МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Теория игр и исследование операций»

Тема: Релаксация линейного программирования.

Студент гр. 5381 _____ Кобылянский А.В.

Преподаватель _____ Шолохов А.В.

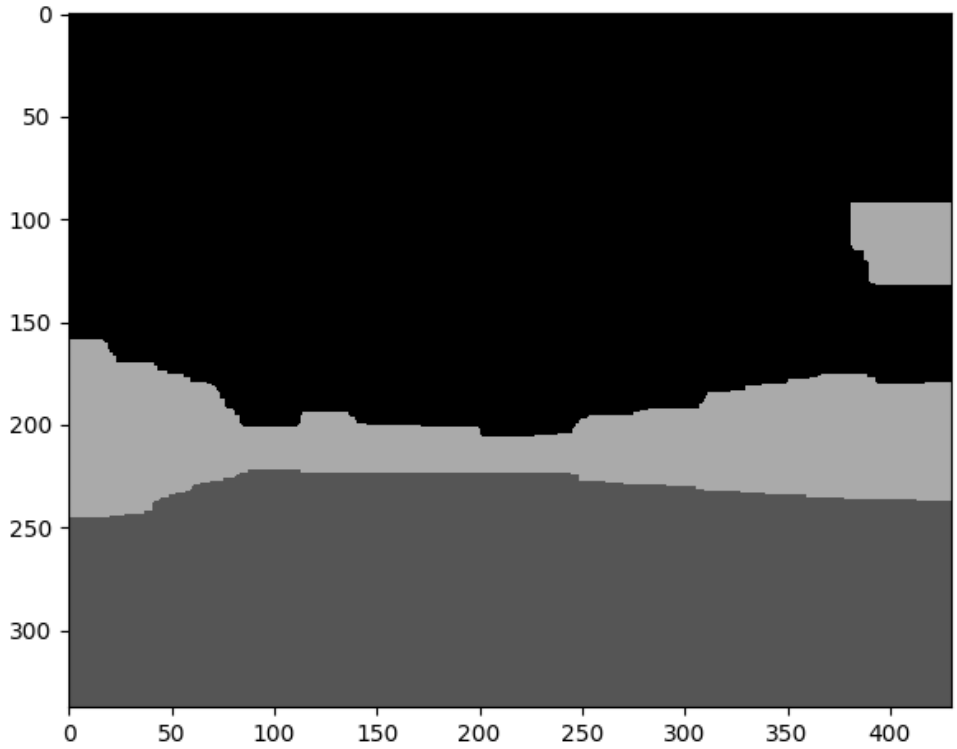
Санкт-Петербург
2018

Задание 1. Сегментация изображения.

