Applications of Genetic Algorithms to Procedural Image Generation

Akhila Ananthram & Griffin Brodman
asa225 & gb282

December 16th

1 Introduction

Genetic Algorithms are a very interesting and unexplored area of Artificial Intelligence. We decided it would be interesting to apply genetic algorithms to different solved problems, and see if we could find improvements in performance.

The problem we decided to analyze was that of generating an image out of translucent polygons. We found a project by Roger Alsing that given a source image, will modify a canvas by adding polygons, removing polygons, adding or subtracting vertexes to polygons, moving polygons, changing their color or their transparency. He claimed that he was using genetic algorithms, but on inspection, he had actually used hill stepping. He would generate a child through mutation, see if this was closer to the source image, if it was keep it, if else, throw it away. This is not really an example of genetic algorithms as there is a guarantee that the next generation is better than the previous one. This is not guaranteed for genetic algorithms as they are stochastic. We were wondering if genetic algorithms could be applied to this problem, and even show some improvements in performance.

We theorized that looking at how genetic algorithms change the performance of this algorithm would give us a much better understanding of real world applications of genetic algorithms, and what their best use case is. From here, we could more accurately identify problems that would benefit from this approach. We varied our approach to genetic algorithms, and tried a several modifications to it, such as elitism and multiple parents. We then planned to compare the amount of iterations it takes, on average, for hill stepping vs genetic algorithms to get to a certain fitness threshold when compared to the source image.

We found that, at least for this type of problem, hill stepping was a far faster and superior approach to the problem. We ca not speak for sure about the number of iterations needed to get to a certain fitness, but we can say that the time needed for one iteration in hill stepping is several magnitudes lower than the time needed for genetic algorithms.

2 Problem Definition and Methods

2.1 Task Definition

We wanted to analyze whether hill stepping or genetic algorithms would have better performance in solving the problem of generating an image with procedurally generated polygons. We looked to see, in numbers of iterations, how long it would take our multiple methods to get to a certain fitness. We first ran it under hill stepping, then under a variety of genetic algorithm modifications.

2.2 Algorithms and Methods

We developed two different fitness functions, and sampled using both. Our basic fitness function was one that went pixel by pixel, and calculated the euclidean distance between the two images. Our second one relied on opencv feature matching. We calculated the descriptors for the source image and the current image. Then we used a brute force matcher to find the matches and then calculated the sum of the distances between the matches.

Our hill stepping algorithm is simple. We mutate an array of polygons, either adding or removing a polygon, adding or removing a vertex to a polygon, or changing the color or transparency of a polygon. If this new array has a better fitness than the old array we keep it, else we throw it away.

Our basic genetic algorithm follows the methods covered in class. First we created a population of random instances, where an instance is a list of polygons. Then we evolve the population by crossbreeding and applying mutations to the children. We do this until it converges. The list of possible mutations is the same for both the hill stepping and genetic algorithm.

For crossbreeding, we select two parents based on their fitness. We calculated the fitness of the entire population. Based on the fitness scores, we assigned a probability to each individual of how likely it was to be a parent. We then randomly selected two parents to crossbreed. When selecting genes from a parent, we used reservoir sampling. First we determined the number of genes we would want from a parent. We simply picked the average length of the parents. Our implementation of reservoir sampling follows the wiki page.

We added variations to this basic method to tune the genetic algorithm. This includes niche penalty, elitism, random individuals, and varying our parameters. Individuals whose difference in fitness score was within a certain threshold are part of a niche. For these individuals, we applied a penalty to avoid having our algorithm converge to this local minimum. Elitism is the idea that certain parents can live on to the next generation if they are the most fit. The idea behind this is that the most fit should be able to survive. To avoid approaching a local minimum, we also added a random person to the population every few generations. Lastly, we also made our program capable of varying the parameters to better tune the algorithm.

3 Experimental Evaluation

3.1 Methodology

Because our finished product was an image, much of our analysis was qualitative. This is also how we determined our threshold for the fitness. To compare the results of hill stepping and all the variations on genetic algorithms, we had planned to use this threshold and compare the number of iterations needed to reach this level.

