Department of Computer Science
Technical University of Cluj-Napoca

Intelligent Systems

Laboratory activity 2018-2019

Project title: OriGAmi
Tool: DEAP

Name: Kelemen Máté
Group: 30432
Email: mate.kelemen009@gmail.com

Assoc. Prof. dr. eng. Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Chapter 1

Getting to know the tool

The tool used for this project is a Python library called DEAP (Distributed Evolutionary Algorithms in Python).
DEAP is a novel evolutionary computation framework for rapid prototyping and testing of ideas. It incorporates the data structures and tools required to implement most common evolutionary computation techniques such as genetic algorithm, genetic programming, evolution strategies, particle swarm optimization, differential evolution, traffic flow and estimation of distribution algorithm. It is developed at Université Laval since 2009.
To use the library, python is needed to be installed on the computer. Installing the DEAP library is like installing any general module for python: type pip install deap on the command terminal.

Chapter 2

Running and understanding the example: OneMax

This algorithm maximizes the sum of a list of integers, each of which can be 0 or 1

```
import random

from deap import base
from deap import creator
from deap import tools

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

toolbox = base.Toolbox()

# Attribute generator
#                      define 'attr_bool' to be an attribute ('gene')
#                      which corresponds to integers sampled uniformly
#                      from the range [0,1] (i.e. 0 or 1 with equal
#                      probability)
toolbox.register("attr_bool", random.randint, 0, 1)

# Structure initializers
#                         define 'individual' to be an individual
#                         consisting of 100 'attr_bool' elements ('genes')
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attr_bool, 100)

# define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list, toolbox.individual)


# the goal ('fitness') function to be maximized
def evalOneMax(individual):
    return sum(individual),
```

```
# ----------
# Operator registration
# ----------
# register the goal / fitness function
toolbox.register("evaluate", evalOneMax)

# register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)

# register a mutation operator with a probability to
# flip each attribute/gene of 0.05
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)

# operator for selecting individuals for breeding the next
# generation: each individual of the current generation
# is replaced by the 'fittest' (best) of three individuals
# drawn randomly from the current generation.
toolbox.register("select", tools.selTournament, tournsize=3)


# ----------

def main():
    random.seed(64)

    # create an initial population of 300 individuals (where
    # each individual is a list of integers)
    pop = toolbox.population(n=300)

    # CXPB  is the probability with which two individuals
    #       are crossed
    #
    # MUTPB is the probability for mutating an individual
    CXPB, MUTPB = 0.5, 0.2

    print("Start of evolution")

    # Evaluate the entire population
    fitnesses = list(map(toolbox.evaluate, pop))
    for ind, fit in zip(pop, fitnesses):
        ind.fitness.values = fit

    print("  Evaluated %i individuals" % len(pop))

    # Extracting all the fitnesses of
    fits = [ind.fitness.values[0] for ind in pop]

    # Variable keeping track of the number of generations
    g = 0
```

```
# Begin the evolution
while max(fits) < 100 and g < 1000:
    # A new generation
    g = g + 1
    print("-- Generation %i --" % g)

    # Select the next generation individuals
    offspring = toolbox.select(pop, len(pop))
    # Clone the selected individuals
    offspring = list(map(toolbox.clone, offspring))

    # Apply crossover and mutation on the offspring
    for child1, child2 in zip(offspring[::2], offspring[1::2]):

        # cross two individuals with probability CXPB
        if random.random() < CXPB:
            toolbox.mate(child1, child2)

            # fitness values of the children
            # must be recalculated later
            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:

        # mutate an individual with probability MUTPB
        if random.random() < MUTPB:
            toolbox.mutate(mutant)
            del mutant.fitness.values

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

    print("  Evaluated %i individuals" % len(invalid_ind))

    # The population is entirely replaced by the offspring
    pop[:] = offspring

    # Gather all the fitnesses in one list and print the stats
    fits = [ind.fitness.values[0] for ind in pop]

    length = len(pop)
    mean = sum(fits) / length
    sum2 = sum(x * x for x in fits)
    std = abs(sum2 / length - mean ** 2) ** 0.5

    print("  Min %s" % min(fits))
```

```
        print("  Max %s" % max(fits))
        print("  Avg %s" % mean)
        print("  Std %s" % std)

    print("-- End of (successful) evolution --")

    best_ind = tools.selBest(pop, 1)[0]
    print("Best individual is %s, %s" % (best_ind, best_ind.fitness.values))


if __name__ == "__main__":
    main()
```