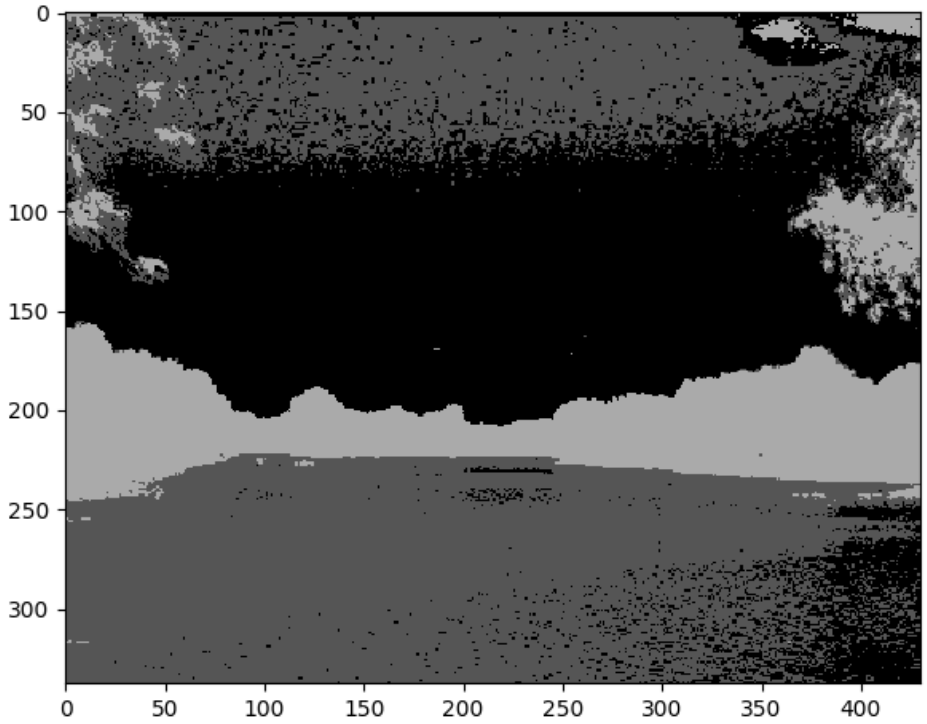
Код для решения задачи:

```
import numpy as np
from pystruct.inference import inference_dispatch
from pystruct.utils import make_grid_edges
from scipy.io import loadmat
import matplotlib.pyplot as plt
def task1():
	data = loadmat('data/mrf_potentials_segmentation.mat')
	unary_potentials = data.get('unary_potentials')  # type: np.ndarray (338, 430, 3)
	pairwise_potentials_vert = data.get('pairwise_potentials_vert')  # type: np.ndarray (3, 3)
	pairwise_potentials_horz = data.get('pairwise_potentials_horz')  # type: np.ndarray (3, 3)

	edges = make_grid_edges(unary_potentials)  # (289912, 2)
	height, width, n_states = unary_potentials.shape
	n_edges = edges.shape[0]
	pairwise_potentials = np.zeros(shape=(n_edges, n_states, n_states))
	for i, edge in enumerate(edges):
		vert_1, vert_2 = edge
		y1, x1 = divmod(vert_1, width)
		y2, x2 = divmod(vert_2, width)

		if abs(y1 - y2) == 1:
			# vertical edge
			pairwise_potentials[i, :, :] = pairwise_potentials_vert
		elif abs(x1 - x2) == 1:
			# horisontal edge
			pairwise_potentials[i, :, :] = pairwise_potentials_horz

	# QPBO because max-product works slow.
	result = inference_dispatch(-1.0*unary_potentials, -1.0*pairwise_potentials, edges, inference_method='qpbo')

	picture = result.reshape(height, width)
	plt.imshow(picture, vmin=0, vmax=n_states, cmap='gray')
	plt.show()
if __name__ == '__main__':
	task1()
```

Результат:



Результат работы программы с использование только унарных потенциалов (*inference_method*=“*unary*” в функции *inference_dispatch*).



Задание 2. Склеивание изображений.