We also wanted to compare speed and memory usage of the two algorithms, as we knew that genetic algorithms were more computationally intensive.

3.2 Results

Unfortunately, we didn't get to test this as much as we had planned. Python ended up being extremely slow, so even one iteration of genetic algorithms would take about an hour, whereas our hill stepping, where an iteration would take a few seconds at the most, needed over a thousand iterations to converge to acceptable fitness. Additionally, because we were using Python, we did not have the ability to manage memory ourselves. Thus, we eventually experienced memory problems as Python's garbage collector did not recognize that we were no longer using certain generations.

We attempted to fix both of these problems, starting with speed. Following the usual methods to improve speed, we wanted to parallelize the program as much as possible. Our first attempt was using Python's ThreadPool. Unfortunately, Python uses a Global Interpreter Lock. Thus, adding multi-threading did not improve the performance. In fact, it actually slowed us down. We then looked into multi-processing. Because of hardare limitations, we decided to use 3 sub-processes. We were able to parallelize the fitness function and the creation of children. The addition of multi-processing improved our performance. However, we reached an interesting roadblock on Windows with multi-processing when we were attempting to access global variables from a sub-process. On Unix based systems, Python's multiprocessing uses fork(), giving every child process a copy of its parent's address space, including global variables. However, this is not the case for Windows. Any variable from a parent process that is accessed by a child must be explicitely passed along. Also, to improve speed was to allocate all arrays we were planning to use as part of their initialization instead of using Python's append. Our hope with this was to avoid having to allocate memory continuously and save time. Lastly, we attempted to speed up our program by adding the ability to apply the fitness function to just a sample of the image. As the polygons are never going to match the picture's pixels exactly, we can look at just a sample. This change is not noticable to the human eye, but saves time.

To address the memory problem, we looked into ways to manually manage Python's memory. Unfortunately, Python is not designed well for manual memory management. There are a few libraries that can help identify memory leaks and then we can use this knowledge to manually call del on an object so Python's garbage collector can take clean it. However, they do not support multi-proccessing and thus we were unable to improve in this area.

4 Related Work

As our results were magnitudes slower, we wanted to see problems that were more easily solved with genetic algorithms. Examples included optimizations of telecommunications routing, applications of the traveling salesman problem, and the aerodynamics of automotive design. After thinking about all these issues, we realized that the reason these used genetic algorithms was that the fitness to evaluate all the potential children was trivial and cheap. Our fitness function ended up being our bottleneck, it was very computationally intensive, so it turned out that genetic algorithms would be a far inferior choice to basic hill stepping.

5 Future Work

There were a few things that developed as we worked on the project. For one, working in Python severly limited our speed. We did get the results we needed, but we were constrained to smaller images. It would have been nice to support much larger images, which would have opened up different fitness functions to us, as it would have improved feature matching. The project we had used as inspiration had been written in c#, and though we didn't think the difference in languages would have such a significant effect, it seems to be huge.

Because this is a very visual project, we also would expand on a visualization of the process. Users could find it useful and interesting to see how the population progresses on each iteration, as the image starts to look more and more like the source.

6 Conclusion

In the end, we did accomplish our goal of comparing genetic algorithms to hill stepping in the setting of procedural image generation through transparent polygons. We learned that genetic algorithms are best used in situations that are computationally intensive. We also learned through our research more about genetic algorithms, and discovered several variations that went beyond what we learned in lecture.