The first thing in this program is defining the types using the *creator* class: fitness and individual. Then comes defining the attribute generator. An individual only has one attribute: *attr_bool*, which can be 0 or 1 with equal probability. Then comes initializing the structures and the operators: the individual and the population, evaluation function, mutate, crossover and selection operators. The algorithm is straightforward: evaluate the initial population, then in the main loop mutate, crossover, reevaluate individuals with invalid fitnesses, replace the population with the offspring and continue until either a fitness of 100 or 1000 generations is reached.

Chapter 3

Project description

3.1 Narrative description

The purpose of this project is to create a tool for designing tree-like origami structures. These structures then can be shaped into anything we want. This will be done by running the first degree optimization of an arrangement of points (described by Robert J Lang in the tree method chapter of Origami Design Secrets [4, 401], namely scale optimization in [4] in the Algorithms chapter). The program takes as input a weighted tree stored in a file, and outputs an arrangement of points with coordinates between 0 and 1 which correspond to tips of the flaps on the unfolded paper 3.3. These points then, can be mapped onto a real paper, and by using geometric techniques, the crease pattern that is needed to fold the base, can be constructed 3.2. The character of this project is purely experimental, since folded origami models do not solve any real world problem, instead they point to how paper-like objects (eg. thin sheet of steel) can be manipulated to optimize space requirements of many mechanical objects (eg. how solar panels are folded when sent into space inside satellites).

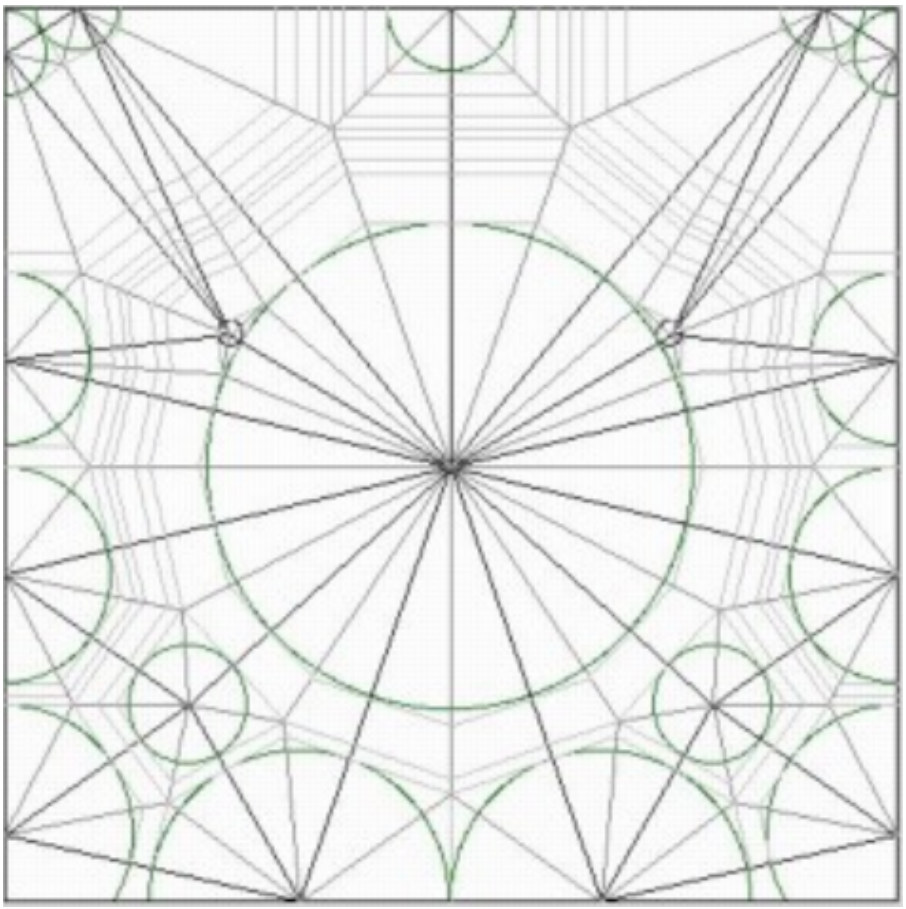
3.2 About Origami Design

Origami is the art of paperfolding. When paper is folded, the line that is created by unfolding the paper is called a crease. Collectively, all the creases on the paper form the crease pattern of the model. For each crease pattern there is a folded model. Scientists deduced that there are levels of abstractions that we can employ to design origami models [4]. Example of a crease pattern, base and folded model can be seen on figure 3.1