Код программы:

```
import imageio
import matplotlib.pyplot as plt
import numpy as np
import visvis as vv
from numpy.linalg import norm
from pystruct.inference import inference_dispatch
from pystruct.utils import make_grid_edges
class ImagesData:
    def __init__(self, images_names, markup_name=None):
        self.shape = None
        self.images = []
        self.markup = None
        self.__load_images(images_names)
        if markup_name is not None:
            self.__load_markup(markup_name)
    def __check_shape(self, image):
        if self.shape is None:
            self.shape = image.shape
        else:
            if self.shape != image.shape:
                print('images shapes are not equile')
                print(self.shape, image.shape)
                exit(1)
    def __load_images(self, images_names):
        self.images = []
        for image_name in images_names:
            self.images.append(imageio.imread(image_name))
            self.__check_shape(self.images[-1])
    def __load_markup(self, markup_name):
        self.markup = imageio.imread(markup_name) // 20
        self.__check_shape(self.markup)
    def __scale_image(self, image, scale):
        height, width, colors = self.shape
        new_height, new_width = height // scale, width // scale
        new_image = np.zeros(shape=(new_height, new_width, colors))
        for x in range(new_width):
            for y in range(new_height):
                square = image[scale * y:scale * (y + 1), scale * x:scale * (x + 1)]
                new_image[y, x] = np.rint(np.mean(square, axis=(0, 1)))
        return new_image
    def __scale_markup(self, markup, scale):
        height, width, colors = self.shape
        new_height, new_width = height // scale, width // scale
        new_markup = np.zeros(shape=(new_height, new_width, colors))
        n_states = len(self.images)
        for x in range(new_width):
            for y in range(new_height):
                value = 0
                count = 0
                for i in range(scale * x, scale * (x + 1)):
                    for j in range(scale * y, scale * (y + 1)):
                        if markup[j, i, 0] < n_states:
                            value = markup[j, i, 0]
                            count += 1
                if count >= scale ** 2 / 2:
                    new_markup[y, x] = np.array([value, value, value])
                else:
                    new_markup[y, x] = np.array([255 // 20, 255 // 20, 255 // 20])
        return new_markup
    def scale(self, scale: int):
        # scale images and markup to 1/scale times
        # each new pixel is mean of scale x scale square in origin image
        self.images = [self.__scale_image(image, scale) for image in self.images]
```

```
            if self.markup is not None:
                self.markup = self.__scale_markup(self.markup, scale)
            height, width, colors = self.shape
            self.shape = (height // scale, width // scale, colors)
class Task2:
    def __init__(self, images_data: ImagesData, unary_potentials):
            self.imgs = images_data
            self.unary_potentials = unary_potentials
            self.__result = None
            self.path_to_dump_file = None
            self.diff_function = diff_standart
    def __compute_pairwise_potentials(self, edges):
            n_states = len(self.imgs.images)
            n_edges = edges.shape[0]  # 849946
            height, width, _ = self.imgs.shape
            pairwise_potentials = np.zeros(shape=(n_edges, n_states, n_states))

            for i, edge in enumerate(edges):
                if i % 1000 == 0:
                    print('{} / {}'.format(i // 1000, len(edges) // 1000))
                vert_1, vert_2 = edge
                y1, x1 = divmod(vert_1, width)
                y2, x2 = divmod(vert_2, width)
                for state_1 in range(n_states):
                    for state_2 in range(n_states):
                        if state_1 != state_2:
                            pairwise_potentials[i, state_1, state_2] = self.diff_function(self.imgs.images, state_1,
                                                                                          state_2, x1,
y1, x2, y2)
            return pairwise_potentials
    def __load_pairwise_potentials(self, path):
            n_states = len(self.imgs.images)
            res = np.fromfile(path)
            elements_count = res.shape[0]
            return res.reshape((elements_count // n_states ** 2, n_states, n_states))
    def __get_regions(self):
            if self.__result is not None:
                height, width, _ = self.imgs.shape
                return self.__result.reshape(height, width)
    def set_dump_file(self, path_to_dump_file: str):
            self.path_to_dump_file = path_to_dump_file
    def set_diff_function(self, diff_function):
            self.diff_function = diff_function
    def draw_regions(self):
            if self.__result is not None:
                regions = self.__get_regions()
                n_states = len(self.imgs.images)
                plt.imshow(regions, vmin=0, vmax=n_states)
                plt.show()
    def draw_picture(self):
            if self.__result is not None:
                regions = self.__get_regions()
                height, width = regions.shape
                picture = np.zeros(shape=(height, width, 3))
                for x in range(width):
                    for y in range(height):
                        region = regions[y, x]
                        picture[y, x] = self.imgs.images[region][y, x]
                vv.imshow(picture)
    def get_pairwise_potentials(self, edges, from_file, save_to_file):
            if (from_file or save_to_file) and self.path_to_dump_file is None:
                raise Exception('Path to dump file is not set')
            if from_file:
                pairwise_potentials = self.__load_pairwise_potentials(self.path_to_dump_file)
            else:
                pairwise_potentials = self.__compute_pairwise_potentials(edges)
            if save_to_file:
                pairwise_potentials.tofile(self.path_to_dump_file)
            return pairwise_potentials
```

```
    def compute(self, eta, from_file=True, save_to_file=False):
        unary_potentials = self.unary_potentials
        edges = make_grid_edges(unary_potentials)
        pairwise_potentials = self.get_pairwise_potentials(edges, from_file, save_to_file)
        print('start computing')
        self.__result = inference_dispatch(-unary_potentials, -eta * pairwise_potentials, edges,
                                           inference_method='qpbo')
        print('stop comuting')

def diff_standart(images, state_1, state_2, x1, y1, x2, y2):
    d1 = images[state_1][y1, x1] - images[state_2][y1, x1]
    d2 = images[state_1][y2, x2] - images[state_2][y2, x2]
    return norm(d1) + norm(d2)
def diff_alternative(images, state_1, state_2, x1, y1, x2, y2):
    d1 = images[state_1][y1, x1] - images[state_2][y1, x1]
    d2 = images[state_1][y2, x2] - images[state_2][y2, x2]
    d3 = images[state_1][y1, x1] - images[state_1][y2, x2]
    d4 = images[state_2][y1, x1] - images[state_2][y2, x2]
    return (norm(d1) + norm(d2)) / (norm(d3) + norm(d4) + 1)
def compute_unary_potentials_family(images_data: ImagesData):
    INF = 10 ** 5
    height, width, _ = images_data.shape
    n_states = len(images_data.images)
    markup = images_data.markup
    unary_potentials = np.zeros(shape=(height, width, n_states))
    for y in range(height):
        for x in range(width):
            if markup[y, x, 0] < n_states:
                for state in range(n_states):
                    if state != markup[y, x, 0]:
                        unary_potentials[y, x, state] = INF
    return unary_potentials
def compute_unary_potentials_pano(images_data: ImagesData):
    INF = 10 ** 5
    height, width, _ = images_data.shape
    n_states = len(images_data.images)
    unary_potentials = np.zeros(shape=(height, width, n_states))
    for y in range(height):
        for x in range(width):
            for state, image in enumerate(images_data.images):
                if np.all(image[y, x] == 0):
                    unary_potentials[y, x, state] = INF
    return unary_potentials
def task2_family():
    images_names = [
        'data/family/small_DSC_0168.png',
        'data/family/small_DSC_0170.png',
        'data/family/small_DSC_0173.png',
        'data/family/small_DSC_0174.png',
        'data/family/small_DSC_0176.png',
    ]
    markup_name = 'data/family/familydatacost.png'
    images_data = ImagesData(images_names, markup_name)
    images_data.scale(2)
    unary_potentials = compute_unary_potentials_family(images_data)
    task2 = Task2(images_data, unary_potentials)
    task2.set_dump_file('data/family_dump')
    task2.compute(eta=0.5, from_file=False, save_to_file=True)
    task2.draw_regions()
    task2.draw_picture()
    input('Press any key')
def task2_pano():
    images_names = [
        'data/pano/NQIMG_0257.PNG',
        'data/pano/NQIMG_0258.PNG',
        'data/pano/NQIMG_0259.PNG',
        'data/pano/NQIMG_0260.PNG',
        'data/pano/NQIMG_0261.PNG',
        'data/pano/NQIMG_0263.PNG',
```