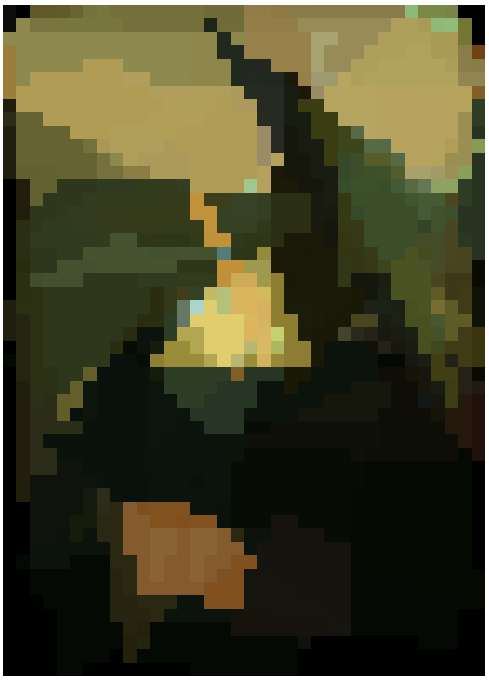
7 Acknowledgements

Professor Selman

8 References

http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/


CORNELL UNIVERSITY
FOUNDED A.D. 1865



```
import cv2
from multiprocessing import Pool
import numpy as np
import random
import argparse
import copy
import math
import time
import sys
import json

THRESH = None
WIDTH = 0
HEIGHT = 0

class Thresholds(object):
    def __init__(self, threshold_file, pop_size):
        #default initiation
        #POLYGON MUTATION
        self.opacity = .2
        self.red = .2
        self.green = .2
        self.blue = .2
        self.points = .2
        self.remove_point = .5
        #POPULATION MUTATION
        self.pop_mutate_poly = .4
        self.modify_list = .6
        self.remove = 0.05
        #EVOLVE
        self.niche = 0
        self.mutation = .1
        self.elitism = None
        self.add_random = 0

        if threshold_file is not None:
            #use json to read in dictionary
            with file(threshold_file) as f:
                thresholds = json.load(f)

                #POLYGON MUTATION
                polygon = thresholds.get("polygon", {})
                self.opacity = polygon.get("opacity", self.opacity)
                self.red = polygon.get("red", self.red)
                self.green = polygon.get("green", self.green)
                self.blue = polygon.get("blue", self.blue)
                self.points = polygon.get("points", self.points)
                self.remove_point = polygon.get("remove", self.remove_point)

                #POPULATION MUTATION
                population = thresholds.get("population", {})
                self.pop_mutate_poly = population.get("mutate", self.pop_mutate_poly
)
                self.modify_list = population.get("modify", self.modify_list)
                remove = population.get("remove", self.remove)
                if 0 <= remove <= 1:
                    self.remove = remove

                #EVOLVE
                evolve = thresholds.get("evolve", {})

                self.niche = abs(evolve.get("niche", self.niche))
                mutation = evolve.get("mutate", self.mutation)
                if 0 <= mutation <= 1:
                    self.mutation = mutation

                self.add_random = evolve.get("random", self.add_random)

                elitism = evolve.get("elitism", self.elitism)
                #not valid elitism
                if elitism <= 0 or elitism is None or elitism > pop_size:
                    self.elitism = None
                #already proportion
```

```
                elif elitism < 1:
                    self.elitism = elitism
                #make into proportion
                else:
                    self.elitism = elitism / pop_size

        total = self.opacity + self.red + self.green + self.blue + self.points + sel
f.remove_point
        self.opacity = self.opacity / total
        self.red = self.red / total
        self.green = self.green / total
        self.blue = self.blue / total
        self.points = self.points / total
        self.remove_point = self.remove_point / total

        total = self.pop_mutate_poly + self.modify_list
        self.pop_mutate_poly = self.pop_mutate_poly / total
        self.modify_list = self.modify_list / total

        self.polygon = [self.opacity, self.red, self.green, self.blue, self.remove_p
oint]
        for i in xrange(1, len(self.polygon)):
            self.polygon[i] += self.polygon[i - 1]

class Polygon(object):
    def __init__(self, points=None, red = None, blue = None, green = None, opacity =
 None):
        self.points = points
        if points is None:
            self.points = [None] * 3

            x = int(random.random() * WIDTH)
            y = int(random.random() * HEIGHT)
            self.points[0] = [x,y]
            x = int(random.random() * WIDTH)
            y = int(random.random() * HEIGHT)
            self.points[1] = [x,y]
            x = int(random.random() * WIDTH)
            y = int(random.random() * HEIGHT)
            self.points[2] = [x,y]
            self.order_vertices()

        self.red = red
        if red is None:
            self.red =   int(random.random() * 256)

        self.blue = blue
        if blue is None:
            self.blue =  int(random.random() * 256)

        self.green = green
        if green is None:
            self.green = int(random.random() * 256)

        self.opacity = opacity
        if opacity is None:
            self.opacity = random.random()

    def add_vertex(self):
        x = int(random.random() * WIDTH)
        y = int(random.random() * HEIGHT)
        self.points.append([x,y])
        self.order_vertices()

    def order_vertices(self):
        #calculate center point
        xc = 0.0
        yc = 0.0
        for x, y in self.points:
            xc += x
            yc += y

        xc = xc / len(self.points)
```

```
        yc = yc / len(self.points)

        #sort
        self.points = sorted(self.points, key=lambda p: math.atan2(p[1] - yc, p[0] -
 xc))

    def remove_vertex(self):
        to_remove = int(random.random() * len(self.points))
        self.points.pop(to_remove)

    def change_opacity(self):
        self.opacity = random.random()

    def change_red(self):
        self.red = int(random.random() * 256)

    def change_green(self):
        self.green = int(random.random() * 256)

    def change_blue(self):
        self.blue = int(random.random() * 256)

    def mutate(self):
        mutation = random.random()
        if(mutation < THRESH.polygon[0]):
            self.change_opacity()
        elif(mutation < THRESH.polygon[1]):
            self.change_red()
        elif(mutation < THRESH.polygon[2]):
            self.change_green()
        elif (mutation < THRESH.polygon[3]):
            self.change_blue()
        else:
            if(len(self.points)> 3):
                if(random.random() < THRESH.remove_point):
                    self.remove_vertex()
                else:
                    self.add_vertex()
            else:
                self.add_vertex()

    def __str__(self):
        poly = {
            "points" : self.points,
            "red" : self.red,
            "blue" : self.blue,
            "green" : self.green,
            "opacity" : self.opacity
            }

        return json.dumps(poly)

def euclidean_helper(args):
    img, original, wstart, wend, step = args
    '''assumes img and self.original have the same size'''
    #set up
    distance = 0.0

    for i in xrange(wstart, wend, step):
        for j in xrange(0, original.shape[0], step):
            distance += np.linalg.norm(img[j][i] - original[j][i])

    return distance

class Fitness(object):
    def __init__(self, original, type="euc", sample=1, pool=None):
        self.original = original

        self.type = type
        if type == "euc":
            if pool is not None:
                self.pool = pool
                self.num_proc = pool._processes
```

```
            else:
                self.num_proc = 3
                self.pool = Pool(self.num_proc)

            self.wstarts = [int((WIDTH / self.num_proc) * i) for i in xrange(self.nu
m_proc)]
            self.wends = [int((WIDTH / self.num_proc) * (i + 1)) for i in xrange(sel
f.num_proc)]
        elif type == "feat":
            self.detector = cv2.ORB()
            self.kp, self.desc = self.detector.detectAndCompute(self.original, None)
            self.matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

        self.step = sample

    def euclidean(self, img):
        '''assumes img and self.original have the same size'''
        distance = 0.0

        #pass to pool
        results = self.pool.map(euclidean_helper, [(img, self.original, self.wstarts
[i], self.wends[i], self.step) for i in xrange(self.num_proc)])

        for r in results:
          distance += r

        distance = distance / (WIDTH * HEIGHT)

        return distance

    def feature_matching(self, img):
        kp, desc = self.detector.detectAndCompute(img, None)

        matches = self.matcher.match(self.desc, desc)

        distance = 0.0

        for m in matches:
            distance += m.distance

        if len(matches) != 0:
            distance = distance / len(matches)


        if distance == 0:
            return sys.maxint

        return 1 / distance

    def score(self, img):
        if (self.type == "euc"):
            return self.euclidean(img)
        elif (self.type == "feat"):
            return self.feature_matching(img)

def mutate(plys):
    val = random.random()
    if(val < THRESH.pop_mutate_poly and len(plys) != 0):
        to_mutate = random.randrange(0, len(plys))
        plys[to_mutate].mutate()
    else:
        if len(plys) > 0:
            if (random.random() < THRESH.remove):
                to_remove = int(random.random() * len(plys))
                p = plys.pop(to_remove)
                del p
            else:
                plys.append(Polygon())
