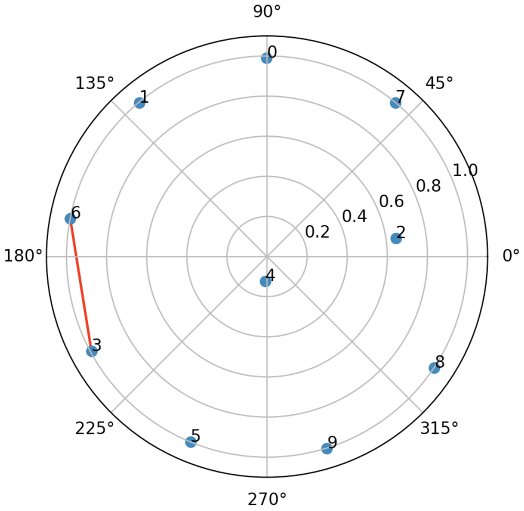
CS 224 Introduction to Python Spring 2022

Programming Assignment 3

University of Wisconsin – La Crosse *Due: April 25*



The Point Scattering Problem: In the circle packing problem, one attempts to arrange, without overlapping, some number n of circles within another geometric shape, typically a rectangle or a larger circle. If the n circles are the same size, the problem is known as congruent circle packing. A typical instance of this problem consists of a square s with side length r_1 and integer n. The answer is the largest r_2 such that n circles with radius r_2 can be placed in s with no overlap. (An equivalent formulation is: given n and r_2, find the smallest r_1.) Circle packing in a circle is related but more difficult. Though congruent circle packing in a circle is the subject of much research in the mathematics community, optimal results remain unproven for values of n as small as 14.

We consider an equivalent variation of congruent circle packing in a circle, sometimes referred to as the *point scattering problem*. An instance of this problem consists of a unit circle c and an integer n that represents a number of points to arrange in c. The goal is to maximize the minimum distance between any two of the n points. This. problem is easy to state and understand but requires significant computation even for small instances. The figure above shows a point scattering instance for $n = 10$. Interestingly, instance difficulty does not increase monotonically with number of points. For example, an optimal solution has been proven for $n = 19$ but not for any $n \in [14..18]$ or greater than 19, though conjectures exist.

Genetic Algorithms: Genetic algorithms are population-based approximation algorithms that excel at finding good solutions to computationally expensive problems using rules based on the principles of Darwinian evolution. Key among these is survival of the fittest. In nature, survival of the fittest rewards faster zebras while those that are slower are eaten by lions. In a genetic algorithm, this takes the form of a *fitness function* that allows the algorithm to evaluate and compare solutions to the problem. Better solutions survive while worse ones are eliminated. Another important aspect of using a GA to solve a problem is finding a way to encode a solution to the problem in an effective and efficient way. This is known as a representation. Though there are many variations, the basic structure of a genetic algorithm is:

1. Create a "population" of size n of randomly generated candidate solutions
2. Do the following in each of a pre-defined number of generations:
   (a) Create offspring by combining parts of the representations from two existing solutions
   (b) Randomly make small mutations in the offspring
   (c) Evaluate the offspring using the fitness function
   (d) Combine the offspring with the existing population
   (e) Choose the best n members for survival (the others are eaten by lions)
3. Choose the best solution based on fitness

The Assignment: I have provided a genetic algorithm for the point scattering problem, implemented in `point_scatter.py`. Look through the code in that file. Pay particular attention to the code for the GA which is in the function named `evolve`. You `must not` edit the code in `point_scatter.py`. The GA depends on three classes that you will write: `Population`, `Individual`, `Point`. You must write these in files named `population.py`, `individual.py`, and `point.py`, respectively.

`The Point Class:` Points are in polar coordinates. The attributes are:

- `radius:` Distance from the origin. Maximum values is 1.
- `theta:` Angle in radians.

You must provide the following methods but may include others as needed:

- `__init__:` Sets attributes to random values.
- `__str__:` Returns a readable string representation of a `Point` object.
- `mutate:` Mutates a point by adding small *deltas* to the attributes of the `Point`. The delta values should be positive or negative with equal probability. Use the `R_FACTOR` and `T_FACTOR` constants from `constants.py` to scale the values. For each attribute, generate a random value within the full range for that attribute and then divide by the appropriate factor. You are free to

explore different values for the factor constants. Ensure that `radius` remains in range.