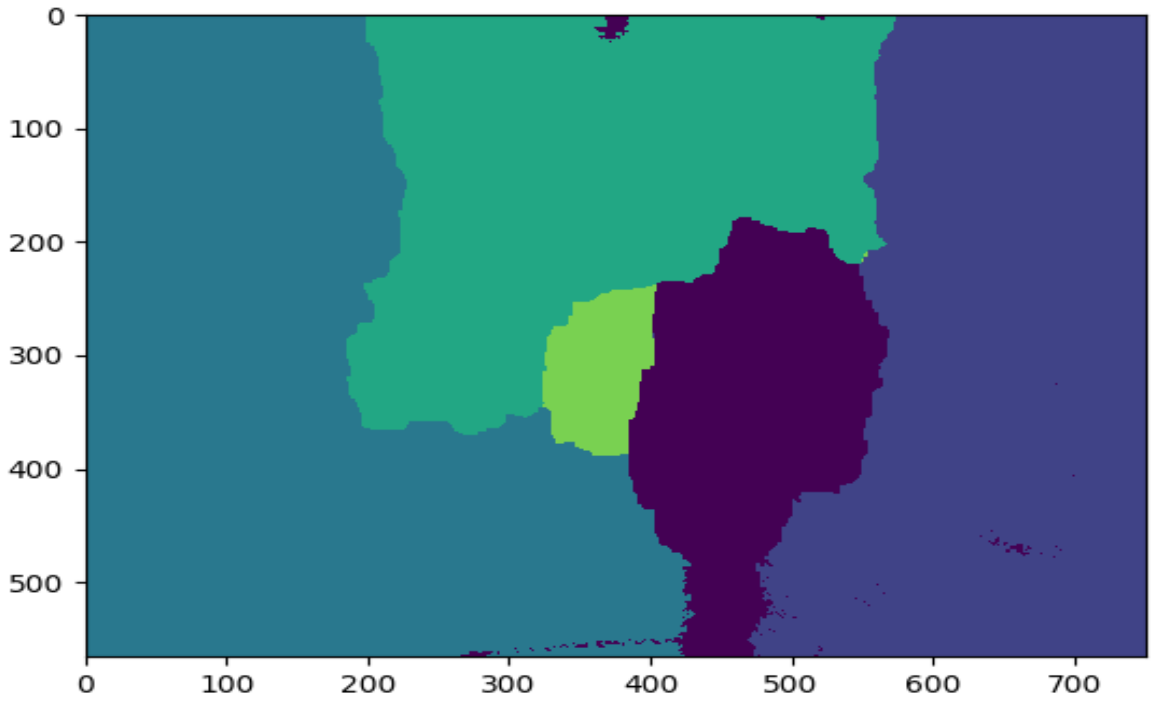
```
            'data/pano/NQIMG_0264.PNG',
        ]
        images_data = ImagesData(images_names)
        images_data.scale(4)
        unary_potentials = compute_unary_potentials_pano(images_data)
        task2 = Task2(images_data, unary_potentials)
        task2.set_dump_file('data/pano_dump2')
        task2.set_diff_function(diff_alternative)
        task2.compute(eta=0.5, from_file=False, save_to_file=True)
        task2.draw_regions()
        task2.draw_picture()
        # input('Press any key')
if __name__ == '__main__':
        # task2_family()
        task2_pano()
```

