Report

Student: Anna Boronina

Group: 5

I think that the most interesting type of genetic algorithms is a Multi-Agent Genetic Algorithms (MAGA). The idea of it is that many agents cooperate and help each other increase fitness of the population. Best agents mate with a bit less best yet not worst to create a new agent which might get into next population

Definition of art

My definition of art:

Art is the *result* of our perception influence on the way we see and reflect the world and our place in it.

Two randomly selected people will never see the world the same way, because we all went through different experiences and have different interests. Even one's current state of mind affects the way he/she sees things.

Result can be anything because it comes from our own perception, current state of mind, atmosphere around. In some cases, it can be one sad piece of canvas with visible attempts to create art.

Why is it art?

The idea of my algorithm is to

1. create a **perception** of the input image - my algorithms adds blur and places random black figures.
2. try to **produce** an output image taking in count the way the input image was perceived

This was of creating art corresponds to the definition of art I gave.

The reason to add blur to the picture and draw random black figures is connected to the state of mind I've been for the last few years. I have mental issues which cause periodical switches between depression and high productivity. The time I started to think about what I want to define as art, I had an episode of depression and I thought that I could take this assignment as an opportunity to show the way I see things when I'm depressed: everything is blur, some things are out of order and some things don't even reach me. These things are hid behind the black circles and rectangles.

The output is the result of the genetic algorithm - the best attempt to show the way the input image was perceived.

Image manipulation techniques

For all image manipulations I used OpenCV library. In more details:

1. *cv2.filter2D()* to blur the input image.
2. Color conversion from RGB to grayscale using *cv2.cvtColor()*. It was used for two reasons: to decrease the runtime and to add darkness to my output picture.

3. The image was represented as an array 512x512x1. Since the input image is converted to grayscale, we need only 1 number (intensity) to represent a pixel.

Genetic Representation

For each population, there are 1 or several MAs. Each of MA has many individuals - agents, each of which draws some part of the new picture. So, agents, that belong to the same MA, cooperate, while MAs should "fight" because only the fittest one of them will get to represent the final result. My laptop is not powerful enough to support many multi-agents, that's why in my algorithm I used only one MA as population. It doesn't change anything, because the amount of MAs created initially would stay the same till the very end). So, we can say that it's one MA that evolves and improves itself over generations.

So, genetic representation of my GA is one multi-agent.

Multi-agent:

- agents[] - all the agents that belong to this MA, each in agents[] is an object created from class Agent() with the following fields:
    - polygon - information about polygon that it being drawn by the agent: position, length, radius, etc
    - colors[][] - array representing pixels for the polygon of this agent.
    - score - personal score of this agent. The closer agent's polygon to the real picture in the same part of it
- fit - fit-score of the multi-agent

Evolution process and selection mechanism

The whole MAGA is ran in *evolution()*.

1. We create initial multi-agent
2. Run while-loop for predefined time limit
3. Inside this while-loop we create new multi-agent, then if it fits, set it to be our current multi-agent and previous multi-agent dies. Or, if newly formed multi-agent is not a good fit, we simply don't take it.

   In more details:

   Field *fit* of a multiagent compare previous generation (*MA0*) to the generation that was just created (*MA1*). If *MA1*'s fit is worse than *MA0*'s fit, then it dies immediately and the next generation (*MA1_v2*) is created. If *MA1_v2* is not good enough again, there will be *MA1_v3*, and so on, until next generation (*MA1_v<i>*) is a better fit than *MA0*.
4. We repeat the loop and looking for a better fit.

```python
# simulate evolution
def evolution(src_img):
    print("starting...")
    # create an initial multi-agent and generate initial amount of agents
    multiagent = MultiAgent()
    multiagent.generate_agents(INIT_AMOUNT_OF_AGENTS)
    # calculate fitness of initial multi-agent
    multiagent.fitness(src_img)

    save_image(multiagent, 'tmp.jpg')
    time_init = time.time(); time_now = time_init

    # loop to keep the algorithm running
    while len(multiagent.agents) > 20 and time_now-time_init < TIME_LIMIT_IN_SEC:
        # form new generation
        new_multiagent = form_new_gen(multiagent)
        # calculate fitness of the newly formed generation
        new_multiagent.fitness(src_img)

        # newly formed generation will be next if and only if its fitness
        # is higher that the previous generation's fitness
        if new_multiagent.fit > multiagent.fit:
            print("old fit: ", multiagent.fit, "; ", "found better: ", new_multiagent.fit)
            print("new multi-agent has ", len(new_multiagent.agents), " agents")
            multiagent = new_multiagent
            save_image(multiagent, 'tmp.jpg')

        time_now = time.time()
    return multiagent
```