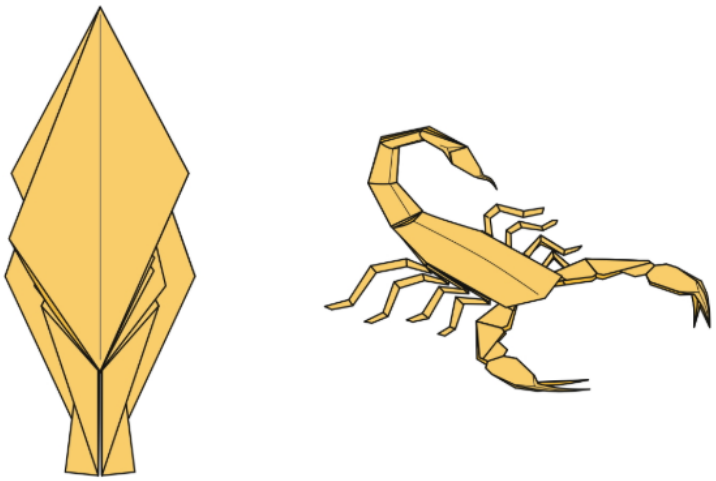
3.3 Tree theory

One form of abstraction is called the tree graph of the model, described by Robert J. Lang 3.1. The tree graph looks like a stick figure, and it captures the number of flaps, their lengths and how they are connected to each other. In an ideal solution, the proportions on the tree graph are equal to the proportions on the paper. When in this form, the folded model is called the base, since we only constructed the skeleton for a concrete model (horse, dragon, etc.). Tree graphs are used to generate uniaxial bases, that is, every flap of the model will lie along an axis. Another interpretation of the tree graph is that it is the shadow cast by the folded model in a plane perpendicular to the layers of paper of the base. The tree theorem says that leaf vertexes in the tree graph will become tips of the flaps. There are 3 types of creases in origami: axial creases, hinge creases and ridge creases. The most important of them are axial creases, these are the creases that lie along the axis of the base, and so, these decide the proportions of

Figure 3.1: Scorpion designed by Robert J Lang.

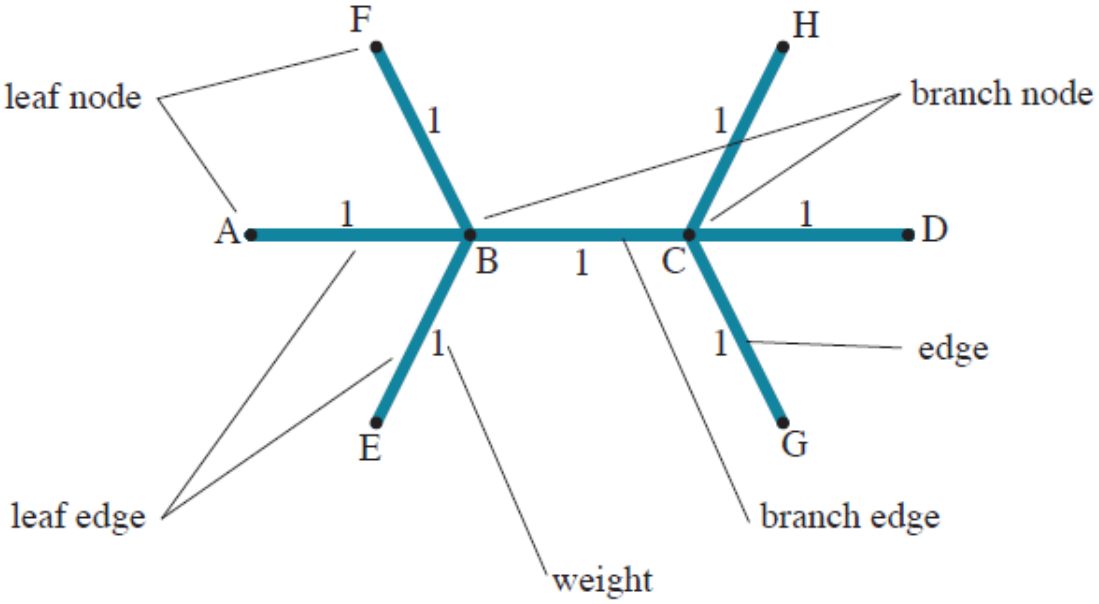


(a) Crease pattern



(b) Base and folded model

Figure 3.2: An example tree graph with its' components 3.3



the model, and how flaps are connected to one another. Example tree graph: 3.2. Ridge and hinge creases can be constructed geometrically based on axial creases. See [4] chapters 2, 9, 10, 11.