Семья.

$$\theta_{st}(y_s, y_t) = \begin{cases} 0, & y_s = y_t \\ \eta\left(\left\|I_{y_s}(s) - I_{y_t}(s)\right\| + \left\|I_{y_s}(t) - I_{y_t}(t)\right\|\right), & y_s \neq y_t \end{cases}$$

При различных η разбиения различаются, но картинка всегда почти одинаковая, без особых артифактов. Приведуем рещультат для $\eta = 0.5$

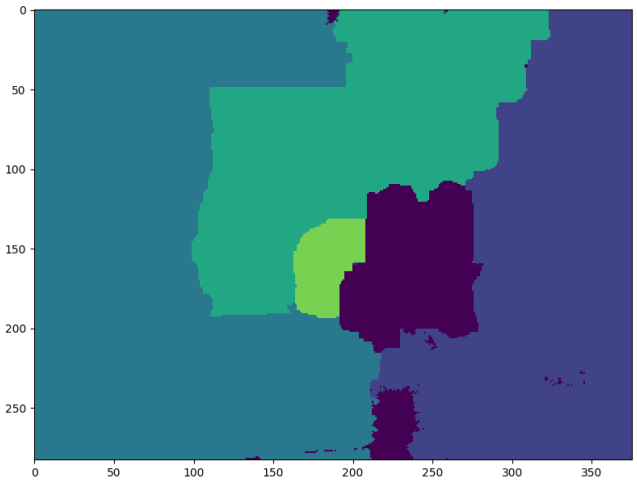
Разбиение:



Результат:

$$\theta_{st}(y_s, y_t) = \begin{cases} 0, & y_s = y_t \\ \eta \dfrac{\|I_{y_s}(s) - I_{y_t}(s)\| + \|I_{y_s}(t) - I_{y_t}(t)\|}{\|I_{y_s}(s) - I_{y_s}(t)\| + \|I_{y_t}(s) - I_{y_t}(t)\| + 1}, & y_s \neq y_t \end{cases}$$