        else:
            plys.append(Polygon())

class Driver(object):
    def __init__(self, args):
```

```
        global WIDTH, HEIGHT
        self.original = cv2.imread(args.path)
        WIDTH = self.original.shape[1]
        HEIGHT = self.original.shape[0]

        self.num_proc = 3
        self.pool = Pool(self.num_proc)
        self.fit = Fitness(self.original, args.fitness, args.sample, self.pool)
        self.iterations = args.iterations
        self.max_poly = 1

    def draw(self, polygons):
        img = np.zeros(self.original.shape)

        for p in polygons:
            poly = np.array(p.points, np.int32)
            h, w, _ = img.shape
            mask = np.zeros((h,w))
            cv2.fillPoly(mask, [poly], 1)

            #opacity
            for i in xrange(w):
                for j in xrange(h):
                    #polygon exists there
                    if mask[j][i]:
                        b0, g0, r0 = img[j][i]
                        r1 = (1.0 - p.opacity) * r0 + p.opacity * p.red
                        g1 = (1.0 - p.opacity) * g0 + p.opacity * p.green
                        b1 = (1.0 - p.opacity) * b0 + p.opacity * p.blue
                        img[j][i] = [r1, g1, b1]

        cv2.imwrite("images/temp.png", img)
        return img

    def random_person(self):
        num_polys = random.randrange(1,self.max_poly + 1)

        person = [None] * num_polys
        for i in xrange(num_polys):
            poly = Polygon()
            num_points = random.randrange(3, 7)
            for j in xrange(3, num_points):
                poly.add_vertex()

            person[i] = poly

        return person

    def fitness(self, plys):
        img = self.draw(plys)
        f = self.fit.score(img)
        del img

        return f

    def run(self):
        return None

class HillSteppingDriver(Driver):
    def __init__(self, args):
        Driver.__init__(self, args)

    def step(self, polygons, fit):
        newpolygons = copy.deepcopy(polygons)
        mutate(newpolygons)

        while(self.fitness(newpolygons) >= fit):
            newpolygons = copy.deepcopy(polygons)
            mutate(newpolygons)
        return newpolygons

    def run(self):
        polygons = self.random_person()
```

```
        iterations = 0
        while True:
            if self.iterations != None and self.iterations == iterations:
                return polygons

            fit = self.fitness(polygons)
            if(fit < 1):
                return polygons
            polygons = self.step(polygons, fit)
            iterations += 1
            print iterations

def reservoir_sampling(parent, num_genes):
    genes = [None] * min(len(parent), num_genes)

    for i in xrange(len(genes)):
        genes[i] = parent[i]

    for i in xrange(len(genes), len(parent)):
        j = random.randrange(0, i)

        if j <= len(genes):
            genes[j] = parent[i]

    return genes

def create_child(args):
    population, pop_thresholds, num_parents = args

    #cross breed
    parent_indices = set()
    parents = [None] * num_parents
    for i in xrange(num_parents):
        parent = None
        while parent == None:
            r = random.random()
            p = 0
            while r > pop_thresholds[p]:
                p += 1
            if p not in parent_indices:
                parent = p
        parent_indices.add(parent)

    child = []
    j = 0
    for i in parent_indices:
        p = population[i]
        parents[j] = population[i]
        j += 1

    #j = 0
    num_from_parent = sum(map(len, parents)) / len(parents)
    for p in parents:
        child = child + copy.deepcopy(reservoir_sampling(p, num_from_parent))

    #mutate
    if random.random() < THRESH.pop_mutate_poly:
        mutate(child)

    return child

class GeneticAlgorithmDriver(Driver):
    def __init__(self, args):
        Driver.__init__(self, args)
        self.pop_size = args.population

        if args.parents < self.pop_size:
            self.num_parents = args.parents
        else:
            self.num_parents = 2

        self.niche_penalty = abs(args.niche)
```

```
    def evolve(self, population, pop_fitness):
        #niche penalty
        if self.niche_penalty != 0:
            temp = pop_fitness
            for i in xrange(len(pop_fitness)):
                for j in xrange(i + 1, len(pop_fitness)):
                    if abs(temp[i] - temp[j]) < THRESH.niche:
                        pop_fitness[i] = max(0, temp[i] - self.niche_penalty)