3.4 The Problem

The question now arises: how can we design a model systematically, by "converting" the tree graph into a base of an origami model? The answer is not so simple. To do this by hand, we have to find an arrangement of points on the paper by trial and error, taking into account the proportions that we want.
Computational origami design is what changes this intuitive approach to a systematic one [1]. The Tree method 3.3 described by Robert J Lang 3.1 breaks down the base further into paths and vertexes. Any leaf vertex of the tree graph will correspond to a point on the paper. The problem is, this mapping is not one-to-one. A leaf vertex on the tree graph could map to any point on the paper. According to Robert J. Lang 3.1, we can enforce mathemathical constraints, such that a feasible arrangement of points results, that when, assigned creases, yields our base with the proportions that we have given in the tree graph, with a computable optimality. Then the problem becomes a constraint based optimization problem. This is how Genetic Algorithms and AI comes into play: we start from a completely random arrangement of points, and slowly head towards optimality, from chaos to order. The program will incrementally make the crease pattern more and more efficient.

3.5 Related Work

The most important work related to this project is Robert J. Lang's treemaker [3], which is a software that uses nonlinear optimization techniques to produce an origami base from a tree graph. The software is much more complex than a simple parameter optimization and it

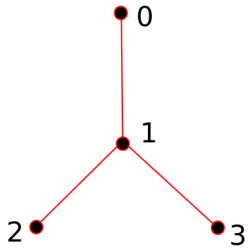


Figure 3.3: Example tree graph with all edges of unit length

implements many other features of origami design. The problem with this software is, that it is outdated, and the optimizers often get stuck. There are other softwares that do not use the tree method and the concept of uniaxial bases, see 6.3.

3.6 Assumptions

It is implicitly assumed, that the input file is valid. That is, the tree graph is an undirected, connected, weighted tree.

3.7 Input

The input file is a weighted connected tree, which is inputted as follows: The contents of the input file are lines that represent edges and weights. That is, the first two parameters are the two vertexes that form the edge, and the third is the weight of the edge. Each edge is written only once, since this is an undirected graph. The vertexes represent indexes, thus, they are inputted as integers, and the weight is a floating point number.

Example 1 (Input for tree graph on figure 3.3) *:*

```
0 1 1
1 2 1
1 3 1
```

3.8 Output

The output of the program will be written on the console. The output will contain the entire timeline of evolution: what is the mean, average, standard deviation and the best and worst of scales of each model, for each generation. The last line contains the best generated crease pattern: its points' pairwise x, y coordinates in a vector of floating point numbers, and its scale in parantheses.

Example 2 (Sample output for given input) *:*

```
...
Generation 200:
Evaluated 864 individuals
Min: 0.0
```

```
Max: 0.5172984684282184
Avg: 0.46344746621237687
Standard deviation: 0.1539550512420958
END OF EVOLUTION
Best individual is [3.755904039425214e-17, 0.7328117365571286, 0.9998186145173935,
 0.9995745682957765, 0.7303271736125168, 5.0490618469532284e-17]
(0.5172984684282184,)
```

Note: really small numbers can be considered to be 0.

Chapter 4

Implementation

4.1 Mathematical Background

To elaborate further on Tree theory, in order to arrive to an efficient crease pattern, there is one more factor to consider: the scale of the crease pattern
The scale represents the relationship between the size of the tree graph and the and the crease pattern on the square. The scale is simply the distance on the square that corresponds to one unit in the tree graph. For every path between leaf vertices u_i and u_j, the leaf vertexes must satisfy the inequality
$|u_i - u_j| \leq ml_{ij}$
for a scale factor m (l_{ij} is the length of path between vertexes i, j in the tree graph). The set of all such equations are called the *path conditions* for the given tree graph.
Thus, the two families of constraints that must be satisfied for any valid crease pattern are:

1. The coordinates of every vertex must lie within the square
2. The separation between any two vertices on the square must be at least as large as the scaled length of the path between their corresponding two nodes as measured along the tree

Throughout the entire evolution, the goal is to make the scale of the circle pattern as large as possible, while taking into account the given constraints. See [4] chapter Algorithms

4.2 Structure of the project

The project is structured in 3 files:

1. *main.py* contains the main loop of the evolution. Here the operators, the functions, the population, the evolution strategies are instantiated, and then the main evolution is ran, and lastly the best individual is printed out on the console
2. *eval.py* contains the functions required to compute the scale, the mutation, evaluation of the fitness of an individual, and to constrain the points to lie on the paper
3. *graph.py* contains the tree graph processing code

4.3 Implementation decisions

Why genetic algorithms?
We are facing a constraint based optimization problem. The scale of the crease pattern is

needed to be optimized, and one strategy is to fix the scale at the beginning, in each generation vary the points' coordinates only, and then recalculate the scale only in case of a crossover or a mutation. The purpose of genetic algorithms is to start off from complete randomness, and arrive to order.
To fix the scale at the beginning, another path constraint is required: at least one *active path*. An active path is guaranteed to be the shortest path between two points, and thus, it is equal to an *axial crease*. This condition is satisfied, when $|u_i - u_j| = ml_{ij}$.
That is, the separation between given u_i and u_j is exactly equal to the scaled length of the path between the corresponding two nodes. This becomes the third and last path condition of the crease pattern. From this, we can deduce an equation for a scale: $m = |u_i - u_j|/l_{ij}$ and, in fact, this is how the initial population is evaluated. Mathemathical theory is described is [4] chapter Algorithms, and [3].

4.4 The Individual

An individual in this project is the crease pattern itself, that is, the attributes that it has, are the coordonates of each of its leaf vertexes. Its fitness function is ultimately the scale of the crease pattern. The attributes of the individual are expressed as a *list*. This list and the input tree graph is what decides the fitness of the individual.

4.5 Evaluation function

The first path condition [4] states that every point must be on the paper, that is, every coordinate of the individual must be between 0 and 1. This is easily achieved with the *constrain-space()* function, and with *toolbox.decorate()*. After a mutation if, by chance, any coordinate falls out of the interval (0, 1), we consider it invalid. We do not need to use a *constrain-scale()* function, as long as the computed scale that we return from this function satisfies the path conditions. The solution here was, to compute a scale for each leaf vertex, and take the minimum of them. Since wee took the minimum, any distance on the paper between two leaf vertexes is either equal to or greater than their corresponding scaled length in the tree graph and thus the second condition is always satisfied.

4.6 Mutation function

The mutation function allows an attribute to not get stuck on a certain value. In the *mutate()* function, we give each attribute (coordinate) a chance to move a tiny bit, to the left, to the right, to the top, to the bottom, using a gaussian distribution with a sigma of 1.

4.7 Main Algorithm

The main algorithm of the genetic evolution is the following:

1. Accept as input the number of generations to be evaluated, the tree graph, the two points that form the active path, calculate number of leaf vertexes
2. Define individual 4.4, fitness function 4.5, mutation 4.6, crossover functions, selection strategy
3. Instantiate population of given number of individuals

4. Define crossover and mutation probabilities
5. Evaluate population
6. While generation count is less than the input number of generations, do:
   (a) Select k best individuals from 50 turns
   (b) Clone current population into offspring
   (c) Crossover two consecutive individuals with given probability
   (d) Mutate an individual with given probability
   (e) Reevaluate invalid fitnesses
   (f) Replace current population with offspring
   (g) Print out the worst, the best, the mean, the average and the standard deviation of the fitnesses from the current generation
7. Print out the best individual with its fitness from the last generation

Chapter 5

Experiments and Results

5.1 Efficiency

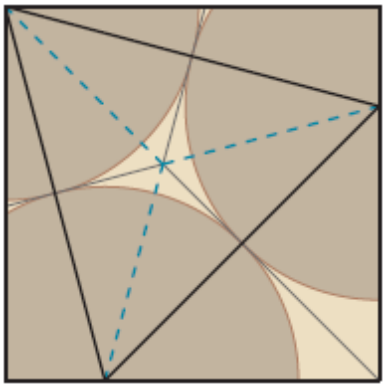
How efficient is a generated crease pattern?
It is proven mathematically, that the most efficient crease pattern for the tree graph on 3.3 is the one on figure 5.1, with a scale of *0.518* [4], ch. 8. We can observe, that after mere 5 generations, the best scale already reaches 0.5, and the average scale reaches 0.466 in about 50 generations, with a crossover probability of 50%, and a mutation probability of 70%. At generation 200, the value of the best scale is 0.5173, which is very close to the global optimum. If we observe the coordinates of the best individual, we can see that is an expected result (Note that we can rotate the paper).

```
Best individual is [0.2671032936490236, 5.4413895288530505e-17,
1.0701361701936522e-18, 0.9996385840306898,
0.9995837191672834, 0.7308238965204266] (0.5173541988234558,)
```

5.2 Fitness growth

A phenomenon that we can observe, it that the values of the best and average scales increase very suddenly at the beginning, and their growth slow down exponentially, as we approach the global optimum. The values of the scales seem to follow a logarithmic curve with increasing generations. Another observation is, that for a low mutation probability, it takes more generations to reach the same value that we would reach with a higher mutation probability. For a mutation probability with 30%, it takes more generations (more than 300) to arrive to the same best scale value than for a mutation probability with 70%. This is due to the fact that

Figure 5.1: The optimal crease pattern for the tree graph on 3.3



the mutation probability impacts greatly whether the best individual is even changed. Results show that for a mutation probability of 30%, the scale of a model stabilizes more slowly, in about 130 generations, while for 70%, this is reached in about 80 generations. Sudden growth can still occur in the later generations, however, due to increased mutation probability. Thus, we can conclude, that the greater the mutation probability, the faster we reach a value close to the global optimum. Results show that manipulating the crossover probability also impacts the scale value and growth. For high probabilities the scale stabilizes quicker. This could be a trap, however, because if the crossover probabily chance is high, we have a higher chance to be stuck at some lower value, because individuals will be similar to one another, and it is more difficult to step out of this value by mutation. The conclusion is, that for reliability, it is best to fix the mutation at 50% - 70%, and keep the crossover probability at 20% - 30%, and after 200 generations, we will arrive to a value that is less than 0.1% off from the global optimum.

Chapter 6

Related work

6.1 TreeMaker

The most important related work to this project is Robert J. Lang's *treemaker* [3], as the aim of this project is the same as treemaker's: to provide a "design tool for creating tree-like origami structures". The main backdraw of treemaker is that it uses outdated nonlinear optimizers, that can often get stuck at a suboptimal crease pattern. In contrast to this project, however, treemaker uses not just *scale optimization*. In addition, it can perform *edge optimization* and *strain optimization*, and we can enforce other path constraints, such as fixed angle, fixed distance between two points, creases being parallel, etc. This project is to be considered as contemporary mirror, using genetic algorithms instead of nonlinear optimizers, being a more robust, but a reduced form of treemaker [4].

6.2 Advantages and limitations

To reiterate, the main advantage of this project in contrast to treemaker is that the optimization doesn't get stuck [3], and it is easy to use (treemaker has many complex features and GUI, that makes a first-time user confused, and they can get lost easily). The most visible limitation of this program is, that it only does the groundwork for an origami designer, as there are many more steps to reach a concreate model from a mere arrangement of points. It does not provide a GUI to show the generated axial creases, just values, that when designing origami, are extremely difficult to draw on a physical paper. The main disadvantage of the program is that it does not cover wholly the concepts described in tree theory. The disadvantage of tree theory is that even though it can algorithmically produce the optimal crease pattern, the pattern is often not symmetrical, and the reference points are really hard to find.

6.3 Other related design tools

- ReferenceFinder Online - for finding reference points by folding
- Tomohiro Tachi's Origamizer - for generating crease patterns that produce triangulated 3D meshes
- Alex Bateman's Tessellations - includes extensive information about origami tessellations and a downloadable program, Tess, for generating them.
- JOrigami - An open source Java implementation of the "Fold and Cut" problem originally solved by Erik Demaine [2]. Contains code and some good references on the problem.

6.4 Possible extensions

First and foremost, the first extension would be to output the axial creases. This is pretty simple to implement, since there is only one equation to be satisfied, and less groundwork is needed from the user. The second extension of the program would be to make the generated crease pattern more customizable, that is, employ edge optimization and strain optimizations, as well as the path constraints described in the algorithms chapter of emphOrigami Design Secrets.

Appendix A

Your original code

This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained. Including in this section any line of code taken from someone else leads to failure of IS class this year. Failing or forgetting to add your code in this appendix leads to grade 1. Don't remove the above lines.

A.1 main.py

```
import Graph
import eval
import random as rand
from deap import tools
from deap import base
from deap import creator
import sys

if __name__ == '__main__':
    input_file = str(sys.argv[1])
    p1_index = int(sys.argv[2])
    p2_index = int(sys.argv[3])

    SIZE = int(sys.argv[4])
    GEN_CNT = int(sys.argv[5])

    g = Graph.Graph(SIZE, input_file)
    p1 = Graph.Point(p1_index, rand.random(), rand.random())
    p2 = Graph.Point(p2_index, rand.random(), rand.random())

    active_path = g.get_path(p1.index, p2.index)
    LEAF_SIZE = len(g.leaf_paths())

    creator.create("FitnessMin", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMin)

    toolbox = base.Toolbox()
    toolbox.register("rand_coord", rand.random)
    toolbox.register("individual", tools.initRepeat,
```

```
creator.Individual, toolbox.rand_coord, n=LEAF_SIZE * 2)
toolbox.register("evaluate", eval.evaluate, graph=g)
toolbox.decorate("evaluate",
tools.DeltaPenality(eval.constrain_space, 0))
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", eval.mutate, sigma=1, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=50)


pop = toolbox.population(n=1000)
crossover_prob = 0.3
mut_prob = 0.7

print("Start of evolution")

fitnesses = list(map(toolbox.evaluate, pop))

gen = 0

while gen < GEN_CNT:
    gen = gen + 1

    print("Generation %i:" % gen)

    offspring = toolbox.select(pop, len(pop))
    offspring = list(map(toolbox.clone, offspring))

    #apply crossover and mutation
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        if rand.random() < crossover_prob:
            toolbox.mate(child1, child2)

            del child1.fitness.values
            del child2.fitness.values

    for mutant in offspring:

        if rand.random() < mut_prob:
            toolbox.mutate(mutant)
            del mutant.fitness.values

    #reevaluate invalid indexes
    invalid_ind = [ind for ind in offspring if not ind.fitness.valid]

    fitnesses = map(toolbox.evaluate, invalid_ind)

    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit
```

```
        print("Evaluated %i individuals" % len(invalid_ind))

        pop[:] = offspring

        fits = [ind.fitness.values[0] for ind in pop]

        length = len(pop)
        mean = sum(fits) / length
        sum2 = sum(x * x for x in fits)
        std = abs(sum2 / length - mean ** 2) ** 0.5

        print("Min: %s" % min(fits))
        print("Max: %s" % max(fits))
        print("Avg: %s" % mean)
        print("Standard deviation: %s" % std)

    print("END OF EVOLUTION")

    best = tools.selBest(pop, 1)[0]

    print("Best individual is %s %s" % (best, best.fitness.values))
```

A.2 eval.py

```
import math
from Graph import *
import random


def calc_scale(graph, path, u, v):
    distance = graph.dist(path[0], path[-1])
    point_dist = math.sqrt(((u.x - v.x) * (u.x - v.x)) + \
                           ((u.y - v.y) * (u.y - v.y)))

    return point_dist / distance


def constrain_space(individual):
    for p in individual:
        if p < 0 or p > 1:
            return False

    return True


def mutate(individual, sigma, indpb):
    for i in range(len(individual)):
        if random.random() < indpb:
            individual[i] = random.gauss(individual[i],
```

```
                                    sigma * individual[i])

    return individual,


def evaluate(individual, graph):
    scale = 999

    #calculate scale of all leaf paths in the graph and select minimum

    leaf_vertexes = graph.leaf_vertices()

    points = list(zip(individual[::2], individual[1:][::2]))

    for i in range(len(leaf_vertexes)):
        #1 2 -> u v
        #path accepts indexes

        u = Point(leaf_vertexes[i], points[i][0], points[i][1])

        for j in range(i + 1, len(leaf_vertexes)):

            v = Point(leaf_vertexes[j], points[j][0], points[j][1])

            path = graph.get_path(u.index, v.index)
            temp_scale = calc_scale(graph, path, u, v)

            if temp_scale < scale:
                scale = temp_scale

    return scale,
```

A.3 graph.py

```
import numpy as np


class Graph:
    def __init__(self, vertices, file):
        self.edges = [[0 for i in range(vertices)] for j in range(vertices)]
        self.vertex_cnt = vertices

        self.read_tree(file)

    def read_tree(self, file):
        with open(file) as f:
            for line in f:
                pair = line.split(' ')
                try:
                    self.edges[int(pair[0])][int(pair[1])] = int(pair[2])
```

```
                self.edges[int(pair[1])][int(pair[0])] = int(pair[2])
            except ValueError:
                print("File error")
    self.edges = np.array(self.edges, dtype=np.int16)

def get_edge_length(self, u, v):
    return self.edges[u][v]

def leaf_vertices(self):
    leaves = []
    for i, j in enumerate(self.edges):
        temp = np.array(j, dtype=np.int16)
        if np.count_nonzero(temp) == 1:
            leaves.append(i)

    return leaves

def bfs(self, start):
    q = [start]
    visited = set()
    prev = [-1 for i in range(self.vertex_cnt)]

    while q:
        current = q.pop(0)
        if current not in visited:
            visited.add(current)

            adj_list = self.adj(current)

            for i in adj_list:
                if i not in visited:
                    prev[i] = current

            q.extend(adj_list)

    return prev

def dist(self, v, u):
    bfs_list = self.bfs(v)
    dist = 0

    current_vertex = u
    prev_vertex = bfs_list[u]

    while current_vertex != -1:
        dist += self.edges[prev_vertex][current_vertex]
        #step up one time
        current_vertex = prev_vertex
        prev_vertex = bfs_list[prev_vertex]
```

```
        return dist

    def candidate_axial_creases(self):
        axial_list = []

        vertex_list = self.leaf_vertices()

        for i in vertex_list:
            #investigate each pair

            for j in vertex_list:
                if i < j:
                    axial_list.append((i, j, self.dist(i, j)))

        return axial_list

    def adj(self, vertex):
        adj_list = []
        for i, j in enumerate(self.edges[vertex]):
            if j != 0:
                adj_list.append(i)

        return adj_list

    def get_path(self, u, v):
        bfs_list = self.bfs(u)
        current_vertex = v
        prev_vertex = bfs_list[v]
        path = []

        while current_vertex != u:
            path.append(current_vertex)
            current_vertex = prev_vertex
            prev_vertex = bfs_list[prev_vertex]

        path.append(u)

        return path

    def leaf_paths(self):
        paths = []

        leaves = self.leaf_vertices()

        for i in leaves:
            for j in leaves:
                if i < j:
                    paths.append(self.get_path(i, j))

        return paths
```

```
    def __repr__(self):
        return self.edges.__repr__()


class Point:
    def __init__(self, index, x, y):
        self.index = index
        self.x = x
        self.y = y

    def __repr__(self):
        string = "{}: {}, {}"
        return string.format(self.index, self.x, self.y)
```

Appendix B

Quick technical guide for running your project

Requirements:
To run the program, python and the DEAP library have to be installed.

1. Download and place the 3 python files (*main.py*, *eval.py*, *graph.py*) into the same folder
2. Open a console terminal and type:

   ```
   > python main.py {input_file} {p1_index} {p2_index} {tree_node_count} {gen_count}
   ```

   Where:

- *input file* - the relative path to the input file that contains the tree
- *p1_index* - *p2_index* - the required active path between two nodes in the tree
- *tree_node_count* - number of nodes in the tree
- *gen_count* - number of generation to be evaluated

Bibliography

[1] Erik D Demaine and Martin L Demaine. Recent results in computational origami. In *Origami3: Third International Meeting of Origami Science, Mathematics and Education*, pages 3–16, 2002.

[2] Erik D Demaine, Martin L Demaine, and Anna Lubiw. Folding and cutting paper. In *Japanese Conference on Discrete and Computational Geometry*, pages 104–118. Springer, 1998.

[3] Robert J Lang. Tree maker. *http://www. langorigami. com/science/treemaker/treemaker5. php4*, 2006.

[4] Robert J Lang. *Origami design secrets: mathematical methods for an ancient art*. AK Peters/CRC Press, 2011.