

# all parameter values are bound between 0 and 1, later to be
expanded:
BOUNDS_LOW, BOUNDS_HIGH = 0.0, 1.0  # boundaries for all
dimensions

toolbox = base.Toolbox()

# define a single objective, minimizing fitness strategy:
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

```
# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMin)


# helper function for creating random real numbers uniformly
distributed within a given range [low, up]
# it assumes that the range is the same for every dimension
def random_float(low, up):
    return [random.uniform(l, u) for l, u in zip([low] *
NUM_OF_PARAMS, [up] * NUM_OF_PARAMS)]


# create an operator that randomly returns a float in the
desired range:
toolbox.register("attrFloat", random_float, BOUNDS_LOW,
BOUNDS_HIGH)

# create an operator that fills up an Individual instance:
toolbox.register("individualCreator",
                 tools.initIterate,
                 creator.Individual,
                 toolbox.attrFloat)

# create an operator that generates a list of individuals:
toolbox.register("populationCreator",
                 tools.initRepeat,
                 list,
                 toolbox.individualCreator)


# fitness calculation using MSE as difference metric:
def getDiff(individual):
    return imageTest.getDifference(individual, "MSE"),
    # return imageTest.getDifference(individual, "SSIM"),


toolbox.register("evaluate", getDiff)

# genetic operators:
toolbox.register("select", tools.selTournament, tournsize=3)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0 / NUM_OF_PARAMS)
```

```
# save the best current drawing every 100 generations (used as a
callback):
def saveImage(gen, polygonData):
    # only every 100 generations:
    if gen % 100 == 0 or gen == 1:

        # create folder if does not exist:
        folder = "images/results/run-{}-{}".format(POLYGON_SIZE,
NUM_OF_POLYGONS)
        if not os.path.exists(folder):
            os.makedirs(folder)

        # save the image in the folder:
        imageTest.saveImage(polygonData,
                            "{}/after-{}-gen.png".format(folder,
gen),
                            "After {} Generations".format(gen))


# Genetic Algorithm flow:
def main():
    # create initial population (generation 0):
    population = toolbox.populationCreator(n=POPULATION_SIZE)

    # prepare the statistics object:
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("min", numpy.min)
    stats.register("avg", numpy.mean)

    # define the hall-of-fame object:
    hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

    # perform the Genetic Algorithm flow with elitism and
'saveImage' callback:
    population, logbook =
ElitismCallback.eaSimpleWithElitismAndCallback(population,

toolbox,

cxpb=P_CROSSOVER,

mutpb=P_MUTATION,

ngen=MAX_GENERATIONS,

callback=saveImage,

stats=stats,

halloffame=hof,
```

```
verbose=True)

    # print best solution found:
    best = hof.items[0]
    print()
    print("Best Solution = ", best)
    print("Best Score = ", best.fitness.values[0])
    print()

    # draw best image next to reference image:
    imageTest.plotImages(imageTest.polygonDataToImage(best))

    # extract statistics:
    minFitnessValues, meanFitnessValues = logbook.select("min",
"avg")

    # plot statistics:
    sns.set_style("whitegrid")
    plt.figure("Stats:")
    plt.plot(minFitnessValues, color='red')
    plt.plot(meanFitnessValues, color='green')
    plt.xlabel('Generation')
    plt.ylabel('Min / Average Fitness')
    plt.title('Min and Average fitness over Generations')

    # show both plots:
    plt.show()


if __name__ == "__main__":
    start = timer()
    main()
    end = timer()
    print(timedelta(seconds=end - start))
```