                        pop_fitness[j] = max(0, temp[j] - self.niche_penalty)

        total = 0
        for i in pop_fitness:
            total += i

        thresholds = [(1.0 - i / total) for i in pop_fitness]
        for i in xrange(1, len(thresholds)):
            thresholds[i] += thresholds[i - 1]

        children = self.pool.map(create_child, [(population, thresholds, self.num_pa
rents) for i in xrange(self.pop_size)])
        #uncomment line for windows systems, and comment line above
        #children = [create_child((population, thresholds, self.num_parents)) for i
in xrange(self.pop_size)]

        if THRESH.elitism is not None:
            #get (self.elitism * self.pop_size) best parents
            population, probabilities = zip(*sorted(zip(population, pop_fitness), ke
y=lambda p:p[1], reverse=True))

            num_parents = int(THRESH.elitism * self.pop_size)
            lasting_parents = population[:num_parents]
            del population[num_parents:]

            #get ((1 - self.elitism) * self.pop_size) best children
            fitness = [self.fitness(c) for c in children]
            children, fitness = zip(*sorted(zip(children, fitness), key=lambda c:c[1
], reverse=True))
            lasting_children = children[:(self.pop_size - num_parents)]
            del children[(self.pop_size - num_parents):]

            children = lasting_parents + lasting_children
        else:
            del population
        del thresholds
        del pop_fitness

        if random.random() < THRESH.add_random:
            #pick a random child to pop and then replace with random
            index = random.randrange(0, len(children))
            children[index] = self.random_person()

        return children

    def run(self):
        #generate population
        population = [self.random_person() for i in xrange(self.pop_size)]
        iterations = 0
        while True:
            if self.iterations != None and self.iterations == iterations:
                fit = [self.fitness(p) for p in population]
                index = np.argmin(np.array(fit))
                return population[index]

            fit = [self.fitness(p) for p in population]
            if (min(fit) < 1):
                index = np.argmin(np.array(fit))
                return population[index]
            population = self.evolve(population, fit)
            #print "leave crossbreed""
            iterations += 1
            print iterations

def parse_args():
```

```
    parser = argparse.ArgumentParser(description="Genetic Programming: Evolution of
Images from Translucent Polygons")
    parser.add_argument("--algorithm", dest="algo", type=str, choices=["hill", "gene
tic"],
        help="Type of algorithm to use", default="genetic")
    parser.add_argument("--fitness", dest="fitness", default="euc", choices=["euc",
"feat"],
        help="Type of fitness function to use.")
    parser.add_argument("--sample", dest="sample", default=1,
        help="Use just a sample of the image for the fitness function.")

    parser.add_argument("--path", dest="path", type=str, help="Path to image. REQUIR
ED", required=True)
    parser.add_argument("--dest", dest="dest", type=str, help="Path for destination
image", default = None)

    parser.add_argument("--iterations", dest="iterations", type=int, default=None, h
elp="Number of iterations to do.")
    parser.add_argument("--population", dest="population", type=int, default=5, help
="Population size.")
    parser.add_argument("--parents", dest="parents", type=int, default=2, help="Numb
er of parents for cross breeding.")
    parser.add_argument("--niche_penalty", dest="niche", type=float, default=0, help
="Penalty to be applied to niches.")

    parser.add_argument("--thresholds", dest="thresholds", type=str, default=None, h
elp="Path to json file of thresholds.")

    args = parser.parse_args()
    return args

if __name__=="__main__":
    args = parse_args()
    THRESH = Thresholds(args.thresholds, args.population)

    if args.algo == "genetic":
        d = GeneticAlgorithmDriver(args)
    else:
        d = HillSteppingDriver(args)

    polygons = d.run()

    #save image
    dest = args.dest
    if args.dest is None:
        dest = args.path.split(".") + "_result.jpg"
    result = d.draw(polygons)
    cv2.imwrite(dest, result)

    #save polygons
    with file("polygons.poly", "w") as f:
        for poly in polygons:
            s = poly.__str__()
            f.write(s + "\n")
```