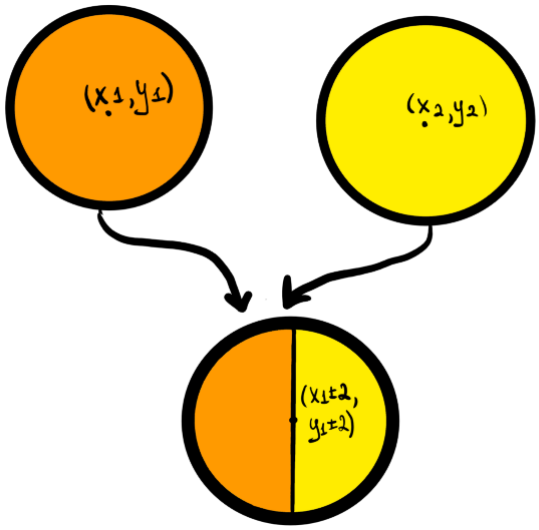
New generation creation

1. Create new generation (multi-agent)
2. Take the best agents from the previous generation
3. Add some new randomly created agents to keep diversity
4. Perform crossover depending on amount of best agents retrieved from the previous generation. It's needed not to overflow the generation and not to keep it too small.
5. Perform mutation. You can read more about crossover and mutation in the following sections

```python
# forms new generation based on the old one
def form_new_gen(old_gen):
    old_best_agents = []
    # select best agents from the previous generation
    old_best_agents = old_gen.agents[:len(old_gen.agents)//4]
    # create new multi-agent
    new_gen = MultiAgent()
    # add a few of random agents to keep diversity and prevent "grouping" of polygons
    new_gen.generate_agents(55)

    crossed, mutated = [], []
    # control of amount of kids from crossover
    if len(old_best_agents) < 50:
        crossed = crossover(old_best_agents, 4)
    elif len(old_best_agents) < 100:
        crossed = crossover(old_best_agents, 2)
    else:
        crossed = crossover(old_best_agents, 1)

    # mutation probability is 66%
    mutated = mutation(old_best_agents)

    for each in crossed:
        new_gen.agents.append(each)
    for each in mutated:
        new_gen.agents.append(each)

    for each in new_gen.agents:
        each.score = 0

    return new_gen
```

Crossover

Crossover is preferable to happen for two best agents. So, a mate for each agent is chosen at random from the the first 10 agents. I decided not to put screenshots of the functions that cross the genes and copy colors. The important thing is that crossover is successful is a kid has properties of both parents. So, when I create a kid, half of its colors is taken from one parent and another half is taken from another parent.

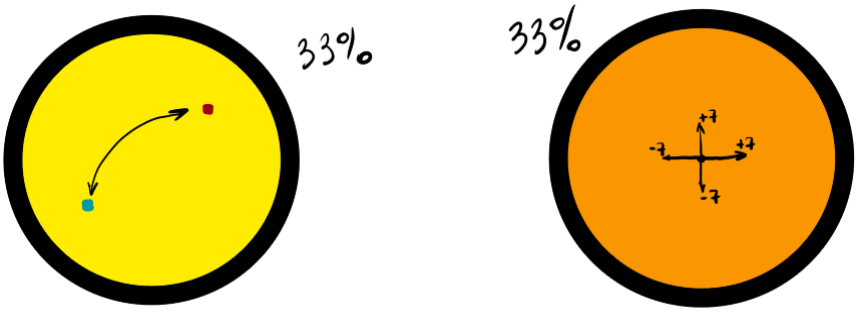


```
# perform a crossover between two agents
def crossover(agents, max_kids):
    new_agents = []
    for i in range (1, len(agents)):
        if np.random.randint(0,2) == 0:
            # agents are sorted from best to not so best so let's cross best with best
            j = np.random.randint(0, 10)
            tmp_agents = cross_genes(agents[i], agents[j], max_kids)
            for k in range (len(tmp_agents)):
                new_agents.append(tmp_agents[k])
    return new_agents
```

Mutation

New agents are created in order not to modify the agents from the previous generation in case this generation won't be a good fit. There are two types of mutation: swapping pixels within one polygon and changing position of the polygon (it can go maximum. Each happens with the probability 33%.

On average, swapping pixels swaps 15 pairs within the polygon.




```
# perform mutation
def mutation(agents):
    new_agents = []
    for agent in agents:
        flag = np.random.randint(0, 3)
        if flag == 0:
            # probability of swapping colors inside the polygon - 33%
            new = Agent()
            new.copy_from(agent)
            new.swap_pixels()
            new_agents.append(new)
        elif flag == 1:
            # probability of small change of position (x+-7, y+-7) - 33%
            new = Agent()
            new.copy_from(agent)
            new.change_position()
            new_agents.append(new)
    return new_agents
```

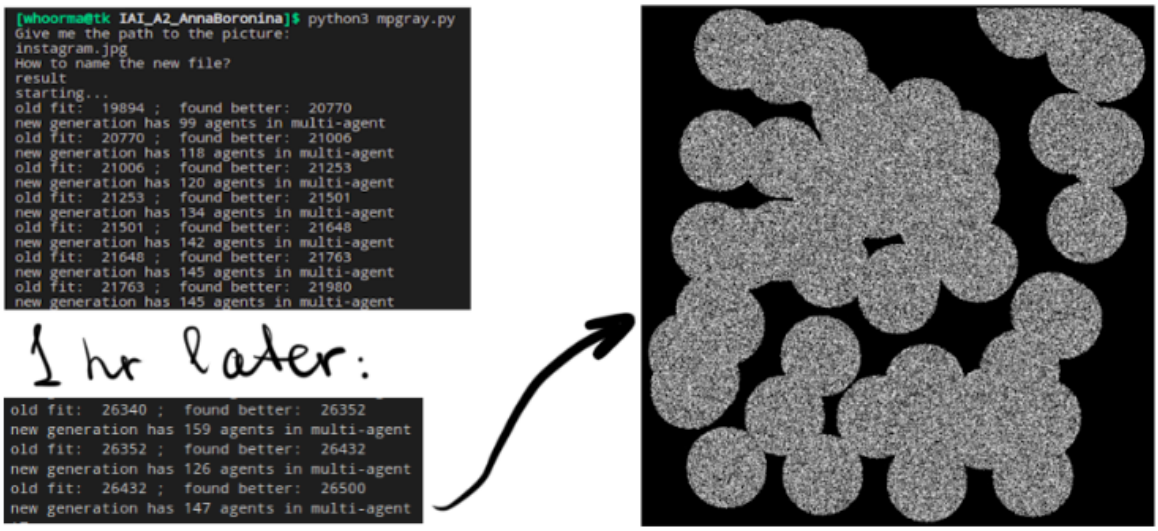
Fitness Function

Function *fitness()* is in class Multi-agent. It calculates score for each agent of multi-agent's agents, then sorts them in descending order, and then calculates fitness of the multi-agent. Then this fitness is used to decide whether this generation (multi-agent) survives or not. Read about it in section "Selection mechanism".

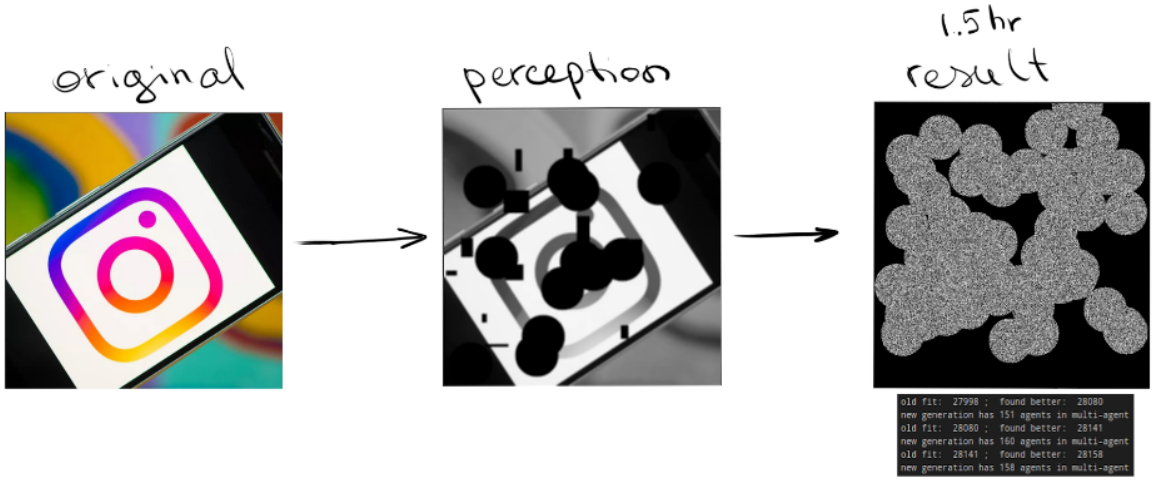
```python
# calulates fitness of the multiagent
def fitness(self, src_img):
    for agent in self.agents:
        agent.score = agent.calc_score(src_img)
    self.fit = 0
    self.quicksort_agents(0, len(self.agents)-1)
    for i in range(20):
        self.fit += self.agents[i].score
```

Examples

While algorithm is running, we can see the process in console (see the picture below), and from what I saw, the population was improving but it was very-very slow. After 60 minutes of running, I got the intermediate output that you can see below.



Then I left it running for 30 more minutes and maximum fit it reached was 28158 (see the picture below). The difference between initial fit and final is 7388.



Personal reflection

I was very excited about my idea at first and I spent a lot of time thinking what I want to see as the result and how to formulate my definition of art. It's a very interesting assignment because we have freedom to do what *we* want.

In theory, everything looked very good, I even created many different shapes: circles, rectangles and triangles, all of random shapes. I removed randomness and unnecessary shapes but the algorithm still didn't converge. I started to play with coefficients, sizes of circles and initial amount of agents in the multi-agent.

In my opinion, I should have created 4 (or more) groups of agents for one multi-agent, for example: left_up, left_bottom, right_up, right_bottom. And each of them would have several agents drawing only their part and not touching any other part. In this way, they would cooperate and maybe produce a better result.

I think I had to use changing pixels as mutation as well, because I implemented it and it could make a big change. For some reason, I forgot about this function.

So, to sum up, I think I had to spend much more time on testing and playing with numbers and ideas and I would get much better results.