`The Individual Class:` An Individual represents a solution to the problem and consists of a number of points, where the number is determined by the global constant `NUM_POINTS`. The attributes are:

- `points:` A list of `Point` objects.
- `fitness:` The current fitness value of the individual.

You must provide the following methods but may include others as needed:

- `__init__:` Creates a list of points and sets `fitness` to -1.
- `__str__:` Creates a readable string representation of the points in an `Individual` object.
- `distance:` Returns the distance between two points in the individual. The points are parameters to the method. Note that the distance formula for polar coordinates is not the same as that for Cartesian coordinates.
- `evaluate:` This is the fitness function. It returns the minimum distance between any two points in the individual. Recall that our goal in the point scattering problem is to maximize that minimum value.
- `min_dist:` Similar to `evaluate` except that it returns the minimum distance and the indices of the two points that generated the minimum distance. This is used to manually verify results and for the plot generated at the end of the GA run.
- `crossover:` Performs crossover on two individuals (the one to which the method was applied and a second that is passed as a parameter). It takes a second parameter called `indpb` with default value 0.75. We are implementing called *uniform crossover*. Consider index `i` into the lists of points for the two individuals. With probability $1-$ `indpb`, the two points at index `i` are exchanged. This is done independently for each valid index.
- `mutate:` Mutates an individual by determining independently whether to mutate each point with probability `indpb`, where that is a parameter with default value 0.7. Calls the `Point mutate` method.

`The Population Class:` The population consists of a number of individuals, where the number is specified by a global constant in `constants.py`. This class contains a single attribute:

- `pop:` A list of `Individual` objects.

You must provide the following methods but may include others as needed:

- `__init__:` Takes an option parameter, representing a list of individuals, with default value of the empty list. If the list is not empty, it is assigned to the

n	Best-known	n	Best-known
2	2.0*	12	0.66015273*
3	1.73205080*	13	0.61803398*
4	1.41421356*	14	0.60088416
5	1.17557050*	15	0.56796286
6	1.0*	16	0.55318521
7	1.0*	17	0.52742146
8	0.86776747*	18	0.51763809
9	0.76536686*	19	0.51763809*
10	0.71097823*	20	. 0.48516360

Table 1: Maximized minimum distances for specified number of points in the unit circle. The Best-known column indicates the best known distances, where the * indicates the distance has been proven to be optimal.

`pop` attribute. Otherwise, the attribute is populated with randomly generated `Individual` objects.

- `__str__`: Returns a readable string representation of the individuals in the population.
- `evaluate`: Evaluates every individual in the population.
- `shuffle`: Randomly permutes the population.
- `sort`: Sorts the `pop` attribute of the `Population` object. Takes parameters `key` and `reverse`.

Miscellaneous Details: Look at the contents of `constants.py` to familiarize yourself with the values there. Once your code works, you will want to change some of the to experiment. For example, you will want to try different values for `NUM_POINTS`. You will also want to perform runs with different numbers of generations, `GENS`. In general, longer runs provide better results, though the law of diminishing returns kicks in at some point. Runs of 20000 generations are certainly reasonable, though for testing you will want to use a much smaller value, such as 200.

Table 1 provides optimal or conjectured optimal results for 2 to 20 points. Note that larger values of `NUM_POINTS` require larger numbers of generations and that as the number of points increases the time required per generation also increases.

Submission: Information on submitting your program appears below.

Adhere to common coding conventions and **comment your code.**. Include a comment at the top of each Python file that looks like this:

```
#
# CS 224 Fall 2021
# Programming Assignment 3
#
# Description of what the code in this file does
#
# Author: Your name here
# Date: Month XX, 2021
#
```

Name your code files as indicated above. All code files, including `point_scatter.py` and `constants.py`, must be in a directory named `Lastname-Prog03` where `Lastname` is replaced with your last name. You must zip your directory prior to submission. Submit your solution via Canvas by 11:59 PM on the due date.