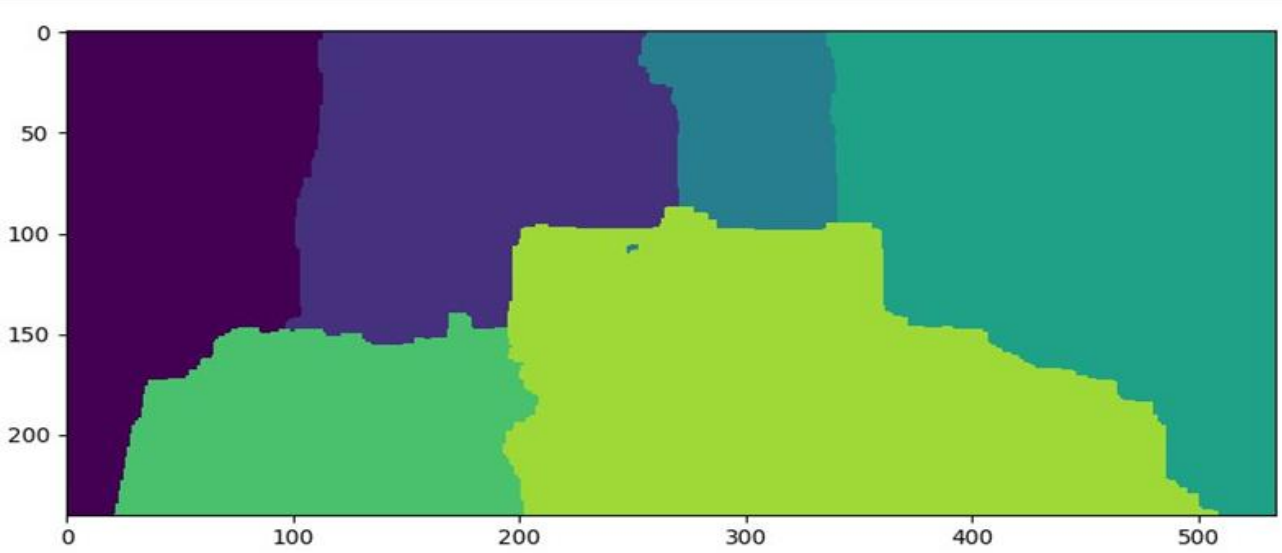
В формуле могло быть деление на ноль, так что я изменил её, добавив +1 в знаменатиле. Результат при $\eta = 0.5$ не особо отличается от первого.



Pano

$$\theta_{st}(y_s, y_t) = \begin{cases} 0, & y_s = y_t \\ \eta\left(\left\|I_{y_s}(s) - I_{y_t}(s)\right\| + \left\|I_{y_s}(t) - I_{y_t}(t)\right\|\right), & y_s \neq y_t \end{cases}$$

Разбиение:



Результат:

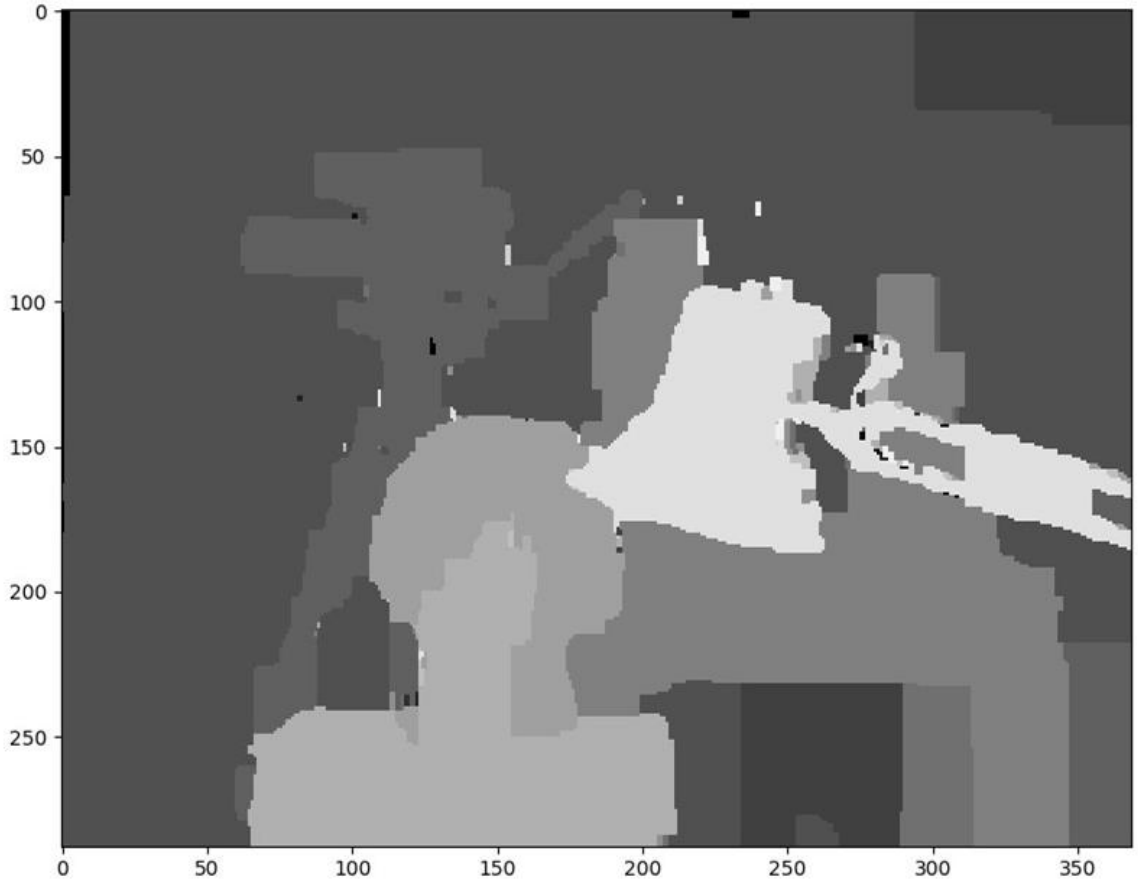
Результат содержит очевидные артефакты склеивания – “обрезанных” людей на леснице. При альтернативном задании $\theta_{st}(y_s, y_t)$ и η все становится только хуже.

Задание 3. Стереозрение.

Код для решения задачи:

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
from pystruct.utils import make_grid_edges
from pystruct.inference import inference_dispatch
from utils import my_inference_dispatch
from skimage.color import rgb2yuv
import imageio
import visvis as vv
T = 1
W = 0.05
SIGMA = 18
K = 15
def load_image(filename: str) -> np.ndarray:
    img = imageio.imread(filename)  # type: Image.Image
    data = rgb2yuv(img)
    return data
def task3():
    n_states = K + 1
    image_left = load_image('./data/imL.png')   # (288, 384, 3)
    image_right = load_image('./data/imR.png')  # (288, 384, 3)
    assert image_left.shape == image_right.shape
    height, width, _ = image_left.shape  # (288, 384, 3)
    cutted_width = width - K
    unary_potentials = np.zeros( shape=(height, cutted_width, n_states))
    for x in range(cutted_width):
        for y in range(height):
            for state in range(n_states):
                diff = image_left[y, x + state] - image_right[y, x]
                unary_potentials[y, x, state] = min(np.linalg.norm(diff), SIGMA)

    edges = make_grid_edges(unary_potentials)  # (212462, 2)
    pairwise_potentials = np.zeros(shape=(n_states, n_states))
    for state_1 in range(n_states):
        for state_2 in range(n_states):
            pairwise_potentials[state_1, state_2] = W*min(abs(state_1 - state_2), T)

    # QPBO because max-product works slow.
    result = inference_dispatch(-unary_potentials, -pairwise_potentials, edges, inference_method='qpbo')

    picture = result.reshape(height, cutted_width)
    plt.imshow(picture, vmin=0, vmax=n_states, cmap='gray')
    plt.show()
if __name__ == '__main__':
    task3()
```

Результат для K = 15: $\sigma = 18$, $\tau = 2$, $w = 0.05$ со всеми связям



Результат только с унарными потенциалами: