inspyred Documentation

Release 1.0

Aaron Garrett

April 03, 2012

CONTENTS

CHAPTER

ONE

OVERVIEW

This chapter presents an overview of the inspyred library.

1.1 Bio-inspired Computation

Biologically-inspired computation encompasses a broad range of algorithms including evolutionary computation, swarm intelligence, and neural networks. These concepts are sometimes grouped together under a similar umbrella term – "computational intelligence" – which is a subfield of artificial intelligence. The common theme among all such algorithms is a decentralized, bottom-up approach which often leads to emergent properties or behaviors.

1.2 Design Methodology

The inspyred library grew out of insights from Ken de Jong's book "Evolutionary Computation: A Unified Approach." The goal of the library is to separate problem-specific computation from algorithm-specific computation. Any bio-inspired algorithm has at least two aspects that are entirely problem-specific: what solutions to the problem look like and how such solutions are evaluated. These components will certainly change from problem to problem. For instance, a problem dealing with optimizing the volume of a box might represent solutions as a three-element list of real values for the length, width, and height, respectively. In contrast, a problem dealing with optimizing a set of rules for escaping a maze might represent solutions as a list of pair of elements, where each pair contains the two-dimensional neighborhood and the action to take in such a case.

On the other hand, there are algorithm-specific components that may make no (or only modest) assumptions about the type of solutions upon which they operate. These components include the mechanism by which parents are selected, the way offspring are generated, and the way individuals are replaced in succeeding generations. For example, the ever-popular tournament selection scheme makes no assumptions whatsoever about the type of solutions it is selecting. The *n*-point crossover operator, on the other hand, does make an assumption that the solutions will be linear lists that can be "sliced up," but it makes no assumptions about the contents of such lists. They could be lists of numbers, strings, other lists, or something even more exotic.

The central design principle for inspyred is to separate problem-specific components from algorithm-specific components in a clean way so as to make algorithms as general as possible across a range of different problems.

For instance, the inspyred library views evolutionary computations as being composed of the following parts:

- Problem-specific components
  - A generator that defines how solutions are created
  - An evaluator that defines how fitness values are calculated for solutions
- Algorithm-specific evolutionary operators

  - An observer that defines how the user can monitor the state of the evolution
  - A terminator that determines whether the evolution should end
  - A selector that determines which individuals should become parents
  - A variator that determines how offspring are created from existing individuals
  - A replacer that determines which individuals should survive into the next generation
  - A migrator that defines how solutions are transferred among differnt populations
  - An archiver that defines how existing solutions are stored outside of the current population

Each of these components is specified by a function (or function-like) callback that the user can supply. The general flow of the `evolve` method in inspyred is as follows, where user-supplied callback functions are in ALL-CAPS:

```
Create the initial population using the specified candidate seeds and the GENERATOR
Evaluate the initial population using the EVALUATOR
Set the number of evaluations to the size of the initial population
Set the number of generations to 0
Call the OBSERVER on the initial population
While the TERMINATOR is not true Loop
    Choose parents via the SELECTOR
    Initialize offspring to the parents
    For each VARIATOR Loop
        Set offspring to the output of the VARIATOR on the offspring
    Evaluate offspring using the EVALUATOR
    Update the number of evaluations
    Replace individuals in the current population using the REPLACER
    Migrate individuals in the current population using the MIGRATOR
    Archive individuals in the current population using the ARCHIVER
    Increment the number of generations
    Call the OBSERVER on the current population
```

The observer, terminator, and variator callbacks may be lists or tuples of functions, rather than just a single function. In each case, the functions are called sequentially in the order listed. Unlike the other two, however, the variator behaves like a pipeline, where the output from one call is used as the input for the subsequent call.

1.3 Installation

The easiest way to install inspyred is to use pip as follows:

```
pip install inspyred
```

The Python Package Index page for inspyred is http://pypi.python.org/pypi/inspyred.

The source code git repository can be found at https://github.com/inspyred/inspyred.

1.4 Getting Help

Any questions about the library and its use can be posted to the inspyred Google group at https://groups.google.com/forum/#!forum/inspyredhelp. If a forum posting is not appropriate or desired, questions can also be emailed directly to aaron.lee.garrett@gmail.com. Feedback is always appreciated. Please let us know how you're using the library and any ideas you might have for enhancements.

CHAPTER

TWO

TUTORIAL

This chapter presents three examples to which inspyred can be applied.

2.1 The Rastrigin Function

The Rastrigin function is a well-known benchmark in the optimization literature. It is defined as follows:

Minimize

$$10n + \sum_{i=1}^{n} \left((x_i - 1)^2 - 10\cos(2\pi(x_i - 1)) \right)$$

for $x_i \in [-5.12, 5.12]$.

Since this problem is defined on a set of continuous-valued variables, using an evolution strategy as our optimizer seems appropriate. However, as always, we'll need to first create the *generator* and the *evaluator* for the candidate solutions. First, the generator...

2.1.1 The Generator

```
from random import Random
from time import time
from math import cos
from math import pi
from inspyred import ec
from inspyred.ec import terminators

def generate_rastrigin(random, args):
    size = args.get('num_inputs', 10)
    return [random.uniform(-5.12, 5.12) for i in range(size)]
```

First, we import all the necessary libraries. `random` and `time` are needed for the random number generation; `math` is needed for the evaluation function; and `inspyred` is, of course, needed for the evolutionary computation.

This function must take the random number generator object along with the keyword arguments. Notice that we can use the `args` variable to pass anything we like to our functions. There is nothing special about the `num_inputs` key. But, as we'll see, we can pass in that value as a keyword argument to the `evolve` method of our evolution strategy.

This code is pretty straightforward. We're simply generating a list of `num_inputs` uniform random values between -5.12 and 5.12. If `num_inputs` has not been specified, then we will default to generating 10 values.

And now we can tackle the evaluator...

2.1.2 The Evaluator

```
def evaluate_rastrigin(candidates, args):
    fitness = []
    for cs in candidates:
        fit = 10 * len(cs) + sum([((x - 1)**2 - 10 * cos(2 * pi * (x - 1))) for x in cs])
        fitness.append(fit)
    return fitness
```

This function takes an iterable object containing the candidates along with the keyword arguments. The function should perform the evaluation of each of the candidates and return an iterable object containing each fitness value in the same order as the candidates [1]. The Rastrigin problem is one of minimization, so we'll need to tell the evolution strategy that we are minimizing (by using `maximize=False` in the call to `evolve`).

2.1.3 The Evolutionary Computation

Now that we have decided upon our generator and evaluator, we can create the EC. In this case since our problem is real-coded, we'll choose a evolution strategy (ES) [2]. The default for an ES is to select the entire population, use each to produce a child via Gaussian mutation, and then use "plus" replacement.

```
rand = Random()
rand.seed(int(time()))
es = ec.ES(rand)
es.terminator = terminators.evaluation_termination
final_pop = es.evolve(generator=generate_rastrigin,
                      evaluator=evaluate_rastrigin,
                      pop_size=100,
                      maximize=False,
                      bounder=ec.Bounder(-5.12, 5.12),
                      max_evaluations=20000,
                      mutation_rate=0.25,
                      num_inputs=3)
# Sort and print the best individual, who will be at index 0.
final_pop.sort(reverse=True)
print(final_pop[0])
```

```
$ python rastrigin.py
[2.038194819402378, 0.9984300139314, 1.9654139554832206, 2.2880088743230576, 2.924447244614667, 2.82
```

As can be seen, we first create our random number generator object, seeding it with the current system time. Then we construct our ES, specifying a terminator (that stops after a given number of function evaluations). Finally, we call the `evolve` method of the ES. To this method, we pass the generator, evaluator, the population size, a flag to denote that we're minimizing in this problem (which defaults to `maximize=True` if unspecified), a bounding function to use for candidate solutions, and a set of keyword arguments that will be needed by one or more of the functions involved. For instance, we pass `num_inputs` to be used by our generator. Likewise, `max_evaluations` will be used by our terminator.

The script outputs the best individual in the final generation, which will be located at index 0 after the final population is sorted. Since the random number generator was seeded with the current time, your particular output will be different when running this script from that presented here. You can `download the full example` to run it yourself.

[1] The evaluator was designed to evaluate all candidates, rather than a single candidate (with iteration happening inside the evolutionary computation), because this allows more complex evaluation functions that make use of the current set of individuals. Of course, such a function would also rely heavily on the choice of selector, as well. If no such elaborate mechanism is needed, then the decorator `@inspyred.ec.evaluators.evaluator` can be used on an evaluation function that operates on a single candidate. See *the reference documentation* for more details.

[2] We can also certainly create real-coded genetic algorithms, among many other choices for our EC. However, for this discussion we are attempting to use the canonical versions to which most people would be accustomed.

2.2 Evolving Polygons

In this example, we will attempt to create a polygon of *n* vertices that has maximal area. We'll also create a custom observer that allows us to display the polygon as it evolves.

2.2.1 The Generator

```
from random import Random
from time import time
from time import sleep
import inspyred
from Tkinter import *
import itertools

def generate_polygon(random, args):
    size = args.get('num_vertices', 6)
    return [(random.uniform(-1, 1), random.uniform(-1, 1)) for i in range(size)]
```

Once again, we import the necessary libraries. In this case, we'll also need to tailor elements of the EC, as well as provide graphical output.

After the libraries have been imported, we define our generator function. It looks for the keyword argument `num_vertices`, and it creates a list of `num_vertices` ordered pairs (tuples) where each coordinate is in the range [-1, 1].

2.2.2 The Evaluator

```
def segments(p):
    return zip(p, p[1:] + [p[0]])

def area(p):
    return 0.5 * abs(sum([x0*y1 - x1*y0 for ((x0, y0), (x1, y1)) in segments(p)]))

def evaluate_polygon(candidates, args):
    fitness = []
    for cs in candidates:
        fit = area(cs)
        fitness.append(fit)
    return fitness
```

In order to evaluate the polygon, we need to calculate its area. The `segments` and `area` functions do this for us. (In case it's not clear from the code, the `segments` function turns a list of coordinate pairs into a list of pairs of adjacent neighbors. For instance, `[(1, 2), (3, 4), (5, 6)]` would return `[((1, 2), (3, 4)), ((3, 4), (5, 6)), ((5, 6), (1, 2))]`.) Therefore, the `evaluate_polygon` function simply needs to assign the fitness to be the value returned as the area.

2.2.3 The Bounder

```
def bound_polygon(candidate, args):
    for i, c in enumerate(candidate):
        x = max(min(c[0], 1), -1)
        y = max(min(c[1], 1), -1)
        candidate[i] = (x, y)
    return candidate
bound_polygon.lower_bound = itertools.repeat(-1)
bound_polygon.upper_bound = itertools.repeat(1)
```

Because our representation is a bit non-standard (a list of tuples), we need to create a bounding function that the EC can use to bound potential candidate solutions. Here, the bounding function is simple enough. It just make sure that each element of each tuple lies in the range [-1, 1]. The `lower_bound` and `upper_bound` attributes are added to the function so that the `mutate_polygon` function can make use of them without being hard-coded. While this is not strictly necessary, it does mimic the behavior of the `Bounder` callable class provided by inspyred.

2.2.4 The Observer

```
def polygon_observer(population, num_generations, num_evaluations, args):
    try:
        canvas = args['canvas']
    except KeyError:
        canvas = Canvas(Tk(), bg='white', height=400, width=400)
        args['canvas'] = canvas

    # Get the best polygon in the population.
    poly = population[0].candidate
    coords = [(100*x + 200, -100*y + 200) for (x, y) in poly]
    old_polys = canvas.find_withtag('poly')
    for p in old_polys:
        canvas.delete(p)
    old_rects = canvas.find_withtag('rect')
    for r in old_rects:
        canvas.delete(r)
    old_verts = canvas.find_withtag('vert')
    for v in old_verts:
        canvas.delete(v)

    canvas.create_rectangle(100, 100, 300, 300, fill='', outline='yellow', width=6, tags='rect')
    canvas.create_polygon(coords, fill='', outline='black', width=2, tags='poly')
    vert_radius = 3
    for (x, y) in coords:
        canvas.create_oval(x-vert_radius, y-vert_radius, x+vert_radius, y+vert_radius, fill='blue', t
    canvas.pack()
    canvas.update()
    print('{0} evaluations'.format(num_evaluations))
    sleep(0.05)
```

Since we are evolving a two-dimensional shape, it makes sense to use a graphical approach to observing the current best polygon during each iteration. The `polygon_observer` accomplishes this by drawing the best polygon in the population to a Tk canvas. Notice that the canvas is passed in via the keyword arguments parameter `args`.

2.2.5 The Evolutionary Computation

For this task, we'll create a custom evolutionary computation by selecting the operators to be used. First, we will need to create a custom mutation operator since none of the pre-defined operators deal particularly well with a list of tuples.

```
def mutate_polygon(random, candidates, args):
    mut_rate = args.setdefault('mutation_rate', 0.1)
    bounder = args['_ec'].bounder
    for i, cs in enumerate(candidates):
        for j, (c, lo, hi) in enumerate(zip(cs, bounder.lower_bound, bounder.upper_bound)):
            if random.random() < mut_rate:
                x = c[0] + random.gauss(0, 1) * (hi - lo)
                y = c[1] + random.gauss(0, 1) * (hi - lo)
                candidates[i][j] = (x, y)
        candidates[i] = bounder(candidates[i], args)
    return candidates
```

Notice that this is essentially a Gaussian mutation on each coordinate of each tuple. Now we can create our custom EC.

```
rand = Random()
rand.seed(int(time()))
my_ec = inspyred.ec.EvolutionaryComputation(rand)
my_ec.selector = inspyred.ec.selectors.tournament_selection
my_ec.variator = [inspyred.ec.variators.uniform_crossover, mutate_polygon]
my_ec.replacer = inspyred.ec.replacers.steady_state_replacement
my_ec.observer = polygon_observer
my_ec.terminator = [inspyred.ec.terminators.evaluation_termination, inspyred.ec.terminators.average_
window = Tk()
window.title('Evolving Polygons')
can = Canvas(window, bg='white', height=400, width=400)
can.pack()

final_pop = my_ec.evolve(generator=generate_polygon,
                         evaluator=evaluate_polygon,
                         pop_size=100,
                         bounder=bound_polygon,
                         max_evaluations=5000,
                         num_selected=2,
                         mutation_rate=0.25,
                         num_vertices=3,
                         canvas=can)
# Sort and print the best individual, who will be at index 0.
final_pop.sort(reverse=True)
print('Terminated due to {0}.'.format(my_ec.termination_cause))
print(final_pop[0])
sleep(5)
```

This EC uses tournament selection, uniform crossover, our custom mutation operator, and steady-state replacement. We also set up the custom observer and create the canvas, which is passed into the `evolve` method as a keyword argument. You can `download the full example` to run it yourself.

2.3 Lunar Explorer

In this example [3], we will evolve the configuration for a space probe designed to travel around the Moon and return to Earth. The space probe is defined by five parameters: its orbital height, mass, boost velocity (both *x* and *y* components), and initial *y* (vertical from Earth) velocity. The physical problem which we are here using optimization to solve is known as "Gravity Assist" or "Gravity Slingshot" and is used by spacecraft to alter the direction and speed of spacecraft, reducing the need for propellant. It was first propsed by Yuri Kondratyuk and first used by the Soviet space probe Luna 3 in 1959 to take the first pictures of the never-before-seen far side of the moon. The computational power available to the designers of the Luna 3 was much smaller than what is available today. The optimization of the space craft's trajectory was therefore a very difficult task. The evaluator presented here makes some simplifying

[3] This example was suggested and implemented by Mike Vella (vellamike@gmail.com).

assumptions, but demonstrates the general principle of using evolutionary computation to solve an engineering or scientific task.

2.3.1 The Generator

```
import os
import math
import pylab
import itertools
from matplotlib import pyplot as plt
from matplotlib.patches import Circle
from random import Random
from time import time
import inspyred

def satellite_generator(random, args):
    chromosome = []
    bounder = args["_ec"].bounder
    # The constraints are as follows:
    #              orbital   satellite   boost velocity      initial y
    #              height    mass        (x,        y)        velocity
    for lo, hi in zip(bounder.lower_bound, bounder.upper_bound):
        chromosome.append(random.uniform(lo, hi))
    return chromosome
```

After the libraries have been imported, we define our generator function. It simply pulls the bounder values for each of the five parameters of the satellite and randomly chooses a value between the minimum and maximum.

2.3.2 The Evaluator

```
def pairwise(iterable):
    """s -> (s0,s1), (s1,s2), (s2, s3), ..."""
    a, b = itertools.tee(iterable)
    next(b, None)
    return itertools.izip(a, b)
```

This function breaks a one-dimensional list into a set of overlapping pairs. This is necessary because the trajectory of the satellite is a set of points, and the total distance traveled is calculated by summing the pairwise distances.

```
def distance_between(position_a, position_b):
    return math.sqrt((position_a[0] - position_b[0])**2 + (position_a[1] - position_b[1])**2)
```

This function calculates the Euclidean distance between points.

```
def gravitational_force(position_a, mass_a, position_b, mass_b):
    """Returns the gravitational force between the two bodies a and b."""
    distance = distance_between(position_a, position_b)

    # Calculate the direction and magnitude of the force.
    angle = math.atan2(position_a[1] - position_b[1], position_a[0] - position_b[0])
    magnitude = G * mass_a * mass_b / (distance**2)

    # Find the x and y components of the force.
    # Determine sign based on which one is the larger body.
    sign = -1 if mass_b > mass_a else 1
    x_force = sign * magnitude * math.cos(angle)
    y_force = sign * magnitude * math.sin(angle)
    return x_force, y_force
```

This function calculates the gravitational force between the two given bodies.

```
def force_on_satellite(position, mass):
    """Returns the total gravitational force acting on the body from the Earth and Moon."""
    earth_grav_force = gravitational_force(position, mass, earth_position, earth_mass)
    moon_grav_force = gravitational_force(position, mass, moon_position, moon_mass)
    F_x = earth_grav_force[0] + moon_grav_force[0]
    F_y = earth_grav_force[1] + moon_grav_force[1]
    return F_x, F_y
```

This function calculates the force on the satellite from both the Earth and the Moon.

```
def acceleration_of_satellite(position, mass):
    """Returns the acceleration based on all forces acting upon the body."""
    F_x, F_y = force_on_satellite(position, mass)
    return F_x / mass, F_y / mass
```

This function calculates the acceleration of the satellite due to the forces acting upon it.

```
def moonshot(orbital_height, satellite_mass, boost_velocity, initial_y_velocity,
             time_step=60, max_iterations=5e4, plot_trajectory=False):
    fitness = 0.0
    distance_from_earth_center = orbital_height + earth_radius
    eqb_velocity = math.sqrt(G * earth_mass / distance_from_earth_center)

    # Start the simulation.
    # Keep up with the positions of the satellite as it moves.
    position = [(earth_radius + orbital_height, 0.0)] # The initial position of the satellite.
    velocity = [0.0, initial_y_velocity]
    time = 0
    min_distance_from_moon = distance_between(position[-1], moon_position) - moon_radius

    i = 0
    keep_simulating = True
    rockets_boosted = False

    while keep_simulating:
        # Calculate the acceleration and corresponding change in velocity.
        # (This is effectively the Forward Euler Algorithm.)
        acceleration = acceleration_of_satellite(position[-1], satellite_mass)
        velocity[0] += acceleration[0] * time_step
        velocity[1] += acceleration[1] * time_step

        # Start the rocket burn:
        # add a boost in the +x direction of 1m/s
        # closest point to the moon
        if position[-1][1] < -100 and position[-1][0] > distance_from_earth_center-100 and not rocket
            launch_point = position[-1]
            velocity[0] += boost_velocity[0]
            velocity[1] += boost_velocity[1]
            rockets_boosted = True

        # Calculate the new position based on the velocity.
        position.append((position[-1][0] + velocity[0] * time_step,
                         position[-1][1] + velocity[1] * time_step))
        time += time_step
        if i >= max_iterations:
            keep_simulating = False

        distance_from_moon_surface = distance_between(position[-1], moon_position) - moon_radius
        distance_from_earth_surface = distance_between(position[-1], earth_position) - earth_radius
        if distance_from_moon_surface < min_distance_from_moon:
            min_distance_from_moon = distance_from_moon_surface

        # See if the satellite crashes into the Moon or the Earth, or
        # if the satellite gets too far away (radio contact is lost).
        if distance_from_moon_surface <= 0:
            fitness += 100000 # penalty of 100,000 km if crash on moon
            keep_simulating = False
        elif distance_from_earth_surface <= 0:
            keep_simulating = False
            fitness -= 100000 # reward of 100,000 km if land on earth
        elif distance_from_earth_surface > 2 * distance_between(earth_position, moon_position):
            keep_simulating = False #radio contact lost
        i += 1

    # Augment the fitness to include the minimum distance (in km)
    # that the satellite made it to the Moon (lower without crashing is better).
    fitness += min_distance_from_moon / 1000.0

    # Augment the fitness to include 1% of the total distance
    # traveled by the probe (in km). This means the probe
    # should prefer shorter paths.
    total_distance = 0
    for p, q in pairwise(position):
        total_distance += distance_between(p, q)
    fitness += total_distance / 1000.0 * 0.01

    if plot_trajectory:
        axes = plt.gca()
        earth = Circle(earth_position, earth_radius, facecolor='b', alpha=1)
        moon = Circle(moon_position, moon_radius, facecolor='0.5', alpha=1)
        axes.add_artist(earth)
        axes.add_artist(moon)
        axes.annotate('Earth', xy=earth_position,  xycoords='data',
                      xytext=(0, 1e2), textcoords='offset points',
                      arrowprops=dict(arrowstyle="->"))
        axes.annotate('Moon', xy=moon_position,  xycoords='data',
                      xytext=(0, 1e2), textcoords='offset points',
                      arrowprops=dict(arrowstyle="->"))
        x = [p[0] for p in position]
        y = [p[1] for p in position]
        cm = pylab.get_cmap('gist_rainbow')
        lines = plt.scatter(x, y, c=range(len(x)), cmap=cm, marker='o', s=2)
        plt.setp(lines, edgecolors='None')
        plt.axis("equal")
        plt.grid("on")
        projdir = os.path.dirname(os.getcwd())
        name = '{0}/{1}.pdf'.format(projdir, str(fitness))
        plt.savefig(name, format="pdf")
        plt.clf()

    return fitness
```

This function does the majority of the work for the evaluation. It accepts the parameters that are being evolved, and it simulates the trajectory of a satellite as it moves around the Moon and back to the Earth. The fitness of the trajectory is as follows:

fitness = minimum distance from moon + 1% of total distance traveled + Moon crash penalty - Earth landing reward

The penalty/reward is 100000, and the fitness is designed to be minimized.

```python
def moonshot_evaluator(candidates, args):
    fitness=[]
    for chromosome in candidates:
        orbital_height = chromosome[0]
        satellite_mass = chromosome[1]
        boost_velocity = (chromosome[2], chromosome[3])
        initial_y_velocity = chromosome[4]
        fitness.append(moonshot(orbital_height, satellite_mass, boost_velocity, initial_y_velocity))
    return fitness
```

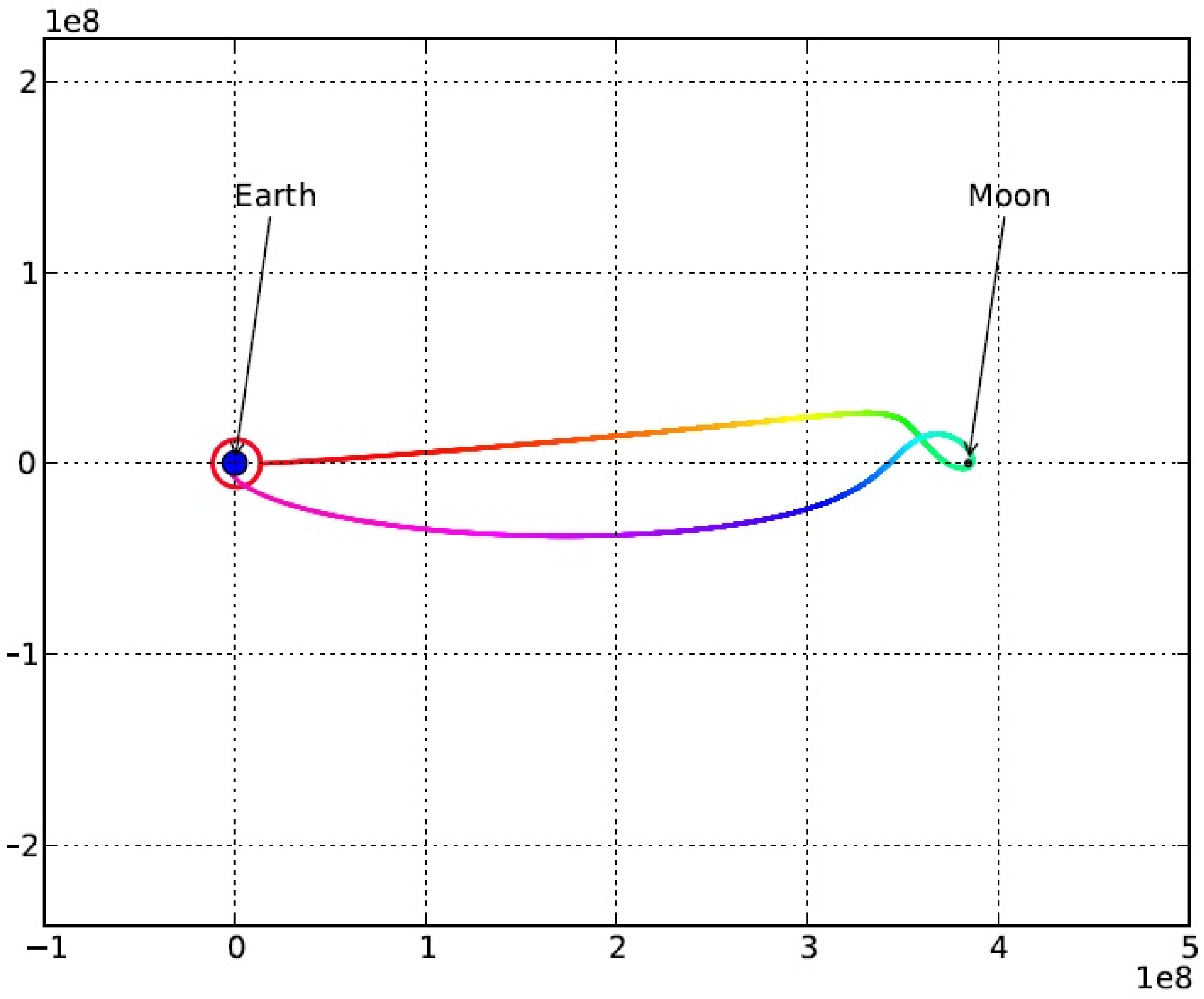
The evaluator simply calls the *moonshot* function.

2.3.3 The Evolutionary Computation

```python
rand = Random()
rand.seed(int(time()))
# The constraints are as follows:
#             orbital   satellite   boost velocity      initial y
#             height    mass        (x,        y)       velocity
constraints=((6e6,      10.0,       3e3,    -10000.0,   4000),
             (8e6,      40.0,       9e3,     10000.0,   6000))

algorithm = inspyred.ec.EvolutionaryComputation(rand)
algorithm.terminator = inspyred.ec.terminators.evaluation_termination
algorithm.observer = inspyred.ec.observers.file_observer
algorithm.selector = inspyred.ec.selectors.tournament_selection
algorithm.replacer = inspyred.ec.replacers.generational_replacement
algorithm.variator = [inspyred.ec.variators.blend_crossover, inspyred.ec.variators.gaussian_mutation]
projdir = os.path.dirname(os.getcwd())

stat_file_name = '{0}/moonshot_ec_statistics.csv'.format(projdir)
ind_file_name = '{0}/moonshot_ec_individuals.csv'.format(projdir)
stat_file = open(stat_file_name, 'w')
ind_file = open(ind_file_name, 'w')
final_pop = algorithm.evolve(generator=satellite_generator,
                             evaluator=moonshot_evaluator,
                             pop_size=100,
                             maximize=False,
                             bounder=inspyred.ec.Bounder(constraints[0], constraints[1]),
                             num_selected=100,
                             tournament_size=2,
                             num_elites=1,
                             mutation_rate=0.3,
                             max_evaluations=600,
                             statistics_file=stat_file,
                             individuals_file=ind_file)

stat_file.close()
ind_file.close()
```

```
# Sort and print the fittest individual, who will be at index 0.
final_pop.sort(reverse=True)
best = final_pop[0]
components = best.candidate
print('\nFittest individual:')
print(best)
moonshot(components[0], components[1], (components[2], components[3]), components[4], plot_trajectory
```

The results, if plotted, will look similar to the figure below. Here, the color denotes the passage of time, from red to violet. You can `download the full example` to run it yourself.



CHAPTER

THREE

EXAMPLES

For many people, it is easiest to learn a new library by adapting existing examples to their purposes. For that reason, many examples are presented in this section in the hope that they will prove useful to others using inspyred.

3.1 Standard Algorithms

The following examples illustrate how to use the different, built-in, evolutionary computations. They are (hopefully) simple and self-explanatory. Please note that each one uses an existing benchmark problem. This is just an expedience; it is not necessary. The full list of existing benchmarks can be found in the *Library Reference*. See the *Tutorial* for examples that do not make use of a benchmark problem.

3.1.1 Genetic Algorithm

In this example, a GA is used to evolve a solution to the binary version of the Schwefel benchmark. [`download`]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Binary(inspyred.benchmarks.Schwefel(2),
                                         dimension_bits=30)
    ea = inspyred.ec.GA(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          maximize=problem.maximize,
                          bounder=problem.bounder,
                          max_evaluations=30000,
                          num_elites=1)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea
```

```
if __name__ == '__main__':
    main(display=True)
```

3.1.2 Evolution Strategy

In this example, an ES is used to evolve a solution to the Rosenbrock benchmark. [download]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Rosenbrock(2)
    ea = inspyred.ec.ES(prng)
    ea.terminator = [inspyred.ec.terminators.evaluation_termination,
                     inspyred.ec.terminators.diversity_termination]
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.1.3 Simulated Annealing

In this example, an SA is used to evolve a solution to the Sphere benchmark. [download]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Sphere(2)
    ea = inspyred.ec.SA(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(evaluator=problem.evaluator,
                          generator=problem.generator,
                          maximize=problem.maximize,
                          bounder=problem.bounder,
                          max_evaluations=30000)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.1.4 Differential Evolution Algorithm

In this example, a DEA is used to evolve a solution to the Griewank benchmark. [download]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Griewank(2)
    ea = inspyred.ec.DEA(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.1.5 Estimation of Distribution Algorithm

In this example, an EDA is used to evolve a solution to the Rastrigin benchmark. [download]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Rastrigin(2)
    ea = inspyred.ec.EDA(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(evaluator=problem.evaluator,
                          generator=problem.generator,
                          pop_size=1000,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000,
                          num_selected=500,
                          num_offspring=1000,
                          num_elites=1)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.1.6 Pareto Archived Evolution Strategy (PAES)

In this example, a PAES is used to evolve a solution to the Kursawe multiobjective benchmark. [download]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Kursawe(3)
    ea = inspyred.ec.emo.PAES(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=10000,
                          max_archive_size=100,
                          num_grid_divisions=4)

    if display:
        final_arc = ea.archive
        print('Best Solutions: \n')
        for f in final_arc:
            print(f)
        import pylab
        x = []
        y = []
        for f in final_arc:
            x.append(f.fitness[0])
            y.append(f.fitness[1])
        pylab.scatter(x, y, color='b')
        pylab.savefig('{0} Example ({1}).pdf'.format(ea.__class__.__name__,
                                                              problem.__class__.__name__),
                        format='pdf')
            pylab.show()
        return ea

if __name__ == '__main__':
    main(display=True)
```

3.1.7 Nondominated Sorting Genetic Algorithm (NSGA-II)

In this example, an NSGA2 is used to evolve a solution to the Kursawe multiobjective benchmark. [download]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Kursawe(3)
    ea = inspyred.ec.emo.NSGA2(prng)
    ea.variator = [inspyred.ec.variators.blend_crossover,
                   inspyred.ec.variators.gaussian_mutation]
    ea.terminator = inspyred.ec.terminators.generation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          maximize=problem.maximize,
                          bounder=problem.bounder,
                          max_generations=80)

    if display:
        final_arc = ea.archive
        print('Best Solutions: \n')
        for f in final_arc:
            print(f)
        import pylab
        x = []
        y = []
        for f in final_arc:
            x.append(f.fitness[0])
            y.append(f.fitness[1])
        pylab.scatter(x, y, color='b')
        pylab.savefig('{0} Example ({1}).pdf'.format(ea.__class__.__name__,
                                                      problem.__class__.__name__),
                      format='pdf')
        pylab.show()
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.1.8 Particle Swarm Optimization

In this example, a PSO is used to evolve a solution to the Ackley benchmark. [download]

```python
from time import time
from random import Random
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Ackley(2)
    ea = inspyred.swarm.PSO(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    ea.topology = inspyred.swarm.topologies.ring_topology
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000,
                          neighborhood_size=5)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.1.9 Ant Colony Optimization

In this example, an ACS is used to evolve a solution to the TSP benchmark. [download]

```python
from random import Random
from time import time
import math
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    points = [(110.0, 225.0), (161.0, 280.0), (325.0, 554.0), (490.0, 285.0),
              (157.0, 443.0), (283.0, 379.0), (397.0, 566.0), (306.0, 360.0),
              (343.0, 110.0), (552.0, 199.0)]
    weights = [[0 for _ in range(len(points))] for _ in range(len(points))]
    for i, p in enumerate(points):
        for j, q in enumerate(points):
            weights[i][j] = math.sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)

    problem = inspyred.benchmarks.TSP(weights)
    ac = inspyred.swarm.ACS(prng, problem.components)
    ac.terminator = inspyred.ec.terminators.generation_termination
    final_pop = ac.evolve(generator=problem.constructor,
                          evaluator=problem.evaluator,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          pop_size=10,
                          max_generations=50)

    if display:
        best = max(ac.archive)
        print('Best Solution:')
        for b in best.candidate:
            print(points[b.element[0]])
        print(points[best.candidate[-1].element[1]])
        print('Distance: {0}'.format(1/best.fitness))
    return ac

if __name__ == '__main__':
    main(display=True)
```

3.2 Customized Algorithms

The true benefit of the inspyred library is that it allows the programmer to customize almost every aspect of the algorithm. This is accomplished primarily through the use of function (or function-like) callbacks that can be specified by the programmer. The following examples show how to customize many different parts of the evolutionary computation.

3.2.1 Custom Evolutionary Computation

In this example, an evolutionary computation is created which uses tournament selection, uniform crossover, Gaussian mutation, and steady-state replacement. [download]

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Ackley(2)
    ea = inspyred.ec.EvolutionaryComputation(prng)
    ea.selector = inspyred.ec.selectors.tournament_selection
    ea.variator = [inspyred.ec.variators.uniform_crossover,
                   inspyred.ec.variators.gaussian_mutation]
    ea.replacer = inspyred.ec.replacers.steady_state_replacement
    ea.terminator = inspyred.ec.terminators.generation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          tournament_size=7,
                              num_selected=2,
                              max_generations=300,
                              mutation_rate=0.2)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.2.2 Custom Archiver

The purpose of the archiver is to provide a mechanism for candidate solutions to be maintained without necessarily remaining in the population. This is important for most multiobjective evolutionary approaches, but it can also be useful for single-objective problems, as well. In this example, an archiver is created that maintains the *worst* individual found. (There is no imaginable reason why one might actually do this. It is just for illustration purposes.) [download]

```
from random import Random
from time import time
import inspyred

def my_archiver(random, population, archive, args):
    worst_in_pop = min(population)
    if len(archive) > 0:
        worst_in_arc = min(archive)
        if worst_in_pop < worst_in_arc:
            return [worst_in_pop]
        else:
            return archive
    else:
        return [worst_in_pop]

if __name__ == '__main__':
    prng = Random()
    prng.seed(time())

    problem = inspyred.benchmarks.Rosenbrock(2)
    ea = inspyred.ec.ES(prng)
    ea.observer = [inspyred.ec.observers.stats_observer,
                   inspyred.ec.observers.archive_observer]
    ea.archiver = my_archiver
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000)
    best = max(final_pop)
    print('Best Solution: \n{0}'.format(str(best)))
    print(ea.archive)
```

3.2.3 Custom Observer

Sometimes it is helpful to see certain aspects of the current population as it evolves. The purpose of the "observer" functions is to provide a function that executes at the end of each generation so that the process can be monitored accordingly. In this example, the only information desired at each generation is the current best individual. [download]

```
from random import Random
from time import time
import inspyred

def my_observer(population, num_generations, num_evaluations, args):
    best = max(population)
    print('{0:6} -- {1} : {2}'.format(num_generations,
                                      best.fitness,
                                      str(best.candidate)))

if __name__ == '__main__':
    prng = Random()
    prng.seed(time())

    problem = inspyred.benchmarks.Rastrigin(2)
    ea = inspyred.ec.ES(prng)
    ea.observer = my_observer
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000)
    best = max(final_pop)
    print('Best Solution: \n{0}'.format(str(best)))
```

3.2.4 Custom Replacer

The replacers are used to determine which of the parents, offspring, and current population should survive into the next generation. In this example, survivors are determined to be top 50% of individuals from the population along with 50% chosen randomly from the offspring. (Once again, this is simply an example. There may be no good reason to create such a replacement scheme.) [download]

```
from random import Random
from time import time
import inspyred

def my_replacer(random, population, parents, offspring, args):
    psize = len(population)
    population.sort(reverse=True)
    survivors = population[:psize // 2]
    num_remaining = psize - len(survivors)
    for i in range(num_remaining):
        survivors.append(random.choice(offspring))
    return survivors

if __name__ == '__main__':
    prng = Random()
    prng.seed(time())
    problem = inspyred.benchmarks.Ackley(2)
    ea = inspyred.ec.ES(prng)
    ea.replacer = my_replacer
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000)
    best = max(final_pop)
    print('Best Solution: \n{0}'.format(str(best)))
```

3.2.5 Custom Selector

The selectors are used to determine which individuals in the population should become parents. In this example, parents are chosen such that 50% of the time the best individual in the population is chosen to be a parent and 50% of the time a random individual is chosen. As before, this is an example selector that may have little practical value. [download]

```
from random import Random
from time import time
import inspyred

def my_selector(random, population, args):
    n = args.get('num_selected', 2)
    best = max(population)
    selected = []
    for i in range(n):
        if random.random() <= 0.5:
            selected.append(best)
        else:
            selected.append(random.choice(population))
    return selected

if __name__ == '__main__':
    prng = Random()
    prng.seed(time())

    problem = inspyred.benchmarks.Griewank(2)
    ea = inspyred.ec.DEA(prng)
    ea.selector = my_selector
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=100,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000)

    best = max(final_pop)
    print('Best Solution: \n{0}'.format(str(best)))
```

3.2.6 Custom Terminator

The terminators are used to determine when the evolutionary process should end. All terminators return a Boolean value where True implies that the evolution should end. In this example, the evolution should continue until the average Hamming distance between all combinations of candidates falls below a specified minimum. [download]

```
from random import Random
from time import time
import itertools
import inspyred

def my_terminator(population, num_generations, num_evaluations, args):
    min_ham_dist = args.get('minimum_hamming_distance', 30)
    ham_dist = []
    for x, y in itertools.combinations(population, 2):
        ham_dist.append(sum(a != b for a, b in zip(x.candidate, y.candidate)))
    avg_ham_dist = sum(ham_dist) / float(len(ham_dist))
    return avg_ham_dist <= min_ham_dist


if __name__ == '__main__':
    prng = Random()
    prng.seed(time())

    problem = inspyred.benchmarks.Binary(inspyred.benchmarks.Schwefel(2),
                                         dimension_bits=30)
    ea = inspyred.ec.GA(prng)
    ea.terminator = my_terminator
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=10,
                          maximize=problem.maximize,
                          bounder=problem.bounder,
                          num_elites=1,
                          minimum_hamming_distance=12)

    best = max(final_pop)
    print('Best Solution: \n{0}'.format(str(best)))
```

3.2.7 Custom Variator

The variators provide what are normally classified as “crossover” and “mutation,” however even more exotic variators can be defined. Remember that a list of variators can be specified that will act as a pipeline with the output of the first being used as the input to the second, etc. In this example, the binary candidate is mutated such that two points are chosen and the bits between each point are put in reverse order. For example, `0010100100` would become `0010010100` if the third and eighth bits are the chosen points. [download]

```
from random import Random
from time import time
import inspyred

# Note that we could have used the @inspyred.ec.variators.mutator
# decorator here and simplified our custom variator to
#
#     def my_variator(random, candidate, args)
#
# where candidate is a single candidate. Such a function would
# just return the single mutant.
def my_variator(random, candidates, args):
    mutants = []
    for c in candidates:
        points = random.sample(range(len(c)), 2)
        x, y = min(points), max(points)
        if x == 0:
            mutants.append(c[y::-1] + c[y+1:])
        else:
            mutants.append(c[:x] + c[y:x-1:-1] + c[y+1:])
    return mutants

if __name__ == '__main__':
    prng = Random()
    prng.seed(time())

    problem = inspyred.benchmarks.Binary(inspyred.benchmarks.Schwefel(2),
                                         dimension_bits=30)
    ea = inspyred.ec.GA(prng)
    ea.variator = my_variator
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=10,
                          maximize=problem.maximize,
                          bounder=problem.bounder,
                          num_elites=1,
                          max_evaluations=20000)

    best = max(final_pop)
    print('Best Solution: \n{0}'.format(str(best)))
```

3.3 Advanced Usage

The examples in this section deal with less commonly used aspects of the library. Be aware that these parts may not have received as much testing as the more core components exemplified above.

3.3.1 Discrete Optimization

Discrete optimization problems often present difficulties for naive evolutionary computation approaches. Special care must be taken to generate and maintain feasible solutions and/or to sufficiently penalize infeasible solutions. Ant colony optimization approaches were created to deal with discrete optimization problems. In these examples, we consider two of the most famous discrete optimization benchmark problems – the Traveling Salesman Problem (TSP) and the Knapsack problem. The background on these problems is omitted here because it can easily be found elsewhere.

The Traveling Salesman Problem

Candidate solutions for the TSP can be most easily be represented as permutations of the list of city numbers (enumerating the order in which cities should be visited). For instance, if there are 5 cities, then a candidate solution might be `[4, 1, 0, 2, 3]`. This is how the `TSP` benchmark represents solutions. However, this simple representation forces us to work harder to ensure that our solutions remain feasible during crossover and mutation. Therefore, we need to use variators suited to the task, as shown in the example below. [download]

```python
from random import Random
from time import time
import math
import inspyred


def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    points = [(110.0, 225.0), (161.0, 280.0), (325.0, 554.0), (490.0, 285.0),
              (157.0, 443.0), (283.0, 379.0), (397.0, 566.0), (306.0, 360.0),
              (343.0, 110.0), (552.0, 199.0)]
    weights = [[0 for _ in range(len(points))] for _ in range(len(points))]
    for i, p in enumerate(points):
        for j, q in enumerate(points):
            weights[i][j] = math.sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)

    problem = inspyred.benchmarks.TSP(weights)
    ea = inspyred.ec.EvolutionaryComputation(prng)
    ea.selector = inspyred.ec.selectors.tournament_selection
    ea.variator = [inspyred.ec.variators.partially_matched_crossover,
                   inspyred.ec.variators.inversion_mutation]
    ea.replacer = inspyred.ec.replacers.generational_replacement
    ea.terminator = inspyred.ec.terminators.generation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          pop_size=100,
                          max_generations=50,
                          tournament_size=5,
                          num_selected=100,
                          num_elites=1)

    if display:
        best = max(ea.population)
        print('Best Solution: {0}: {1}'.format(str(best.candidate), 1/best.fitness))
    return ea

if __name__ == '__main__':
    main(display=True)
```

As an alternative, we can use ant colony optimization to solve the TSP. This example was shown previously, but it is presented again here for completeness. [download]

```python
from random import Random
from time import time
import math
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    points = [(110.0, 225.0), (161.0, 280.0), (325.0, 554.0), (490.0, 285.0),
              (157.0, 443.0), (283.0, 379.0), (397.0, 566.0), (306.0, 360.0),
              (343.0, 110.0), (552.0, 199.0)]
    weights = [[0 for _ in range(len(points))] for _ in range(len(points))]
    for i, p in enumerate(points):
        for j, q in enumerate(points):
            weights[i][j] = math.sqrt((p[0] - q[0])**2 + (p[1] - q[1])**2)

    problem = inspyred.benchmarks.TSP(weights)
    ac = inspyred.swarm.ACS(prng, problem.components)
    ac.terminator = inspyred.ec.terminators.generation_termination
    final_pop = ac.evolve(generator=problem.constructor,
                          evaluator=problem.evaluator,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          pop_size=10,
                          max_generations=50)

    if display:
        best = max(ac.archive)
        print('Best Solution:')
        for b in best.candidate:
            print(points[b.element[0]])
        print(points[best.candidate[-1].element[1]])
        print('Distance: {0}'.format(1/best.fitness))
    return ac

if __name__ == '__main__':
    main(display=True)
```

The Knapsack Problem

Candidate solutions for the Knapsack problem can be represented as either a binary list (for the 0/1 Knapsack) or as a list of non-negative integers (for the Knapsack with duplicates). In each case, the list is the same length as the number of items, and each element of the list corresponds to the quantity of the corresponding item to place in the knapsack. For the evolutionary computation, we can use `uniform_crossover` *and* `gaussian_mutation`. The reason we are able to use Gaussian mutation here, even though the candidates are composed of discrete values, is because the bounder created by the `Knapsack` benchmark is an instance of `DiscreteBounder`, which automatically moves an illegal component to its nearest legal value. [download]

```
from random import Random
from time import time
import inspyred


def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    items = [(7,369), (10,346), (11,322), (10,347), (12,348), (13,383),
             (8,347), (11,364), (8,340), (8,324), (13,365), (12,314),
             (13,306), (13,394), (7,326), (11,310), (9,400), (13,339),
             (5,381), (14,353), (6,383), (9,317), (6,349), (11,396),
             (14,353), (9,322), (5,329), (5,386), (5,382), (4,369),
             (6,304), (10,392), (8,390), (8,307), (10,318), (13,359),
             (9,378), (8,376), (11,330), (9,331)]
    problem = inspyred.benchmarks.Knapsack(15, items, duplicates=True)
    ea = inspyred.ec.EvolutionaryComputation(prng)
    ea.selector = inspyred.ec.selectors.tournament_selection
    ea.variator = [inspyred.ec.variators.uniform_crossover,
                   inspyred.ec.variators.gaussian_mutation]
    ea.replacer = inspyred.ec.replacers.steady_state_replacement
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          pop_size=100,
                          max_evaluations=2500,
                          tournament_size=5,
                          num_selected=2)

    if display:
        best = max(ea.population)
        print('Best Solution: {0}: {1}'.format(str(best.candidate),
                                               best.fitness))
    return ea

if __name__ == '__main__':
    main(display=True)
```

Once again, as an alternative we can use ant colony optimization. Just for variety, we'll use it to solve the 0/1 Knapsack problem (`duplicates=False`). [`download`]

```
from random import Random
from time import time
import math
import inspyred


def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    items = [(7,369), (10,346), (11,322), (10,347), (12,348), (13,383),
             (8,347), (11,364), (8,340), (8,324), (13,365), (12,314),
             (13,306), (13,394), (7,326), (11,310), (9,400), (13,339),
             (5,381), (14,353), (6,383), (9,317), (6,349), (11,396),
             (14,353), (9,322), (5,329), (5,386), (5,382), (4,369),
             (6,304), (10,392), (8,390), (8,307), (10,318), (13,359),
             (9,378), (8,376), (11,330), (9,331)]

    problem = inspyred.benchmarks.Knapsack(15, items, duplicates=False)
    ac = inspyred.swarm.ACS(prng, problem.components)
    ac.terminator = inspyred.ec.terminators.generation_termination
    final_pop = ac.evolve(problem.constructor, problem.evaluator,
                          maximize=problem.maximize, pop_size=50,
                          max_generations=50)

    if display:
        best = max(ac.archive)
        print('Best Solution: {0}: {1}'.format(str(best.candidate),
                                               best.fitness))
    return ac

if __name__ == '__main__':
    main(display=True)
```

3.3.2 Evaluating Individuals Concurrently

One of the most lauded aspects of many bio-inspired algorithms is their inherent parallelism. Taking advantage of this parallelism is important in real-world problems, which are often computationally expensive. There are two approaches available in inspyred to perform parallel evaluations for candidate solutions. The first makes use of the `multiprocessing` module that exists in core Python 2.6+. The second makes use of a third-party library called Parallel Python (pp), which can be used for either multi-core processing on a single machine or distributed processing across a network.

The `multiprocessing` Module

Using the `multiprocessing` approach to parallel evaluations is probably the better choice if all evaluations are going to be split among multiple processors or cores on a single machine. This is because the module is part of the core Python library (2.6+) and provides a simple, standard interface for setting up the parallelism. As shown in the example below, the only additional parameter is the name of the "actual" evaluation function and the (optional) number of CPUs to use. [`download`]

```
from random import Random
from time import time
import inspyred
import math

def generate_rastrigin(random, args):
    size = args.get('num_inputs', 10)
    return [random.uniform(-5.12, 5.12) for i in xrange(size)]

def evaluate_rastrigin(candidates, args):
    fitness = []
    for cs in candidates:
        fit = 10 * len(cs) + sum([((x - 1)**2 - 10 *
                                   math.cos(2 * math.pi * (x - 1)))
                                   for x in cs])
        fitness.append(fit)
    return fitness

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    ea = inspyred.ec.DEA(prng)
    if display:
        ea.observer = inspyred.ec.observers.stats_observer
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=generate_rastrigin,
                          evaluator=inspyred.ec.evaluators.parallel_evaluation_mp,
                          mp_evaluator=evaluate_rastrigin,
                          mp_num_cpus=8,
                          pop_size=8,
                          bounder=inspyred.ec.Bounder(-5.12, 5.12),
                          maximize=False,
                          max_evaluations=256,
                          num_inputs=3)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

Parallel Python

The Parallel Python approach to multiprocessing is best suited for use across a network of computers (though it *can* be used on a single machine, as well). It takes just a little effort to install and setup pp on each machine, but once it's complete, the network becomes a very efficient computing cluster. This is a capability that the `multiprocessing` approach simply does not have, and it provides incredible scalability. However, pp requires additional, non-standard parameters to be passed in, as illustrated in the example below. [`download`]

```
from random import Random
from time import time
import inspyred
import math

# Define an additional "necessary" function for the evaluator
# to see how it must be handled when using pp.
def my_squaring_function(x):
    return x**2

def generate_rastrigin(random, args):
    size = args.get('num_inputs', 10)
    return [random.uniform(-5.12, 5.12) for i in xrange(size)]

def evaluate_rastrigin(candidates, args):
    fitness = []
    for cs in candidates:
        fit = 10 * len(cs) + sum([(my_squaring_function(x - 1) -
                                   10 * math.cos(2 * math.pi * (x - 1)))
                                   for x in cs])
        fitness.append(fit)
    return fitness

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    ea = inspyred.ec.DEA(prng)
    if display:
        ea.observer = inspyred.ec.observers.stats_observer
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=generate_rastrigin,
                          evaluator=inspyred.ec.evaluators.parallel_evaluation_pp,
                          pp_evaluator=evaluate_rastrigin,
                          pp_dependencies=(my_squaring_function,),
                          pp_modules=("math",),
                          pop_size=8,
                          bounder=inspyred.ec.Bounder(-5.12, 5.12),
                          maximize=False,
                          max_evaluations=256,
                          num_inputs=3)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

3.3.3 Replacement via Niching

An example of a not-quite-standard replacer is the `crowding_replacement` which provides a niching capability. An example using this replacer is given below. Here, the candidates are just single numbers (but created as singleton lists so as to be `Sequence` types for some of the built-in operators) between 0 and 26. Their fitness values are simply the value of the sine function using the candidate value as input. Since the sine function is periodic and goes through four periods between 0 and 26, this function is multimodal. So we can use `crowding_replacement` to ensure that all maxima are found. [download]

```
import random
import time
import math
import itertools
import inspyred

def my_distance(x, y):
    return sum([abs(a - b) for a, b in zip(x, y)])

def generate(random, args):
    return [random.uniform(0, 26)]

def evaluate(candidates, args):
    fitness = []
    for cand in candidates:
        fit = sum([math.sin(c) for c in cand])
        fitness.append(fit)
    return fitness

def main(prng=None, display=False):
    if prng is None:
        prng = random.Random()
        prng.seed(time.time())

    ea = inspyred.ec.EvolutionaryComputation(prng)
    ea.selector = inspyred.ec.selectors.tournament_selection
    ea.replacer = inspyred.ec.replacers.crowding_replacement
    ea.variator = inspyred.ec.variators.gaussian_mutation
    ea.terminator = inspyred.ec.terminators.evaluation_termination

    final_pop = ea.evolve(generate, evaluate, pop_size=30,
                          bounder=inspyred.ec.Bounder(0, 26),
                          max_evaluations=10000,
                          num_selected=30,
                          mutation_rate=1.0,
                          crowding_distance=10,
                          distance_function=my_distance)

    if display:
        import pylab
        x = []
        y = []
        for p in final_pop:
            x.append(p.candidate[0])
            y.append(math.sin(p.candidate[0]))
        t = [(i / 1000.0) * 26.0 for i in range(1000)]
        s = [math.sin(a) for a in t]
        pylab.plot(t, s, color='b')
        pylab.scatter(x, y, color='r')
        pylab.axis([0, 26, 0, 1.1])
        pylab.savefig('niche_example.pdf', format='pdf')
        pylab.show()
    return ea

if __name__ == '__main__':
    main(display=True)
```

CHAPTER

FOUR

RECIPES

This section provides a set of recipes that can be used to add additional functionality to inspyred. These recipes are not a part of the core library, but they have proven to be useful in the past for real-world programs. If they continue to be useful, they may be incorporated into inspyred in a future version.

4.1 Lexicographic Ordering

In multiobjective optimization problems, alternatives to Pareto preference include linear weighting of the objectives and prioritizing the objectives from most to least important. Both of these methods essentially reduce the problem to a single objective optimization. Obviously, the weighting of the objectives would be handled entirely in the evaluator for the problem, so no special recipe is needed. But the prioritizing of the objectives, which is also known as lexicographic ordering, requires some additional effort.

The fitness values for two individuals, *x* and *y*, should be compared such that, if the first objective for *x* is "better" (i.e., lower when minimizing or higher when maximizing) than the first objective for *y*, then *x* is considered "better" than *y*. If they are equal in that objective, then the second objective is considered in the same way. This process is repeated for all objectives.

The following recipe provides a class to deal with such comparisons that is intended to function much like the `inspyred.ec.emo.Pareto` class. [`download`]

```
import functools

@functools.total_ordering
class Lexicographic(object):
    def __init__(self, values=None, maximize=True):
        if values is None:
            values = []
        self.values = values
        try:
            iter(maximize)
        except TypeError:
            maximize = [maximize for v in values]
        self.maximize = maximize

    def __len__(self):
        return len(self.values)

    def __getitem__(self, key):
        return self.values[key]

    def __iter__(self):
        return iter(self.values)

    def __lt__(self, other):
        for v, o, m in zip(self.values, other.values, self.maximize):
            if m:
                if v < o:
                    return True
                elif v > o:
                    return False
            else:
                if v > o:
                    return True
                elif v < o:
                    return False
        return False

    def __eq__(self, other):
        return (self.values == other.values and self.maximize == other.maximize)

    def __str__(self):
        return str(self.values)

    def __repr__(self):
        return str(self.values)


def my_evaluator(candidates, args):
    fitness = []
    for candidate in candidates:
        f = candidate[0] ** 2 + 1
        g = candidate[0] ** 2 - 1
        fitness.append(Lexicographic([f, g], maximize=False))
    return fitness

def my_generator(random, args):
    return [random.random()]

if __name__ == '__main__':
    a = Lexicographic([1, 2, 3], maximize=True)
    b = Lexicographic([1, 3, 2], maximize=True)
    c = Lexicographic([2, 1, 3], maximize=True)
    d = Lexicographic([2, 3, 1], maximize=True)
    e = Lexicographic([3, 1, 2], maximize=True)
    f = Lexicographic([3, 2, 1], maximize=True)

    u = Lexicographic([1, 2, 3], maximize=False)
    v = Lexicographic([1, 3, 2], maximize=False)
    w = Lexicographic([2, 1, 3], maximize=False)
    x = Lexicographic([2, 3, 1], maximize=False)
    y = Lexicographic([3, 1, 2], maximize=False)
    z = Lexicographic([3, 2, 1], maximize=False)

    for p in [a, b, c, d, e, f]:
        for q in [a, b, c, d, e, f]:
            print('%s < %s : %s' % (p, q, p < q))
    print('----------------------------------------')
    for p in [u, v, w, x, y, z]:
        for q in [u, v, w, x, y, z]:
            print('%s < %s : %s' % (p, q, p < q))
```

4.2 Constraint Selection

Optimization problems often have to deal with constraints and constraint violations. The following recipe provides one example of how to handle such a thing with inspyred. Here, candidates represent ordered pairs and their fitness is simply their distance from the origin. However, we provide a constraint that punishes candidates that lie outside of the unit circle. Such a scenario should produce a candidate that lies on the unit circle. Note also that `crowding_replacement` or some other fitness sharing or niching scheme could be used to generate many such points on the circle. [download]

```
import random
from inspyred import ec
from inspyred.ec import variators
from inspyred.ec import replacers
from inspyred.ec import terminators
from inspyred.ec import observers

def my_constraint_function(candidate):
    """Return the number of constraints that candidate violates."""
    # In this case, we'll just say that the point has to lie
    # within a circle centered at (0, 0) of radius 1.
    if candidate[0]**2 + candidate[1]**2 > 1:
        return 1
    else:
        return 0

def my_generator(random, args):
    # Create pairs in the range [-2, 2].
    return [random.uniform(-2.0, 2.0) for i in range(2)]

def my_evaluator(candidates, args):
    # The fitness will be how far the point is from
    # the origin. (We're maximizing, in this case.)
    # Note that the constraint heavily punishes individuals
    # who go beyond the unit circle. Therefore, these
    # two functions combined focus the evolution toward
    # finding individual who lie ON the circle.
    fitness = []
    for c in candidates:
        if my_constraint_function(c) > 0:
            fitness.append(-1)
        else:
            fitness.append(c[0]**2 + c[1]**2)
    return fitness

def constrained_tournament_selection(random, population, args):
    num_selected = args.setdefault('num_selected', 1)
    constraint_func = args.setdefault('constraint_function', None)
    tournament_size = 2
    pop = list(population)
    selected = []
    for _ in range(num_selected):
        tournament = random.sample(pop, tournament_size)
        # If there is not a constraint function,
        # just do regular tournament selection.
        if constraint_func is None:
            selected.append(max(tournament))
        else:
            cons = [constraint_func(t.candidate) for t in tournament]
            # If no constraints are violated, just do
            # regular tournament selection.
            if max(cons) == 0:
                selected.append(max(tournament))
            # Otherwise, choose the least violator
            # (which may be a non-violator).
            else:
                selected.append(tournament[cons.index(min(cons))])
    return selected

r = random.Random()
myec = ec.EvolutionaryComputation(r)
myec.selector = constrained_tournament_selection
myec.variator = variators.gaussian_mutation
myec.replacer = replacers.generational_replacement
myec.terminator = terminators.evaluation_termination
myec.observer = observers.stats_observer
pop = myec.evolve(my_generator, my_evaluator,
                  pop_size=100,
                  bounder=ec.Bounder(-2, 2),
                  num_selected=100,
                  constraint_func=my_constraint_function,
                  mutation_rate=0.5,
                  max_evaluations=2000)

import pylab
x = []
y = []
c = []
pop.sort()
num_feasible = len([p for p in pop if p.fitness >= 0])
feasible_count = 0
for i, p in enumerate(pop):
    x.append(p.candidate[0])
    y.append(p.candidate[1])
    if i == len(pop) - 1:
        c.append('r')
    elif p.fitness < 0:
        c.append('0.98')
    else:
        c.append(str(1 - feasible_count / float(num_feasible)))
        feasible_count += 1
angles = pylab.linspace(0, 2*pylab.pi, 100)
pylab.plot(pylab.cos(angles), pylab.sin(angles), color='b')
pylab.scatter(x, y, color=c)
pylab.savefig('constraint_example.pdf', format='pdf')
```

4.3 Meta-Evolutionary Computation

The following recipe shows how an evolutionary computation can be used to evolve near-optimal operators and parameters for another evolutionary computation. In the EC literature, such a thing is generally referred to as a “meta-EC”.

[download]

```
import csv
import time
import random
from inspyred import ec
from inspyred.ec import selectors
from inspyred.ec import replacers
from inspyred.ec import variators
from inspyred.ec import terminators
from inspyred.ec import observers


class MetaEC(ec.EvolutionaryComputation):
    def __init__(self, random):
        ec.EvolutionaryComputation.__init__(self, random)
        self.selector = selectors.tournament_selection
        self.replacer = replacers.generational_replacement
        self.variator = [variators.uniform_crossover, self._internal_variator]
        self.terminator = self._internal_meta_terminator

    def _create_selector_replacer(self, random):
        pop_size = random.randint(1, 100)
        selector = random.choice(range(0, 5))
        replacer = random.choice(range(0, 8))
        sel = [selector]
        if selector > 0:
            if replacer == 0 or replacer == 2 or replacer == 3:
                sel.append(pop_size)
            else:
                sel.append(random.randint(1, pop_size))
        if selector == 2:
            sel.append(random.randint(min(2, pop_size), pop_size))
        rep = [replacer]
        if replacer == 1:
            rep.append(random.randint(min(2, pop_size), pop_size))
        elif replacer == 3 or replacer == 5:
            rep.append(random.randint(0, pop_size))
        return [pop_size, sel, rep]

    def _create_variators(self, random):
        crossover = random.choice([0, 1, 2, 3, 4, 6])
        mutator = random.choice([5, 6])
        variators = ([crossover], [mutator])
        if crossover == 0 or crossover == 4:
            variators[0].append(random.random())
            variators[0].append(random.random())
        elif crossover == 1:
            variators[0].append(random.random())
        elif crossover == 2:
            variators[0].append(random.random())
            variators[0].append(random.randint(1, 10))
        elif crossover == 3:
            variators[0].append(random.randint(0, 30))
        if mutator == 5:
            variators[1].append(random.random())
            variators[1].append(random.random())
        return variators
    def _internal_generator(self, random, args):
        cross, mut = self._create_variators(random)
        return [self._create_selector_replacer(random), cross, mut]

    def _internal_variator(self, random, candidates, args):
        cs_copy = list(candidates)
        for i, cs in enumerate(cs_copy):
            if random.random() < 0.1:
                cs_copy[i][0] = self._create_selector_replacer(random)
            if random.random() < 0.1:
                cross, mut = self._create_variators(random)
                cs_copy[i][1] = cross
                cs_copy[i][2] = mut
        return cs_copy

    def _internal_observer(self, population, num_generations, num_evaluations, args):
        for i, p in enumerate(population):
            self._observer_file.write('{0}, {1}, {2}\n'.format(i, p.fitness, str(p.candidate)))
            self._observer_file.flush()

    def _internal_terminator(self, population, num_generations, num_evaluations, args):
        maxevals = args.get('max_evaluations', 0)
        self._meta_evaluations += num_evaluations
        return num_evaluations >= maxevals or self._meta_evaluations >= self._max_meta_evaluations

    def _internal_meta_terminator(self, population, num_generations, num_evaluations, args):
        return self._meta_evaluations >= self._max_meta_evaluations

    def _internal_evaluator(self, candidates, args):
        the_generator = args.get('the_generator')
        the_evaluator = args.get('the_evaluator')
        do_maximize = args.get('do_maximize', True)

        fitness = []
        for candidate in candidates:
            popsize, selector, replacer, crossover, mutator, myargs = self.interpret_candidate(candid
            myargs['max_evaluations'] = args.get('num_trial_evaluations', popsize * 10)
            num_trials = args.get('num_trials', 1)
            evo = ec.EvolutionaryComputation(self._random)
            evo.terminator = self._internal_terminator
            evo.observer = self._internal_observer
            evo.selector = selector
            evo.variator = [crossover, mutator]
            evo.replacer = replacer
            best_fit = []
            for i in range(num_trials):
                final_pop = evo.evolve(generator=the_generator,
                                       evaluator=the_evaluator,
                                       pop_size=popsize,
                                       maximize=do_maximize,
                                       args=myargs)
                best_fit.append(final_pop[0].fitness)
            fitness.append(sum(best_fit) / float(len(best_fit)))
        return fitness

    def interpret_candidate(self, candidate):
        selector_mapping = (selectors.default_selection,
                            selectors.rank_selection,
                            selectors.tournament_selection,
                            selectors.truncation_selection,
                            selectors.uniform_selection)
        variator_mapping = (variators.blend_crossover,
                            variators.heuristic_crossover,
                            variators.n_point_crossover,
                            variators.simulated_binary_crossover,
                            variators.uniform_crossover,
                            variators.gaussian_mutation,
                            variators.default_variation)
        replacer_mapping = (replacers.comma_replacement,
                            replacers.crowding_replacement,
                            replacers.default_replacement,
                            replacers.generational_replacement,
                            replacers.plus_replacement,
                            replacers.random_replacement,
                            replacers.steady_state_replacement,
                            replacers.truncation_replacement)

        myargs = dict()
        # Selectors
        if candidate[0][1][0] == 1:
            myargs['num_selected'] = candidate[0][1][1]
        elif candidate[0][1][0] == 2:
            myargs['num_selected'] = candidate[0][1][1]
            myargs['tournament_size'] = candidate[0][1][2]
        elif candidate[0][1][0] == 3:
            myargs['num_selected'] = candidate[0][1][1]
        elif candidate[0][1][0] == 4:
            myargs['num_selected'] = candidate[0][1][1]

        # Replacers
        if candidate[0][2][0] == 1:
            myargs['crowding_distance'] = candidate[0][2][1]
        elif candidate[0][2][0] == 3:
            myargs['num_elites'] = candidate[0][2][1]
        elif candidate[0][2][0] == 5:
            myargs['num_elites'] = candidate[0][2][1]

        # Crossovers
        if candidate[1][0] == 0:
            myargs['crossover_rate'] = candidate[1][1]
            myargs['blx_alpha'] = candidate[1][2]
        elif candidate[1][0] == 1:
            myargs['crossover_rate'] = candidate[1][1]
        elif candidate[1][0] == 2:
            myargs['crossover_rate'] = candidate[1][1]
            myargs['num_crossover_points'] = candidate[1][2]
        elif candidate[1][0] == 3:
            myargs['sbx_distribution_index'] = candidate[1][1]
        elif candidate[1][0] == 4:
            myargs['crossover_rate'] = candidate[1][1]
            myargs['ux_bias'] = candidate[1][2]

        # Mutators
        if candidate[2][0] == 5:
            myargs['mutation_rate'] = candidate[2][1]
            myargs['gaussian_stdev'] = candidate[2][2]
        return (candidate[0][0],
                selector_mapping[candidate[0][1][0]],
                replacer_mapping[candidate[0][2][0]],
                variator_mapping[candidate[1][0]],
                variator_mapping[candidate[2][0]],
                myargs)

    def evolve(self, generator, evaluator, pop_size=100, seeds=[], maximize=True, **args):
        args.setdefault('the_generator', generator)
        args.setdefault('the_evaluator', evaluator)
        args.setdefault('do_maximize', maximize)
        args.setdefault('num_elites', 1)
        args.setdefault('num_selected', pop_size)
        self._observer_file = open('metaec-individuals-file-' + time.strftime('%m%d%Y-%H%M%S') + '.cs
        self._meta_evaluations = 0
        self._max_meta_evaluations = args.get('max_evaluations', 0)
        final_pop = ec.EvolutionaryComputation.evolve(self, self._internal_generator,
                                                      self._internal_evaluator, pop_size,
                                                      seeds, maximize, **args)
        self._observer_file.close()
        return final_pop

if __name__ == '__main__':
    import math
    import inspyred

    prng = random.Random()
    prng.seed(time.time())
    problem = inspyred.benchmarks.Rastrigin(3)
    mec = MetaEC(prng)
    mec.observer = observers.stats_observer
    final_pop = mec.evolve(generator=problem.generator,
                           evaluator=problem.evaluator,
                           pop_size=10,
                           maximize=problem.maximize,
                           bounder=problem.bounder,
                           num_trials=1,
                           num_trial_evaluations=5000,
                           max_evaluations=100000)

    pop_size, selector, replacer, crossover, mutator, args = mec.interpret_candidate(final_pop[0].can
    print('Best Fitness: {0}'.format(final_pop[0].fitness))
    print('Population Size: {0}'.format(pop_size))
    print('Selector: {0}'.format(selector.__name__))
    print('Replacer: {0}'.format(replacer.__name__))
    print('Crossover: {0}'.format(crossover.__name__))
    print('Mutator: {0}'.format(mutator.__name__))
    print('Parameters:')
    for key in args:
        print('    {0}: {1}'.format(key, args[key]))
    print('Actual Evaluations Used: {0}'.format(mec._meta_evaluations))
```

4.4 Micro-Evolutionary Computation

Another approach that has been successfully applied to some difficult problems is to use many small-population EC's for small numbers of evaluations in succession. Each succeeding EC is seeded with the best solution from the previous run. This is somewhat akin to a random-restart hill-climbing approach, except that information about the best solution so far is passed along during each restart. [download]

```
from inspyred import ec
from inspyred.ec import terminators

class MicroEC(ec.EvolutionaryComputation):
    def __init__(self, random):
        ec.EvolutionaryComputation.__init__(self, random)

    def evolve(self, generator, evaluator, pop_size=10, seeds=[], maximize=True, bounder=ec.Bounder()
        self._kwargs = args
        self._kwargs['_ec'] = self
        self.termination_cause = None
        self.generator = generator
        self.evaluator = evaluator
        self.bounder = bounder
        self.maximize = maximize
        self.population = []
        self.archive = []
        self.num_generations = 0
        self.num_evaluations = 0
        microseeds = seeds
        args.setdefault('min_diversity', 0.05)
        while not self._should_terminate(self.population, self.num_generations, self.num_evaluations)
            microec = ec.EvolutionaryComputation(self._random)
            microec.selector = self.selector
            microec.variator = self.variator
            microec.replacer = self.replacer
            microec.terminator = terminators.diversity_termination
            result = microec.evolve(generator=generator, evaluator=evaluator,
                                    pop_size=pop_size, seeds=microseeds,
                                    maximize=maximize, **args)
            result.sort(reverse=True)
            microseeds = [result[0].candidate]
            self.population = list(result)
            self.num_evaluations += microec.num_evaluations

            # Migrate individuals.
            self.population = self.migrator(random=self._random,
                                            population=self.population,
                                            args=self._kwargs)

            # Archive individuals.
            pop_copy = list(self.population)
            arc_copy = list(self.archive)
            self.archive = self.archiver(random=self._random, archive=arc_copy,
                                         population=pop_copy, args=self._kwargs)

            self.num_generations += microec.num_generations
            if isinstance(self.observer, (list, tuple)):
                for obs in self.observer:
                    obs(population=self.population, num_generations=self.num_generations,
                        num_evaluations=self.num_evaluations, args=self._kwargs)
            else:
                self.observer(population=self.population, num_generations=self.num_generations,
                              num_evaluations=self.num_evaluations, args=self._kwargs)
        return self.population

if __name__ == '__main__':
    import random
    import math
    import time
    from inspyred import ec
    from inspyred.ec import observers
    from inspyred.ec import terminators
    from inspyred.ec import selectors
    from inspyred.ec import replacers
    from inspyred.ec import variators
    from inspyred.ec import archivers


    def rastrigin_generator(random, args):
        return [random.uniform(-5.12, 5.12) for _ in range(2)]

    def rastrigin_evaluator(candidates, args):
        fitness = []
        for cand in candidates:
            fitness.append(10 * len(cand) + sum([x**2 - 10 * (math.cos(2*math.pi*x)) for x in cand]))
        return fitness

    prng = random.Random()
    prng.seed(time.time())
    micro = MicroEC(prng)
    micro.selector = selectors.tournament_selection
    micro.replacer = replacers.steady_state_replacement
    micro.variator = [variators.uniform_crossover, variators.gaussian_mutation]
    micro.archiver = archivers.best_archiver
    micro.observer = observers.stats_observer
    micro.terminator = terminators.evaluation_termination
    final_pop = micro.evolve(rastrigin_generator,
                             rastrigin_evaluator,
                             pop_size=10,
                             maximize=False,
                             bounder=ec.Bounder(-5.12, 5.12),
                             max_evaluations=3000,
                             num_selected=2,
                             gaussian_stdev=0.1)

    print('Actual evaluations: {0}'.format(micro.num_evaluations))

    for p in micro.archive:
        print p
```

4.5 Network Migrator

The following custom migrator is a callable class (because the migrator must behave like a callback function) that allows solutions to migrate from one network machine to another. It is assumed that the EC islands are running on the given IP:port combinations. [download]

```
import sys
import socket
import pickle
import threading
import collections
import SocketServer

class NetworkMigrator(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    """Defines a migration function across a network.

    This callable class acts as a migration function that
    allows candidate solutions to migrate from one population
    to another via TCP/IP connections.

    The migrator is constructed by specifying the IP address
    of the server (hosting the population from which individuals
    emigrate) as an IP-port tuple and the addresses of the clients
    (hosting the populations to which individuals from the server
    immigrate) as a list of IP-port tuples. The ``max_migrants``
    parameter specifies the size of the queue of migrants waiting
    to immigrate to the server from the clients; the newest migrants
    replace older ones in the queue.

    Note: In order to use this migration operator, individuals
    must be pickle-able.

    The following is an example of the use of this operator::

        m = NetworkMigrator(('192.168.1.10', 25125),
                            [('192.168.1.11', 12345), ('192.168.1.12', 54321)],
                            max_migrants=3)

    Since the NetworkMigrator object is a server, it should always
    call the ``shutdown()`` method when it is no longer needed, in
    order to give back its resources.

    Public Attributes:

    - *client_addresses* -- the list of IP address tuples
      (IP, port) to which individuals should migrate
    - *migrants* -- the deque of migrants (of maximum size
      specified by ``max_migrants``) waiting to immigrate
      to client populations

    """
    def __init__(self, server_address, client_addresses, max_migrants=1):
        self._lock = threading.Lock()
        SocketServer.TCPServer.__init__(self, server_address, None)
        self.client_addresses = client_addresses
        self.migrants = collections.deque(maxlen=max_migrants)
        t = threading.Thread(target=self.serve_forever)
        t.setDaemon(True)
        t.start()
        self.__name__ = self.__class__.__name__

    def finish_request(self, request, client_address):
        try:
            rbufsize = -1
            wbufsize = 0
            rfile = request.makefile('rb', rbufsize)
            wfile = request.makefile('wb', wbufsize)

            pickle_data = rfile.readline().strip()
            migrant = pickle.loads(pickle_data)
            with self._lock:
                self.migrants.append(migrant)

            if not wfile.closed:
                wfile.flush()
            wfile.close()
            rfile.close()
        finally:
            sys.exc_traceback = None

    def __call__(self, random, population, args):
        """Perform the migration.

        This function serves as the migration operator. Here, a random address
        is chosen from the ``client_addresses`` list, and a random individual
        is chosen from the population to become the migrant. A socket is opened
        to the chosen client address, and the chosen migrant is pickled and
        sent to the NetworkMigrator object running at the client address. Then
        the migrant queue on the current machine is queried for a migrant
        to replace the one sent. If one is found, it replaces the newly
        migrated individual; otherwise, the individual remains in the population.

        Any immigrants may also be re-evaluated before insertion into the
        current population by setting the ``evaluate_migrant`` keyword
        argument in ``args`` to True. This is useful if the evaluation
        functions in different populations are different and we want to compare
        "apples to apples," as they say.

        Arguments:

        - *random* -- the random number generator object
        - *population* -- the population of Individuals
        - *args* -- a dictionary of keyword arguments

        Optional keyword arguments in the ``args`` parameter:

        - *evaluate_migrant* -- whether to re-evaluate the immigrant (default False)

        """
        evaluate_migrant = args.setdefault('evaluate_migrant', False)
        client_address = random.choice(self.client_addresses)
        migrant_index = random.randint(0, len(population) - 1)
        pickle_data = pickle.dumps(population[migrant_index])
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        try:
            sock.connect(client_address)
            sock.send(pickle_data + '\n')
        finally:
            sock.close()
        migrant = None
        with self._lock:
            if len(self.migrants) > 0:
                migrant = self.migrants.popleft()
        if migrant is not None:
            if evaluate_migrant:
                fit = args._ec.evaluator([migrant], args)
                migrant.fitness = fit[0]
                args._ec.num_evaluations += 1
            population[migrant_index] = migrant
        return population

    def __str__(self):
        return str(self.migrants)
```

CHAPTER

FIVE

LIBRARY REFERENCE

This chapter provides a complete reference to all of the functionality included in inspyred.

5.1 Evolutionary Computation

5.1.1 `ec` – Evolutionary computation framework

This module provides a framework for creating evolutionary computations.

class `inspyred.ec.`**`Bounder`**`(`*lower_bound=None*, *upper_bound=None*`)`

Defines a basic bounding function for numeric lists.

This callable class acts as a function that bounds a numeric list between the lower and upper bounds specified. These bounds can be single values or lists of values. For instance, if the candidate is composed of five values, each of which should be bounded between 0 and 1, you can say `Bounder([0, 0, 0, 0, 0], [1, 1, 1, 1, 1])` or just `Bounder(0, 1)`. If either the `lower_bound` or `upper_bound` argument is `None`, the Bounder leaves the candidate unchanged (which is the default behavior).

As an example, if the bounder above were used on the candidate `[0.2, -0.1, 0.76, 1.3, 0.4]`, the resulting bounded candidate would be `[0.2, 0, 0.76, 1, 0.4]`.

A bounding function is necessary to ensure that all evolutionary operators respect the legal bounds for candidates. If the user is using only custom operators (which would be aware of the problem constraints), then those can obviously be tailored to enforce the bounds on the candidates themselves. But the built-in operators make only minimal assumptions about the candidate solutions. Therefore, they must rely on an external bounding function that can be user-specified (so as to contain problem-specific information).

In general, a user-specified bounding function must accept two arguments: the candidate to be bounded and the keyword argument dictionary. Typically, the signature of such a function would be the following:

```
bounded_candidate = bounding_function(candidate, args)
```

This function should return the resulting candidate after bounding has been performed.

Public Attributes:

- *lower_bound* – the lower bound for a candidate
- *upper_bound* – the upper bound for a candidate

class `inspyred.ec.`**`DiscreteBounder`**`(`*values*`)`

Defines a basic bounding function for numeric lists of discrete values.

This callable class acts as a function that bounds a numeric list to a set of legitimate values. It does this by resolving a given candidate value to the nearest legitimate value that can be attained. In the event that a candidate value is the same distance to multiple legitimate values, the legitimate value appearing earliest in the list will be used.

For instance, if `[1, 4, 8, 16]` was used as the *values* parameter, then the candidate `[6, 10, 13, 3, 4, 0, 1, 12, 2]` would be bounded to `[4, 8, 16, 4, 4, 1, 1, 8, 1]`.

Public Attributes:

- *values* – the set of attainable values
- *lower_bound* – the smallest attainable value
- *upper_bound* – the largest attainable value

class `inspyred.ec.`**`Individual`**(*candidate=None*, *maximize=True*)

Represents an individual in an evolutionary computation.

An individual is defined by its candidate solution and the fitness (or value) of that candidate solution. Individuals can be compared with one another by using <, <=, >, and >=. In all cases, such comparisons are made using the individuals' fitness values. The `maximize` attribute is respected in all cases, so it is better to think of, for example, < (less-than) to really mean "worse than" and > (greater-than) to mean "better than". For instance, if individuals a and b have fitness values 2 and 4, respectively, and if `maximize` were `True`, then a < b would be true. If `maximize` were `False`, then a < b would be false (because a is "better than" b in terms of the fitness evaluation, since we're minimizing).

Note: `Individual` objects are almost always created by the EC, rather than the user. The `evolve` method of the EC also has a `maximize` argument, whose value is passed directly to all created individuals.

Public Attributes:

- *candidate* – the candidate solution
- *fitness* – the value of the candidate solution
- *birthdate* – the system time at which the individual was created
- *maximize* – Boolean value stating use of maximization

exception `inspyred.ec.`**`Error`**

An empty base exception.

exception `inspyred.ec.`**`EvolutionExit`**

An exception that may be raised and caught to end the evolution.

This is an empty exception class that can be raised by the user at any point in the code and caught outside of the `evolve` method.

Note: Be aware that ending the evolution in such a way will almost certainly produce an erroneous population (e.g., not all individuals will have been reevaluated, etc.). However, this approach can be viable if solutions have been archived such that the current population is not of critical importance.

class `inspyred.ec.`**`EvolutionaryComputation`**(*random*)

Represents a basic evolutionary computation.

This class encapsulates the components of a generic evolutionary computation. These components are the selection mechanism, the variation operators, the replacement mechanism, the migration scheme, the archival mechanism, the terminators, and the observers.

The `observer`, `terminator`, and `variator` attributes may be specified as lists of such operators. In the case of the `observer`, all elements of the list will be called in sequence during the observation phase. In the case of the `terminator`, all elements of the list will be combined via logical `or` and, thus, the evolution will terminate if any of the terminators return True. Finally, in the case of the `variator`, the elements of the list will be applied one after another in pipeline fashion, where the output of one variator is used as the input to the next.

Public Attributes:

- *selector* – the selection operator (defaults to `default_selection`)
- *variator* – the (possibly list of) variation operator(s) (defaults to `default_variation`)
- *replacer* – the replacement operator (defaults to `default_replacement`)
- *migrator* – the migration operator (defaults to `default_migration`)
- *archiver* – the archival operator (defaults to `default_archiver`)
- *observer* – the (possibly list of) observer(s) (defaults to `default_observer`)
- *terminator* – the (possibly list of) terminator(s) (defaults to `default_termination`)
- *logger* – the logger to use (defaults to the logger 'inspyred.ec')

The following attributes do not have legitimate values until after the `evolve` method executes:

- *termination_cause* – the name of the function causing `evolve` to terminate, in the event that multiple terminators are used
- *generator* – the generator function passed to `evolve`
- *evaluator* – the evaluator function passed to `evolve`
- *bounder* – the bounding function passed to `evolve`
- *maximize* – Boolean stating use of maximization passed to `evolve`
- *archive* – the archive of individuals
- *population* – the population of individuals
- *num_evaluations* – the number of fitness evaluations used
- *num_generations* – the number of generations processed

Note that the attributes above are, in general, not intended to be modified by the user. (They are intended for the user to query during or after the `evolve` method's execution.) However, there may be instances where it is necessary to modify them within other functions. This is possible to do, but it should be the exception, rather than the rule.

If logging is desired, the following basic code segment can be used in the `main` or calling scope to accomplish that:

```
import logging
logger = logging.getLogger('inspyred.ec')
logger.setLevel(logging.DEBUG)
file_handler = logging.FileHandler('inspyred.log', mode='w')
file_handler.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)
```

Protected Attributes:

- *_random* – the random number generator object
- *_kwargs* – the dictionary of keyword arguments initialized from the *args* parameter in the *evolve* method

evolve(*generator*, *evaluator*, *pop_size=100*, *seeds=None*, *maximize=True*, *bounder=None*, ***args*)
Perform the evolution.

This function creates a population and then runs it through a series of evolutionary epochs until the terminator is satisfied. The general outline of an epoch is selection, variation, evaluation, replacement, migration, archival, and observation. The function returns a list of elements of type `Individual` representing the individuals contained in the final population.

Arguments:

- *generator* – the function to be used to generate candidate solutions
- *evaluator* – the function to be used to evaluate candidate solutions
- *pop_size* – the number of Individuals in the population (default 100)
- *seeds* – an iterable collection of candidate solutions to include in the initial population (default None)
- *maximize* – Boolean value stating use of maximization (default True)
- *bounder* – a function used to bound candidate solutions (default None)
- *args* – a dictionary of keyword arguments

The *bounder* parameter, if left as `None`, will be initialized to a default `Bounder` object that performs no bounding on candidates. Note that the *_kwargs* class variable will be initialized to the *args* parameter here. It will also be modified to include the following 'built-in' keyword argument:

- *_ec* – the evolutionary computation (this object)

class `inspyred.ec.`**`GA`**(*random*)
Evolutionary computation representing a canonical genetic algorithm.

This class represents a genetic algorithm which uses, by default, rank selection, *n*-point crossover, bit-flip mutation, and generational replacement. In the case of bit-flip mutation, it is expected that each candidate solution is a `Sequence` of binary values.

Optional keyword arguments in `evolve` args parameter:

- *num_selected* – the number of individuals to be selected (default len(population))
- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *num_crossover_points* – the *n* crossover points used (default 1)
- *mutation_rate* – the rate at which mutation is performed (default 0.1)
- *num_elites* – number of elites to consider (default 0)

class `inspyred.ec.`**`ES`**(*random*)
Evolutionary computation representing a canonical evolution strategy.

This class represents an evolution strategy which uses, by default, the default selection (i.e., all individuals are selected), an internal adaptive mutation using strategy parameters, and 'plus' replacement. It is expected that each candidate solution is a `Sequence` of real values.

The candidate solutions to an ES are augmented by strategy parameters of the same length (using `inspyred.ec.generators.strategize`). These strategy parameters are evolved along with the candidates and are used as the mutation rates for each element of the candidates. The evaluator is modified internally to use only the actual candidate elements (rather than also the strategy parameters), so normal evaluator functions may be used seamlessly.

Optional keyword arguments in `evolve` args parameter:

- *tau* – a proportionality constant (default None)

- *tau_prime* – a proportionality constant (default None)
- *epsilon* – the minimum allowed strategy parameter (default 0.00001)

If *tau* is `None`, it will be set to `1 / sqrt(2 * sqrt(n))`, where `n` is the length of a candidate. If *tau_prime* is `None`, it will be set to `1 / sqrt(2 * n)`. The strategy parameters are updated as follows:

$$\sigma_i' = \sigma_i + e^{\tau \cdot N(0,1) + \tau' \cdot N(0,1)}$$
$$\sigma_i' = max(\sigma_i', \epsilon)$$

class `inspyred.ec.`**`EDA`**`(`*random*`)`

Evolutionary computation representing a canonical estimation of distribution algorithm.

This class represents an estimation of distribution algorithm which uses, by default, truncation selection, an internal estimation of distribution variation, and generational replacement. It is expected that each candidate solution is a `Sequence` of real values.

The variation used here creates a statistical model based on the set of candidates. The offspring are then generated from this model. This function also makes use of the bounder function as specified in the EC's `evolve` method.

Optional keyword arguments in `evolve` args parameter:

- *num_selected* – the number of individuals to be selected (default len(population)/2)
- *num_offspring* – the number of offspring to create (default len(population))
- *num_elites* – number of elites to consider (default 0)

class `inspyred.ec.`**`DEA`**`(`*random*`)`

Evolutionary computation representing a differential evolutionary algorithm.

This class represents a differential evolutionary algorithm which uses, by default, tournament selection, heuristic crossover, Gaussian mutation, and steady-state replacement. It is expected that each candidate solution is a `Sequence` of real values.

Optional keyword arguments in `evolve` args parameter:

- *num_selected* – the number of individuals to be selected (default 2)
- *tournament_size* – the tournament size (default 2)
- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *mutation_rate* – the rate at which mutation is performed (default 0.1)
- *gaussian_mean* – the mean used in the Gaussian function (default 0)
- *gaussian_stdev* – the standard deviation used in the Gaussian function (default 1)

class `inspyred.ec.`**`SA`**`(`*random*`)`

Evolutionary computation representing simulated annealing.

This class represents a simulated annealing algorithm. It accomplishes this by using default selection (i.e., all individuals are parents), Gaussian mutation, and simulated annealing replacement. It is expected that each candidate solution is a `Sequence` of real values. Consult the documentation for the `simulated_annealing_replacement` for more details on the keyword arguments listed below.

Note: The `pop_size` parameter to `evolve` will always be set to 1, even if a different value is passed.

Optional keyword arguments in `evolve` args parameter:

- *temperature* – the initial temperature

- *cooling_rate* – a real-valued coefficient in the range (0, 1) by which the temperature should be reduced
- *mutation_rate* – the rate at which mutation is performed (default 0.1)
- *gaussian_mean* – the mean used in the Gaussian function (default 0)
- *gaussian_stdev* – the standard deviation used in the Gaussian function (default 1)

5.1.2 `emo` – Evolutionary multiobjective optimization

This module provides the framework for making multiobjective evolutionary computations.

class `inspyred.ec.emo.`**`NSGA2`**`(`*random*`)`

Evolutionary computation representing the nondominated sorting genetic algorithm.

This class represents the nondominated sorting genetic algorithm (NSGA-II) of Kalyanmoy Deb et al. It uses nondominated sorting with crowding for replacement, binary tournament selection to produce *population size* children, and a Pareto archival strategy. The remaining operators take on the typical default values but they may be specified by the designer.

class `inspyred.ec.emo.`**`PAES`**`(`*random*`)`

Evolutionary computation representing the Pareto Archived Evolution Strategy.

This class represents the Pareto Archived Evolution Strategy of Joshua Knowles and David Corne. It is essentially a (1+1)-ES with an adaptive grid archive that is used as a part of the replacement process.

class `inspyred.ec.emo.`**`Pareto`**`(`*values=None*, *maximize=True*`)`

Represents a Pareto multiobjective solution.

A Pareto solution is a set of multiobjective values that can be compared to other Pareto values using Pareto preference. This means that a solution dominates, or is better than, another solution if it is better than or equal to the other solution in all objectives and strictly better in at least one objective.

Since some problems may mix maximization and minimization among different objectives, an optional *maximize* parameter may be passed upon construction of the Pareto object. This parameter may be a list of Booleans of the same length as the set of objective values. If this parameter is used, then the *maximize* parameter of the evolutionary computation's `evolve` method should be left as the default True value in order to avoid confusion. (Setting the *evolve*'s parameter to False would essentially invert all of the Booleans in the Pareto *maximize* list.) So, if all objectives are of the same type (either maximization or minimization), then it is best simply to use the *maximize* parameter of the *evolve* method and to leave the *maximize* parameter of the Pareto initialization set to its default True value. However, if the objectives are mixed maximization and minimization, it is best to leave the `evolve`'s *maximize* parameter set to its default True value and specify the Pareto's *maximize* list to the appropriate Booleans.

The typical usage is as follows:

```python
@inspyred.ec.evaluators.evaluator
def my_evaluator(candidate, args):
    obj1 = 1 # Calculate objective 1
    obj2 = 2 # Calculate objective 2
    obj3 = 3 # Calculate objective 3
    return emo.Pareto([obj1, obj2, obj3])
```

5.1.3 `analysis` – Optimization result analysis

This module provides analysis methods for the results of evolutionary computations.

`inspyred.ec.analysis.`**`allele_plot`**(*filename*, *normalize=False*, *alleles=None*, *generations=None*)

Plot the alleles from each generation from the individuals file.

This function creates a plot of the individual allele values as they change through the generations. It creates three subplots, one for each of the best, median, and average individual. The best and median individuals are chosen using the fitness data for each generation. The average individual, on the other hand, is actually an individual created by averaging the alleles within a generation. This function requires the pylab library.

Note: This function only works for single-objective problems.

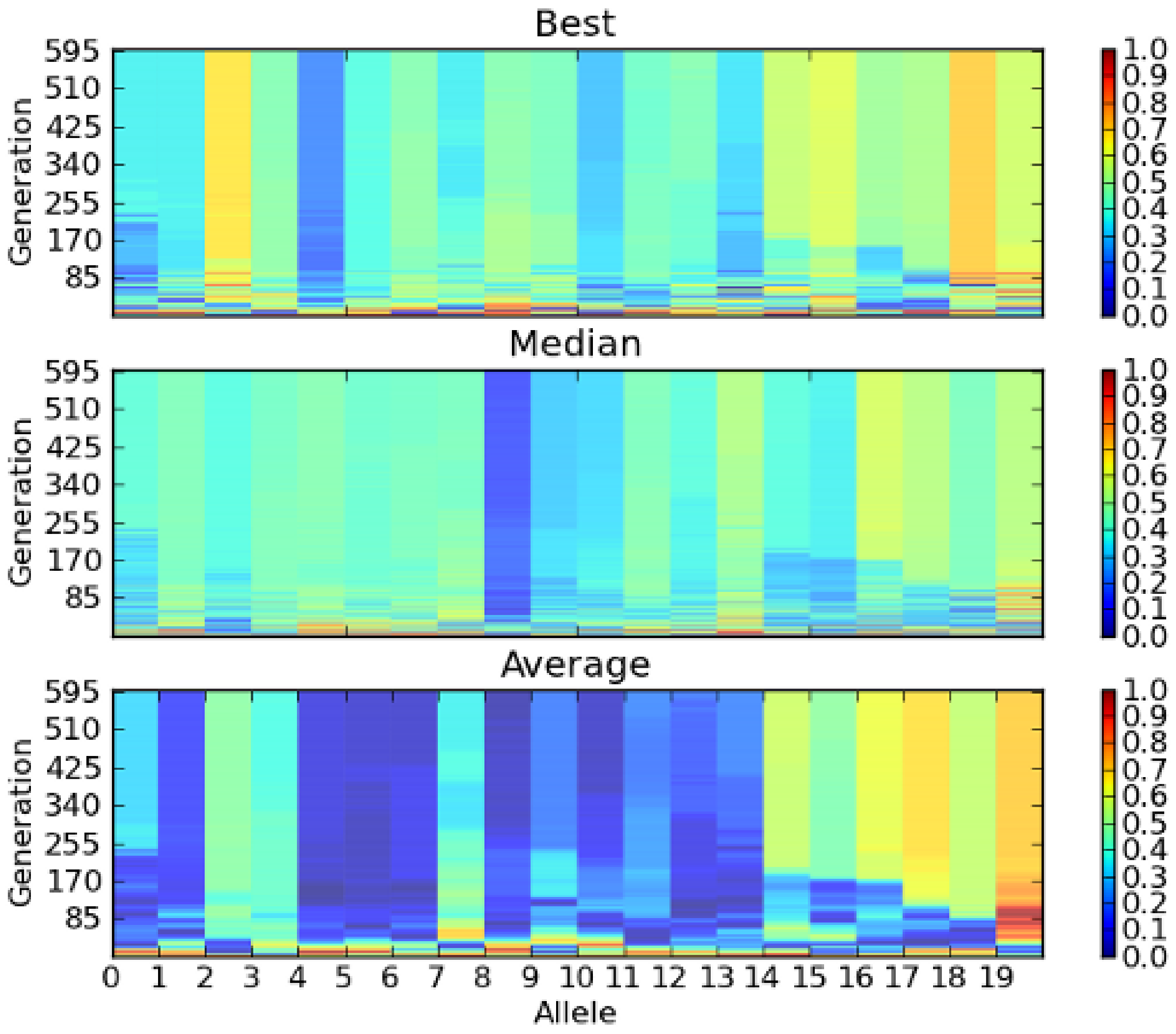


Figure 5.1: An example image saved from the `allele_plot` function.

Arguments:

- *filename* – the name of the individuals file produced by the file_observer
- *normalize* – Boolean value stating whether allele values should be normalized before plotting (default False)
- *alleles* – a list of allele index values that should be plotted (default None)

- *generations* – a list of generation numbers that should be plotted (default None)

If *alleles* is `None`, then all alleles are plotted. Similarly, if *generations* is `None`, then all generations are plotted.

`inspyred.ec.analysis.`**`fitness_statistics`**`(`*population*`)`

Return the basic statistics of the population's fitness values.

This function returns a dictionary containing the "best", "worst", "mean", "median", and "std" fitness values in the population. ("std" is the standard deviation.) A typical usage would be similar to the following:

```
stats = fitness_statistics(population)
print(stats['best'])
print(stats['worst'])
print(stats['mean'])
print(stats['median'])
print(stats['std'])
```

Note: This function makes use of the numpy library for calculations. If that library is not found, it attempts to complete the calculations internally. However, this second attempt will fail for multiobjective fitness values and will return `nan` for the mean, median, and standard deviation.

Arguments:

- *population* – the population of individuals

`inspyred.ec.analysis.`**`generation_plot`**`(`*filename*, *errorbars=True*`)`

Plot the results of the algorithm using generation statistics.

This function creates a plot of the generation fitness statistics (best, worst, median, and average). This function requires the pylab and matplotlib libraries.

Note: This function only works for single-objective problems.

Arguments:

- *filename* – the name of the statistics file produced by the file_observer
- *errorbars* – Boolean value stating whether standard error bars should be drawn (default True)

`inspyred.ec.analysis.`**`hypervolume`**`(`*pareto_set*, *reference_point=None*`)`

Calculates the hypervolume by slicing objectives (HSO).

This function calculates the hypervolume (or S-measure) of a nondominated set using the Hypervolume by Slicing Objectives (HSO) procedure of While, et al. (IEEE CEC 2005). The *pareto_set* should be a list of lists of objective values. The *reference_point* may be specified or it may be left as the default value of None. In that case, the reference point is calculated to be the maximum value in the set for all objectives (the ideal point). This function assumes that objectives are to be maximized.

Arguments:

- *pareto_set* – the list or lists of objective values comprising the Pareto front
- *reference_point* – the reference point to be used (default None)

5.1.4 `utilities` – Optimization utility functions

This module provides utility classes and decorators for evolutionary computations.

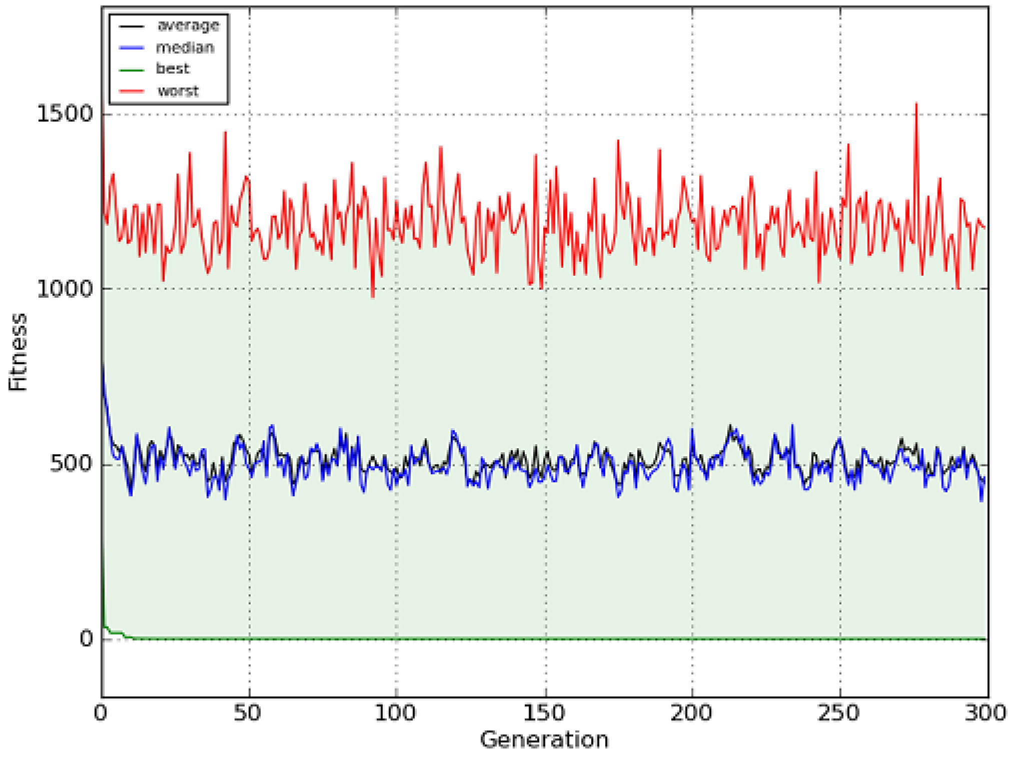


Figure 5.2: An example image saved from the `generation_plot` function (without error bars).

class `inspyred.ec.utilities.`**`Objectify`**(*func*)

Create an "objectified" version of a function.

This function allows an ordinary function passed to it to become essentially a callable instance of a class. For inspyred, this means that evolutionary operators (selectors, variators, replacers, etc.) can be created as normal functions and then be given the ability to have attributes *that are specific to the object*. Python functions can always have attributes without employing any special mechanism, but those attributes exist for the function, and there is no way to create a new "object" except by implementing a new function with the same functionality. This class provides a way to "objectify" the same function multiple times in order to provide each "object" with its own set of independent attributes.

The attributes that are created on an objectified function are passed into that function via the ubiquitous `args` variable in inspyred. Any user-specified attributes are added to the `args` dictionary and replace any existing entry if necessary. If the function modifies those entries in the dictionary (e.g., when dynamically modifying parameters), the corresponding attributes are modified as well.

Essentially, a local copy of the `args` dictionary is created into which the attributes are inserted. This modified local copy is then passed to the function. After the function returns, the values of the attributes from the dictionary are retrieved and are used to update the class attributes.

The typical usage is as follows:

```
def typical_function(*args, **kwargs):
    # Implementation of typical function
    pass

fun_one = Objectify(typical_function)
fun_two = Objectify(typical_function)
fun_one.attribute = value_one
fun_two.attribute = value_two
```

`inspyred.ec.utilities.`**`memoize`**(*func=None*, *maxlen=None*)

Cache a function's return value each time it is called.

This function serves as a function decorator to provide a caching of evaluated fitness values. If called later with the same arguments, the cached value is returned instead of being re-evaluated.

This decorator assumes that candidates are individually pickleable, and their pickled values are used for hashing into a dictionary. It should be used when evaluating an *expensive* fitness function to avoid costly re-evaluation of those fitnesses. The typical usage is as follows:

```
@memoize
def expensive_fitness_function(candidates, args):
    # Implementation of expensive fitness calculation
    pass
```

It is also possible to provide the named argument *maxlen*, which specifies the size of the memoization cache to use. (If *maxlen* is `None`, then an unbounded cache is used.) Once the size of the cache has reached *maxlen*, the oldest element is replaced by the newest element in order to keep the size constant. This usage is as follows:

```
@memoize(maxlen=100)
def expensive_fitness_function(candidates, args):
    # Implementation of expensive fitness calculation
    pass
```

Warning: The `maxlen` parameter must be passed as a named keyword argument, or an `AttributeError` will be raised (e.g., saying `@memoize(100)` will cause an error).

5.1.5 Operators

An evolutionary computation is composed of many parts:

- an archiver – stores solutions separate from the population (e.g., in a multiobjective EC)
- an evaluator – measures the fitness of candidate solutions; problem-dependent
- a generator – creates new candidate solutions; problem-dependent
- a migrator – moves individuals to other populations (in the case of distributed ECs)
- observers – view the progress of an EC in operation; may be a list of observers
- a replacer – determines the survivors of a generation
- a selector – determines the parents of a generation
- terminators – determine whether the evolution should stop; may be a list of terminators
- variators – modify candidate solutions; may be a list of variators

Each of these parts may be specified to create custom ECs to suit particular problems.

`archivers` – Solution archival methods

This module provides pre-defined archivers for evoluationary computations.

All archiver functions have the following arguments:

- *random* – the random number generator object
- *population* – the population of individuals
- *archive* – the current archive of individuals
- *args* – a dictionary of keyword arguments

Each archiver function returns the updated archive.

Note: The *population* is really a shallow copy of the actual population of the evolutionary computation. This means that any activities like sorting will not affect the actual population.

`inspyred.ec.archivers.`**`adaptive_grid_archiver`**(*random*, *population*, *archive*, *args*)

Archive only the best individual(s) using a fixed size grid.

This function archives the best solutions by using a fixed-size grid to determine which existing solutions should be removed in order to make room for new ones. This archiver is designed specifically for use with the Pareto Archived Evolution Strategy (PAES).

Optional keyword arguments in args:

- *max_archive_size* – the maximum number of individuals in the archive (default len(population))
- *num_grid_divisions* – the number of grid divisions (default 1)

`inspyred.ec.archivers.`**`best_archiver`**(*random*, *population*, *archive*, *args*)

Archive only the best individual(s).

This function archives the best solutions and removes inferior ones. If the comparison operators have been overloaded to define Pareto preference (as in the `Pareto` class), then this archiver will form a Pareto archive.

`inspyred.ec.archivers.`**`default_archiver`**`(`*random*, *population*, *archive*, *args*`)`

Do nothing.

This function just returns the existing archive (which is probably empty) with no changes.

`inspyred.ec.archivers.`**`population_archiver`**`(`*random*, *population*, *archive*, *args*`)`

Archive the current population.

This function replaces the archive with the individuals of the current population.

`evaluators` – Fitness evaluation methods

Evaluator functions are problem-specific. This module provides pre-defined evaluators for evolutionary computations.

All evaluator functions have the following arguments:

- *candidates* – the candidate solutions
- *args* – a dictionary of keyword arguments

`inspyred.ec.evaluators.`**`evaluator`**`(`*evaluate*`)`

Return an inspyred evaluator function based on the given function.

This function generator takes a function that evaluates only one candidate. The generator handles the iteration over each candidate to be evaluated.

The given function `evaluate` must have the following signature:

```
fitness = evaluate(candidate, args)
```

This function is most commonly used as a function decorator with the following usage:

```
@evaluator
def evaluate(candidate, args):
    # Implementation of evaluation
    pass
```

The generated function also contains an attribute named `single_evaluation` which holds the original evaluation function. In this way, the original single-candidate function can be retrieved if necessary.

`inspyred.ec.evaluators.`**`parallel_evaluation_mp`**`(`*candidates*, *args*`)`

Evaluate the candidates in parallel using `multiprocessing`.

This function allows parallel evaluation of candidate solutions. It uses the standard multiprocessing library to accomplish the parallelization. The function assigns the evaluation of each candidate to its own job, all of which are then distributed to the available processing units.

Note: All arguments to the evaluation function must be pickleable. Those that are not will not be sent through the `args` variable and will be unavailable to your function.

Required keyword arguments in args:

- *mp_evaluator* – actual evaluation function to be used (This function should have the same signature as any other inspyred evaluation function.)

Optional keyword arguments in args:

- *mp_nprocs* – number of processors that will be used (default machine cpu count)

`inspyred.ec.evaluators.`**`parallel_evaluation_pp`**`(`*candidates*, *args*`)`

Evaluate the candidates in parallel using Parallel Python.

This function allows parallel evaluation of candidate solutions. It uses the Parallel Python (pp) library to accomplish the parallelization. This library must already be installed in order to use this function. The function assigns the evaluation of each candidate to its own job, all of which are then distributed to the available processing units.

Note: All arguments to the evaluation function must be pickleable. Those that are not will not be sent through the `args` variable and will be unavailable to your function.

Required keyword arguments in args:

- *pp_evaluator* – actual evaluation function to be used (This function should have the same signature as any other inspyred evaluation function.)

Optional keyword arguments in args:

- *pp_dependencies* – tuple of functional dependencies of the serial evaluator (default ())
- *pp_modules* – tuple of modules that must be imported for the functional dependencies (default ())
- *pp_servers* – tuple of servers (on a cluster) that will be used for parallel processing (default ("*",))
- *pp_secret* – string representing the secret key needed to authenticate on a worker node (default "inspyred")
- *pp_nprocs* – integer representing the number of worker processes to start on the local machine (default "autodetect", which sets it to the number of processors in the system)

For more information about these arguments, please consult the documentation for Parallel Python.

`generators` – Solution generation methods

Generator functions are problem-specific. They are used to create the initial set of candidate solutions needed by the evolutionary computation.

All generator functions have the following arguments:

- *random* – the random number generator object
- *args* – a dictionary of keyword arguments

class `inspyred.ec.generators.`**`diversify`**`(`*generator*`)`

Ensure uniqueness of candidates created by a generator.

This function decorator is used to enforce uniqueness of candidates created by a generator. The decorator maintains a list of previously created candidates, and it ensures that new candidates are unique by checking a generated candidate against that list, regenerating if a duplicate is found. The typical usage is as follows:

```
@diversify
def generator_function(random, args):
    # Normal generator function
    pass
```

If a list of seeds is used, then these can be specified prior to the generator's use by saying the following:

```
@diversify
def generator_function(random, args):
    # Normal generator function
    pass
generator_function.candidates = seeds
```

`inspyred.ec.generators.`**`strategize`**`(`*generator*`)`

Add strategy parameters to candidates created by a generator.

This function decorator is used to provide a means of adding strategy parameters to candidates created by a generator. The generator function is modifed to extend the candidate with `len(candidate)` strategy parameters (one per candidate element). Each strategy parameter is initialized to a random value in the range [0, 1]. The typical usage is as follows:

```
@strategize
def generator_function(random, args):
    # Normal generator function
    pass
```

`migrators` – Solution migration methods

This module provides pre-defined migrators for evolutionary computations.

All migrator functions have the following arguments:

- *random* – the random number generator object
- *population* – the population of Individuals
- *args* – a dictionary of keyword arguments

Each migrator function returns the updated population.

Migrator functions would typically be used for multi-population approaches, such as island-model evolutionary computations. They provide a means for individuals to be transferred from one population to another during the evolutionary process.

class `inspyred.ec.migrators.`**`MultiprocessingMigrator`**(*max_migrants=1*)

Migrate among processes on the same machine.

This callable class allows individuals to migrate from one process to another on the same machine. It maintains a queue of migrants whose maximum length can be fixed via the `max_migrants` parameter in the constructor. If the number of migrants in the queue reaches this value, new migrants are not added until earlier ones are consumed. The unreliability of a multiprocessing environment makes it difficult to provide guarantees. However, migrants are theoretically added and consumed at the same rate, so this value should determine the "freshness" of individuals, where smaller queue sizes provide more recency.

An optional keyword argument in `args` requires the migrant to be evaluated by the current evolutionary computation before being inserted into the population. This can be important when different populations use different evaluation functions and you need to be able to compare "apples with apples," so to speak.

Optional keyword arguments in args:

- *evaluate_migrant* – should new migrants be evaluated before adding them to the population (default False)

`inspyred.ec.migrators.`**`default_migration`**(*random*, *population*, *args*)

Do nothing.

This function just returns the existing population with no changes.

`observers` – Algorithm monitoring methods

This module provides pre-defined observers for evolutionary computations.

All observer functions have the following arguments:

- *population* – the population of Individuals
- *num_generations* – the number of elapsed generations

- *num_evaluations* – the number of candidate solution evaluations
- *args* – a dictionary of keyword arguments

Note: The *population* is really a shallow copy of the actual population of the evolutionary computation. This means that any activities like sorting will not affect the actual population.

---

class `inspyred.ec.observers.`**`EmailObserver`**(*username*, *password*, *server*, *port=587*)

Email the population statistics, individuals, and optional file observer data.

This callable class allows information about the current generation to be emailed to a user. This is useful when dealing with computationally expensive optimization problems where the evolution must progress over hours or days. The `generation_step` attribute can be set to an integer greater than 1 to ensure that emails are only sent on generations that are multiples of the step size.

Note: This function makes use of the `inspyred.ec.analysis.fitness_statistics` function, so it is subject to the same requirements.

A typical instantiation of this class would be the following:

```
import getpass
usr = raw_input("Enter your username: ")
pwd = getpass.getpass("Enter your password: ")
email_observer = EmailObserver(usr, pwd, "my.mail.server")
email_observer.from_address = "me@here.com"
email_observer.to_address = "you@there.com" # or ["you@there.com", "other@somewhere.com"]
email_observer.subject = "My custom subject"
email_observer.generation_step = 10 # Send an email every 10th generation
```

Public Attributes:

- *username* – the mail server username
- *password* – the mail server password
- *server* – the mail server URL or IP address string
- *port* – the mail server port as an integer
- *from_address* – the email address of the sender
- *to_address* – the (possibly list of) email address(es) of the receiver(s)
- *subject* – the subject of the email (default 'inspyred observer report')
- *max_attachment* – the maximum allowable size, in MB, of attachments (default 20 MB)
- *generation_step* – the step size for when a generation's information should be emailed (default 1)

`inspyred.ec.observers.`**`archive_observer`**(*population*, *num_generations*, *num_evaluations*, *args*)

Print the current archive to the screen.

This function displays the current archive of the evolutionary computation to the screen.

`inspyred.ec.observers.`**`best_observer`**(*population*, *num_generations*, *num_evaluations*, *args*)

Print the best individual in the population to the screen.

This function displays the best individual in the population to the screen.

`inspyred.ec.observers.`**`default_observer`**(*population*, *num_generations*, *num_evaluations*, *args*)

Do nothing.

`inspyred.ec.observers.`**`file_observer`**(*population*, *num_generations*, *num_evaluations*, *args*)

Print the output of the evolutionary computation to a file.

This function saves the results of the evolutionary computation to two files. The first file, which by default is named 'inspyred-statistics-file-<timestamp>.csv', contains the basic generational statistics of the population throughout the run (worst, best, median, and average fitness and standard deviation of the fitness values). The second file, which by default is named 'inspyred-individuals-file-<timestamp>.csv', contains every individual during each generation of the run. Both files may be passed to the function as keyword arguments (see below).

The format of each line of the statistics file is as follows:

```
generation number, population size, worst, best, median, average, standard deviation
```

The format of each line of the individuals file is as follows:

```
generation number, individual number, fitness, string representation of candidate
```

Note: This function makes use of the `inspyred.ec.analysis.fitness_statistics` function, so it is subject to the same requirements.

Optional keyword arguments in args:

- *statistics_file* – a file object (default: see text)
- *individuals_file* – a file object (default: see text)

`inspyred.ec.observers.`**`plot_observer`**(*population*, *num_generations*, *num_evaluations*, *args*)

Plot the output of the evolutionary computation as a graph.

This function plots the performance of the EC as a line graph using the pylab library (matplotlib) and numpy. The graph consists of a blue line representing the best fitness, a green line representing the average fitness, and a red line representing the median fitness. It modifies the keyword arguments variable 'args' by including an entry called 'plot_data'.

If this observer is used, the calling script should also import the pylab library and should end the script with

pylab.show()

Otherwise, the program may generate a runtime error.

Note: This function makes use of the pylab and numpy libraries.

`inspyred.ec.observers.`**`population_observer`**(*population*, *num_generations*, *num_evaluations*, *args*)

Print the current population of the evolutionary computation to the screen.

This function displays the current population of the evolutionary computation to the screen in fitness-sorted order.

`inspyred.ec.observers.`**`stats_observer`**(*population*, *num_generations*, *num_evaluations*, *args*)

Print the statistics of the evolutionary computation to the screen.

This function displays the statistics of the evolutionary computation to the screen. The output includes the generation number, the current number of evaluations, the maximum fitness, the minimum fitness, the average fitness, and the standard deviation.

Note: This function makes use of the `inspyred.ec.analysis.fitness_statistics` function, so it is subject to the same requirements.

`replacers` – Survivor replacement methods

This module provides pre-defined replacers for evolutionary computations.

All replacer functions have the following arguments:

- *random* – the random number generator object
- *population* – the population of individuals
- *parents* – the list of parent individuals
- *offspring* – the list of offspring individuals
- *args* – a dictionary of keyword arguments

Each replacer function returns the list of surviving individuals.

`inspyred.ec.replacers.`**`comma_replacement`**(*random*, *population*, *parents*, *offspring*, *args*)

Performs "comma" replacement.

This function performs "comma" replacement, which means that the entire existing population is replaced by the best population-many elements from the offspring. This function makes the assumption that the size of the offspring is at least as large as the original population. Otherwise, the population size will not be constant.

`inspyred.ec.replacers.`**`crowding_replacement`**(*random*, *population*, *parents*, *offspring*, *args*)

Performs crowding replacement as a form of niching.

This function performs crowding replacement, which means that the members of the population are replaced one-at-a-time with each of the offspring. A random sample of *crowding_distance* individuals is pulled from the current population, and the closest individual to the current offspring (where "closest" is determined by the *distance_function*) is replaced by that offspring, if the offspring is better. It is possible for one offspring to replace an earlier offspring in the same generation, given the random sample that is taken of the current survivors for each offspring.

Optional keyword arguments in args:

- *distance_function* – a function that accepts two candidate solutions and returns the distance between them (default Euclidean L2 distance)
- *crowding_distance* – a positive integer representing the number of closest solutions to consider as a "crowd" (default 2)

`inspyred.ec.replacers.`**`default_replacement`**(*random*, *population*, *parents*, *offspring*, *args*)

Performs no replacement, returning the original population.

`inspyred.ec.replacers.`**`generational_replacement`**(*random*, *population*, *parents*, *offspring*, *args*)

Performs generational replacement with optional weak elitism.

This function performs generational replacement, which means that the entire existing population is replaced by the offspring, truncating to the population size if the number of offspring is larger. Weak elitism may also be specified through the *num_elites* keyword argument in args. If this is used, the best *num_elites* individuals in the current population are allowed to survive if they are better than the worst *num_elites* offspring.

Optional keyword arguments in args:

- *num_elites* – number of elites to consider (default 0) `inspyred.ec.replacers.`**`nsga_replacement`**`(`*random*, *population*, *parents*, *offspring*, *args*`)`

Replaces population using the non-dominated sorting technique from NSGA-II.

`inspyred.ec.replacers.`**`paes_replacement`**`(`*random*, *population*, *parents*, *offspring*, *args*`)`

Replaces population using the Pareto Archived Evolution Strategy method.

`inspyred.ec.replacers.`**`plus_replacement`**`(`*random*, *population*, *parents*, *offspring*, *args*`)`

Performs "plus" replacement.

This function performs "plus" replacement, which means that the entire existing population is replaced by the best population-many elements from the combined set of parents and offspring.

`inspyred.ec.replacers.`**`random_replacement`**`(`*random*, *population*, *parents*, *offspring*, *args*`)`

Performs random replacement with optional weak elitism.

This function performs random replacement, which means that the offspring replace random members of the population, keeping the population size constant. Weak elitism may also be specified through the *num_elites* keyword argument in args. If this is used, the best *num_elites* individuals in the current population are allowed to survive if they are better than the worst *num_elites* offspring.

Optional keyword arguments in args:

- *num_elites* – number of elites to consider (default 0)

`inspyred.ec.replacers.`**`simulated_annealing_replacement`**`(`*random*, *population*, *parents*, *offspring*, *args*`)`

Replaces population using the simulated annealing schedule.

This function performs simulated annealing replacement based on a temperature and a cooling rate. These can be specified by the keyword arguments *temperature*, which should be the initial temperature, and *cooling_rate*, which should be the coefficient by which the temperature is reduced. If these keyword arguments are not present, then the function will attempt to base the cooling schedule either on the ratio of evaluations to the maximum allowed evaluations or on the ratio of generations to the maximum allowed generations. Each of these ratios is of the form `(max - current)/max` so that the cooling schedule moves smoothly from 1 to 0.

Optional keyword arguments in args:

- *temperature* – the initial temperature
- *cooling_rate* – a real-valued coefficient in the range (0, 1) by which the temperature should be reduced

`inspyred.ec.replacers.`**`steady_state_replacement`**`(`*random*, *population*, *parents*, *offspring*, *args*`)`

Performs steady-state replacement for the offspring.

This function performs steady-state replacement, which means that the offspring replace the least fit individuals in the existing population, even if those offspring are less fit than the individuals that they replace.

`inspyred.ec.replacers.`**`truncation_replacement`**`(`*random*, *population*, *parents*, *offspring*, *args*`)`

Replaces population with the best of the population and offspring.

This function performs truncation replacement, which means that the entire existing population is replaced by the best from among the current population and offspring, keeping the existing population size fixed. This is similar to so-called "plus" replacement in the evolution strategies literature, except that "plus" replacement considers only parents and offspring for survival. However, if the entire population are parents (which is often the case in evolution strategies), then truncation replacement and plus-replacement are equivalent approaches.

`selectors` – Parent selection methods

This module provides pre-defined selectors for evolutionary computations.

All selector functions have the following arguments:

- *random* – the random number generator object
- *population* – the population of individuals
- *args* – a dictionary of keyword arguments

Each selector function returns the list of selected individuals.

Note: The *population* is really a shallow copy of the actual population of the evolutionary computation. This means that any activities like sorting will not affect the actual population.

`inspyred.ec.selectors.`**`default_selection`**`(`*random*, *population*, *args*`)`

Return the population.

This function acts as a default selection scheme for an evolutionary computation. It simply returns the entire population as having been selected.

`inspyred.ec.selectors.`**`fitness_proportionate_selection`**`(`*random*, *population*, *args*`)`

Return fitness proportionate sampling of individuals from the population.

This function stochastically chooses individuals from the population with probability proportional to their fitness. This is often referred to as "roulette wheel" selection. Note that this selection is not valid for minimization problems.

Optional keyword arguments in args:

- *num_selected* – the number of individuals to be selected (default 1)

`inspyred.ec.selectors.`**`rank_selection`**`(`*random*, *population*, *args*`)`

Return a rank-based sampling of individuals from the population.

This function behaves similarly to fitness proportionate selection, except that it uses the individual's rank in the population, rather than its raw fitness value, to determine its probability. This means that it can be used for both maximization and minimization problems, since higher rank can be defined correctly for both.

Optional keyword arguments in args:

- *num_selected* – the number of individuals to be selected (default 1)

`inspyred.ec.selectors.`**`tournament_selection`**`(`*random*, *population*, *args*`)`

Return a tournament sampling of individuals from the population.

This function selects `num_selected` individuals from the population. It selects each one by using random sampling without replacement to pull `tournament_size` individuals and adds the best of the tournament as its selection. If `tournament_size` is greater than the population size, the population size is used instead as the size of the tournament.

Optional keyword arguments in args:

- *num_selected* – the number of individuals to be selected (default 1)
- *tournament_size* – the tournament size (default 2)

`inspyred.ec.selectors.`**`truncation_selection`**`(`*random*, *population*, *args*`)`

Selects the best individuals from the population.

This function performs truncation selection, which means that only the best individuals from the current population are selected. This is a completely deterministic selection mechanism.

Optional keyword arguments in args:

- *num_selected* – the number of individuals to be selected (default len(population)) `inspyred.ec.selectors.`**`uniform_selection`**(*random*, *population*, *args*)

Return a uniform sampling of individuals from the population.

This function performs uniform selection by randomly choosing members of the population with replacement.

Optional keyword arguments in args:

- *num_selected* – the number of individuals to be selected (default 1)

`terminators` – Algorithm termination methods

This module provides pre-defined terminators for evolutionary computations.

Terminators specify when the evolutionary process should end. All terminators must return a Boolean value where True implies that the evolution should end.

All terminator functions have the following arguments:

- *population* – the population of Individuals
- *num_generations* – the number of elapsed generations
- *num_evaluations* – the number of candidate solution evaluations
- *args* – a dictionary of keyword arguments

Note: The *population* is really a shallow copy of the actual population of the evolutionary computation. This means that any activities like sorting will not affect the actual population.

`inspyred.ec.terminators.`**`average_fitness_termination`**(*population*, *num_generations*, *num_evaluations*, *args*)

Return True if the population's average fitness is near its best fitness.

This function calculates the average fitness of the population, as well as the best fitness. If the difference between those values is less than a specified tolerance, the function returns True.

Optional keyword arguments in args:

- *tolerance* – the minimum allowable difference between average and best fitness (default 0.001)

`inspyred.ec.terminators.`**`default_termination`**(*population*, *num_generations*, *num_evaluations*, *args*)

Return True.

This function acts as a default termination criterion for an evolutionary computation.

`inspyred.ec.terminators.`**`diversity_termination`**(*population*, *num_generations*, *num_evaluations*, *args*)

Return True if population diversity is less than a minimum diversity.

This function calculates the Euclidean distance between every pair of individuals in the population. It then compares the maximum of those distances with a specified minimum required diversity. This terminator is really only well-defined for candidate solutions which are list types of numeric values.

Optional keyword arguments in args:

- *min_diversity* – the minimum population diversity allowed (default 0.001)

`inspyred.ec.terminators.`**`evaluation_termination`**(*population*, *num_generations*, *num_evaluations*, *args*)

Return True if the number of function evaluations meets or exceeds a maximum.

This function compares the number of function evaluations that have been generated with a specified maximum. It returns True if the maximum is met or exceeded.

Optional keyword arguments in args:

- *max_evaluations* – the maximum candidate solution evaluations (default len(population))

`inspyred.ec.terminators.`**`generation_termination`**(*population*, *num_generations*, *num_evaluations*, *args*)

Return True if the number of generations meets or exceeds a maximum.

This function compares the number of generations with a specified maximum. It returns True if the maximum is met or exceeded.

Optional keyword arguments in args:

- *max_generations* – the maximum generations (default 1)

`inspyred.ec.terminators.`**`time_termination`**(*population*, *num_generations*, *num_evaluations*, *args*)

Return True if the elapsed time meets or exceeds a duration of time.

This function compares the elapsed time with a specified maximum. It returns True if the maximum is met or exceeded. If the *start_time* keyword argument is omitted, it defaults to *None* and will be set to the current system time (in seconds). If the *max_time* keyword argument is omitted, it will default to *None* and will immediately terminate. The *max_time* argument can be specified in seconds as a floating-point number, as minutes/seconds as a two-element tuple of floating-point numbers, or as hours/minutes/seconds as a three-element tuple of floating-point numbers.

Optional keyword arguments in args:

- *start_time* – the time from which to start measuring (default None)
- *max_time* – the maximum time that should elapse (default None)

`inspyred.ec.terminators.`**`user_termination`**(*population*, *num_generations*, *num_evaluations*, *args*)

Return True if user presses the ESC key when prompted.

This function prompts the user to press the ESC key to terminate the evolution. The prompt persists for a specified number of seconds before evolution continues. Additionally, the function can be customized to allow any press of the ESC key to be stored until the next time this function is called.

Note: This function makes use of the `msvcrt` (Windows) and `curses` (Unix) libraries. Other systems may not be supported.

Optional keyword arguments in args:

- *termination_response_timeout* – the number of seconds to wait for the user to press the ESC key (default 5)
- *clear_termination_buffer* – whether the keyboard buffer should be cleared before allowing the user to press a key (default True)

`variators` – Solution variation methods

This module provides pre-defined variators for evolutionary computations.

All variator functions have the following arguments:

- *random* – the random number generator object

- *candidates* – the candidate solutions
- *args* – a dictionary of keyword arguments

Each variator function returns the list of modified individuals. In the case of crossover variators, each pair of parents produces a pair of offspring. In the case of mutation variators, each candidate produces a single mutant.

These variators may make some limited assumptions about the type of candidate solutions on which they operate. These assumptions are noted in the table below. First, all variators except for `default_variation` assume that the candidate solutions are `Sequence` types. Those marked under "Real" assume that candidates are composed of real numbers. Those marked "Binary" assume that candidates are composed entirely of 0's and 1's. Those marked "Discrete" assume that candidates are composed of elements from a discrete set where the `DiscreteBounder` has been used. And those marked "Pickle" assume that candidates can be pickled.

Variator	Sequence	Real	Binary	Discrete	Pickle
default_variation					
arithmetic_crossover	X	X			
blend_crossover	X	X			
heuristic_crossover	X	X			X
laplace_crossover	X	X			
n_point_crossover	X				
partially_matched_crossover	X			X	
simulated_binary_crossover	X	X			
uniform_crossover	X				
bit_flip_mutation	X		X		
gaussian_mutation	X	X			
inversion_mutation	X				
nonuniform_mutation	X	X			
random_reset_mutation	X			X	
scramble_mutation	X				

`inspyred.ec.variators.`**`default_variation`**(*random*, *candidates*, *args*)
Return the set of candidates without variation.

`inspyred.ec.variators.`**`crossover`**(*cross*)
Return an inspyred crossover function based on the given function.

This function generator takes a function that operates on only two parent candidates to produce an iterable sequence of offspring (typically two). The generator handles the pairing of selected parents and collecting of all offspring.

The generated function chooses every odd candidate as a 'mom' and every even as a 'dad' (discounting the last candidate if there is an odd number). For each mom-dad pair, offspring are produced via the *cross* function.

The given function `cross` must have the following signature:

```
offspring = cross(random, mom, dad, args)
```

This function is most commonly used as a function decorator with the following usage:

```
@crossover
def cross(random, mom, dad, args):
    # Implementation of paired crossing
    pass
```

The generated function also contains an attribute named `single_crossover` which holds the original crossover function. In this way, the original single-set-of-parents function can be retrieved if necessary.

`inspyred.ec.variators.`**`arithmetic_crossover`**(*random*, *candidates*, *args*)
Return the offspring of arithmetic crossover on the candidates.

This function performs arithmetic crossover (AX), which is similar to a generalized weighted averaging of the candidate elements. The allele of each parent is weighted by the *ax_alpha* keyword argument, and the allele of the complement parent is weighted by 1 - *ax_alpha*. This averaging is only done on the alleles listed in the *ax_points* keyword argument. If this argument is `None`, then all alleles are used. This means that if this function is used with all default values, then offspring are simple averages of their parents. This function also makes use of the bounder function as specified in the EC's `evolve` method.

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *ax_alpha* – the weight for the averaging (default 0.5)
- *ax_points* – a list of points specifying the alleles to recombine (default None)

`inspyred.ec.variators.`**`blend_crossover`**`(`*random*, *candidates*, *args*`)`

Return the offspring of blend crossover on the candidates.

This function performs blend crossover (BLX), which is similar to arithmetic crossover with a bit of mutation. It creates offspring whose values are chosen randomly from a range bounded by the parent alleles but that is also extended by some amount proportional to the *blx_alpha* keyword argument. It is this extension of the range that provides the additional exploration. This averaging is only done on the alleles listed in the *blx_points* keyword argument. If this argument is `None`, then all alleles are used. This function also makes use of the bounder function as specified in the EC's `evolve` method.

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *blx_alpha* – the blending rate (default 0.1)
- *blx_points* – a list of points specifying the alleles to recombine (default None)

`inspyred.ec.variators.`**`heuristic_crossover`**`(`*random*, *candidates*, *args*`)`

Return the offspring of heuristic crossover on the candidates.

It performs heuristic crossover (HX), which is similar to the update rule used in particle swarm optimization. This function also makes use of the bounder function as specified in the EC's `evolve` method.

Note: This function assumes that candidates can be pickled (for hashing as keys to a dictionary).

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)

`inspyred.ec.variators.`**`laplace_crossover`**`(`*random*, *candidates*, *args*`)`

Return the offspring of Laplace crossover on the candidates.

This function performs Laplace crosssover (LX), following the implementation specified in (Deep and Thakur, "A new crossover operator for real coded genetic algorithms," Applied Mathematics and Computation, Volume 188, Issue 1, May 2007, pp. 895–911). This function also makes use of the bounder function as specified in the EC's `evolve` method.

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *lx_location* – the location parameter (default 0)
- *lx_scale* – the scale parameter (default 0.5)

In some sense, the *lx_location* and *lx_scale* parameters can be thought of as analogs in a Laplace distribution to the mean and standard deviation of a Gaussian distribution. If *lx_scale* is near zero, offspring will be produced near the parents. If *lx_scale* is farther from zero, offspring will be produced far from the parents.

`inspyred.ec.variators.`**`n_point_crossover`**(*random*, *candidates*, *args*)

Return the offspring of n-point crossover on the candidates.

This function performs n-point crossover (NPX). It selects *n* random points without replacement at which to 'cut' the candidate solutions and recombine them.

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *num_crossover_points* – the number of crossover points used (default 1)

`inspyred.ec.variators.`**`partially_matched_crossover`**(*random*, *candidates*, *args*)

Return the offspring of partially matched crossover on the candidates.

This function performs partially matched crossover (PMX). This type of crossover assumes that candidates are composed of discrete values that are permutations of a given set (typically integers). It produces offspring that are themselves permutations of the set.

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)

`inspyred.ec.variators.`**`simulated_binary_crossover`**(*random*, *candidates*, *args*)

Return the offspring of simulated binary crossover on the candidates.

This function performs simulated binary crossover (SBX), following the implementation in NSGA-II (Deb et al., ICANNGA 1999).

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *sbx_distribution_index* – the non-negative distribution index (default 10)

A small value of the *sbx_distribution_index* optional argument allows solutions far away from parents to be created as child solutions, while a large value restricts only near-parent solutions to be created as child solutions.

`inspyred.ec.variators.`**`uniform_crossover`**(*random*, *candidates*, *args*)

Return the offspring of uniform crossover on the candidates.

This function performs uniform crossover (UX). For each element of the parents, a biased coin is flipped to determine whether the first offspring gets the 'mom' or the 'dad' element. An optional keyword argument in args, `ux_bias`, determines the bias.

Optional keyword arguments in args:

- *crossover_rate* – the rate at which crossover is performed (default 1.0)
- *ux_bias* – the bias toward the first candidate in the crossover (default 0.5)

`inspyred.ec.variators.`**`mutator`**(*mutate*)

Return an inspyred mutator function based on the given function.

This function generator takes a function that operates on only one candidate to produce a single mutated candidate. The generator handles the iteration over each candidate in the set to be mutated.

The given function `mutate` must have the following signature:

```
mutant = mutate(random, candidate, args)
```

This function is most commonly used as a function decorator with the following usage:

```
@mutator
def mutate(random, candidate, args):
    # Implementation of mutation
    pass
```

The generated function also contains an attribute named `single_mutation` which holds the original mutation function. In this way, the original single-candidate function can be retrieved if necessary.

`inspyred.ec.variators.`**`bit_flip_mutation`**(*random*, *candidates*, *args*)

Return the mutants produced by bit-flip mutation on the candidates.

This function performs bit-flip mutation. If a candidate solution contains non-binary values, this function leaves it unchanged.

Optional keyword arguments in args:

- *mutation_rate* – the rate at which mutation is performed (default 0.1)

The mutation rate is applied on a bit by bit basis.

`inspyred.ec.variators.`**`gaussian_mutation`**(*random*, *candidates*, *args*)

Return the mutants created by Gaussian mutation on the candidates.

This function performs Gaussian mutation. This function makes use of the bounder function as specified in the EC's `evolve` method.

Optional keyword arguments in args:

- *mutation_rate* – the rate at which mutation is performed (default 0.1)
- *gaussian_mean* – the mean used in the Gaussian function (default 0)
- *gaussian_stdev* – the standard deviation used in the Gaussian function (default 1)

The mutation rate is applied on an element by element basis.

`inspyred.ec.variators.`**`inversion_mutation`**(*random*, *candidates*, *args*)

Return the mutants created by inversion mutation on the candidates.

This function performs inversion mutation. It randomly chooses two locations along the candidate and reverses the values within that slice.

Optional keyword arguments in args:

- *mutation_rate* – the rate at which mutation is performed (default 0.1)

The mutation rate is applied to the candidate as a whole (i.e., it either mutates or it does not, based on the rate).

`inspyred.ec.variators.`**`nonuniform_mutation`**(*random*, *candidates*, *args*)

Return the mutants produced by nonuniform mutation on the candidates.

The function performs nonuniform mutation as specified in (Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs," Springer, 1996). This function also makes use of the bounder function as specified in the EC's `evolve` method.

Note: This function **requires** that *max_generations* be specified in the *args* dictionary. Therefore, it is best to use this operator in conjunction with the `generation_termination` terminator.

Required keyword arguments in args:

- *max_generations* – the maximum number of generations for which evolution should take place

Optional keyword arguments in args:

- *mutation_strength* – the strength of the mutation, where higher values correspond to greater variation (default 1)

`inspyred.ec.variators.`**`random_reset_mutation`**`(`*random*, *candidates*, *args*`)`

Return the mutants produced by randomly choosing new values.

This function performs random-reset mutation. It assumes that candidate solutions are composed of discrete values. This function makes use of the bounder function as specified in the EC's `evolve` method, and it assumes that the bounder contains an attribute called *values* (which is true for instances of `DiscreteBounder`).

The mutation moves through a candidate solution and, with rate equal to the *mutation_rate*, randomly chooses a value from the set of allowed values to be used in that location. Note that this value may be the same as the original value.

Optional keyword arguments in args:

- *mutation_rate* – the rate at which mutation is performed (default 0.1)

The mutation rate is applied on an element by element basis.

`inspyred.ec.variators.`**`scramble_mutation`**`(`*random*, *candidates*, *args*`)`

Return the mutants created by scramble mutation on the candidates.

This function performs scramble mutation. It randomly chooses two locations along the candidate and scrambles the values within that slice.

Optional keyword arguments in args:

- *mutation_rate* – the rate at which mutation is performed (default 0.1)

The mutation rate is applied to the candidate as a whole (i.e., it either mutates or it does not, based on the rate).

5.2 Swarm Intelligence

5.2.1 `swarm` – Swarm intelligence

This module provides standard swarm intelligence algorithms.

class `inspyred.swarm.`**`ACS`**`(`*random*, *components*`)`

Represents an Ant Colony System discrete optimization algorithm.

This class is built upon the `EvolutionaryComputation` class making use of an external archive. It assumes that candidate solutions are composed of instances of `TrailComponent`.

Public Attributes:

- *components* – the full set of discrete components for a given problem
- *initial_pheromone* – the initial pheromone on a trail (default 0)
- *evaporation_rate* – the rate of pheromone evaporation (default 0.1)
- *learning_rate* – the learning rate used in pheromone updates (default 0.1)

class `inspyred.swarm.`**`PSO`**`(`*random*`)`

Represents a basic particle swarm optimization algorithm.

This class is built upon the `EvolutionaryComputation` class making use of an external archive and maintaining the population at the previous timestep, rather than a velocity. This approach was outlined in (Deb and Padhye, "Development of Efficient Particle Swarm Optimizers by Using Concepts from Evolutionary Algorithms", GECCO 2010, pp. 55–62). This class assumes that each candidate solution is a `Sequence` of real values.

Public Attributes:

- *topology* – the neighborhood topology (default topologies.star_topology)

Optional keyword arguments in `evolve` args parameter:

- *inertia* – the inertia constant to be used in the particle updating (default 0.5)
- *cognitive_rate* – the rate at which the particle's current position influences its movement (default 2.1)
- *social_rate* – the rate at which the particle's neighbors influence its movement (default 2.1)

class `inspyred.swarm.`**`TrailComponent`**(*element*, *value*, *maximize=True*, *delta=1*, *epsilon=1*)

Represents a discrete component of a trail in ant colony optimization.

An trail component has an element, which is its essence (and which is equivalent to the candidate in the `Individual` parent class); a value, which is its weight or cost; a pheromone level; and a desirability, which is a combination of the value and pheromone level (and which is equivalent to the fitness in the `Individual` parent class). Note that the desirability (and, thus, the fitness) cannot be set manually. It is calculated automatically from the value and pheromone level.

Public Attributes:

- *element* – the actual interpretation of this component
- *value* – the value or cost of the component
- *desirability* – the worth of the component based on value and pheromone level
- *delta* – the exponential contribution of the pheromone level on the desirability
- *epsilon* – the exponential contribution of the value on the desirability
- *maximize* – Boolean value stating use of maximization

5.2.2 `topologies` – Swarm topologies

This module defines various topologies for swarm intelligence algorithms.

Particle swarms make use of topologies, which determine the logical relationships among particles in the swarm (i.e., which ones belong to the same "neighborhood"). All topology functions have the following arguments:

- *random* – the random number generator object
- *population* – the population of Particles
- *args* – a dictionary of keyword arguments

Each topology function returns a list of lists of neighbors for each particle in the population. For example, if a swarm contained 10 particles, then this function would return a list containing 10 lists, each of which contained the neighbors for its corresponding particle in the population.

Rather than constructing and returning a list of lists directly, the topology functions could (and probably *should*, for efficiency) be written as generators that yield each neighborhood list one at a time. This is how the existing topology functions operate.

`inspyred.swarm.topologies.`**`ring_topology`**(*random*, *population*, *args*)

Returns the neighbors using a ring topology.

This function sets all particles in a specified sized neighborhood as neighbors for a given particle. This is known as a ring topology. The resulting list of lists of neighbors is returned.

Optional keyword arguments in args:

- *neighborhood_size* – the width of the neighborhood around a particle which determines the size of the neighborhood (default 3)

`inspyred.swarm.topologies.`**`star_topology`**(*random*, *population*, *args*)
Returns the neighbors using a star topology.

This function sets all particles as neighbors for all other particles. This is known as a star topology. The resulting list of lists of neighbors is returned.

5.3 Benchmark Problems

5.3.1 `benchmarks` – Benchmark optimization functions

This module provides a set of benchmark problems for global optimization.

class `inspyred.benchmarks.`**`Benchmark`**(*dimensions*, *objectives=1*)
Defines a global optimization benchmark problem.

This abstract class defines the basic structure of a global optimization problem. Subclasses should implement the `generator` and `evaluator` methods for a particular optimization problem, which can be used with inspyred's evolutionary computations.

In addition to being used with evolutionary computations, subclasses of this class are also callable. The arguments passed to such a call are combined into a list and passed as the single candidate to the evaluator method. The single calculated fitness is returned. What this means is that a given benchmark can act as a mathematical function that takes arguments and returns the value of the function, like the following example.:

```
my_function = benchmarks.Ackley(2)
output = my_function(-1.5, 4.2)
```

Public Attributes:

- *dimensions* – the number of inputs to the problem
- *objectives* – the number of outputs of the problem (default 1)
- *bounder* – the bounding function for the problem (default None)
- *maximize* – whether the problem is one of maximization (default True)

`evaluator`(*candidates*, *args*)
The evaluator function for the benchmark problem.

`generator`(*random*, *args*)
The generator function for the benchmark problem.

class `inspyred.benchmarks.`**`Binary`**(*benchmark*, *dimension_bits*)
Defines a binary problem based on an existing benchmark problem.

This class can be used to modify an existing benchmark problem to allow it to use a binary representation. The generator creates a list of binary values of size *dimensions*-by-*dimension_bits*. The evaluator accepts a candidate represented by such a binary list and transforms that candidate into a real-valued list as follows:

1. Each set of *dimension_bits* bits is converted to its positive integer representation.
2. Next, that integer value is divided by the maximum integer that can be represented by *dimension_bits* bits to produce a real number in the range [0, 1].
3. That real number is then scaled to the range [lower_bound, upper_bound] for that dimension (which should be defined by the bounder).

Public Attributes:

- *benchmark* – the original benchmark problem
- *dimension_bits* – the number of bits to use to represent each dimension
- *bounder* – a bounder that restricts elements of candidate solutions to the range [0, 1]
- *maximize* – whether the underlying benchmark problem is one of maximization

5.3.2 Single-Objective Benchmarks

class `inspyred.benchmarks.`**`Ackley`**(*dimensions=2*)

Defines the Ackley benchmark problem.

This class defines the Ackley global optimization problem. This is a multimodal minimization problem defined as follows:

$$f(x) = -20e^{-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)} + 20 + e$$

Here, n represents the number of dimensions and $x_i \in [-32, 32]$ for $i = 1, ..., n$.

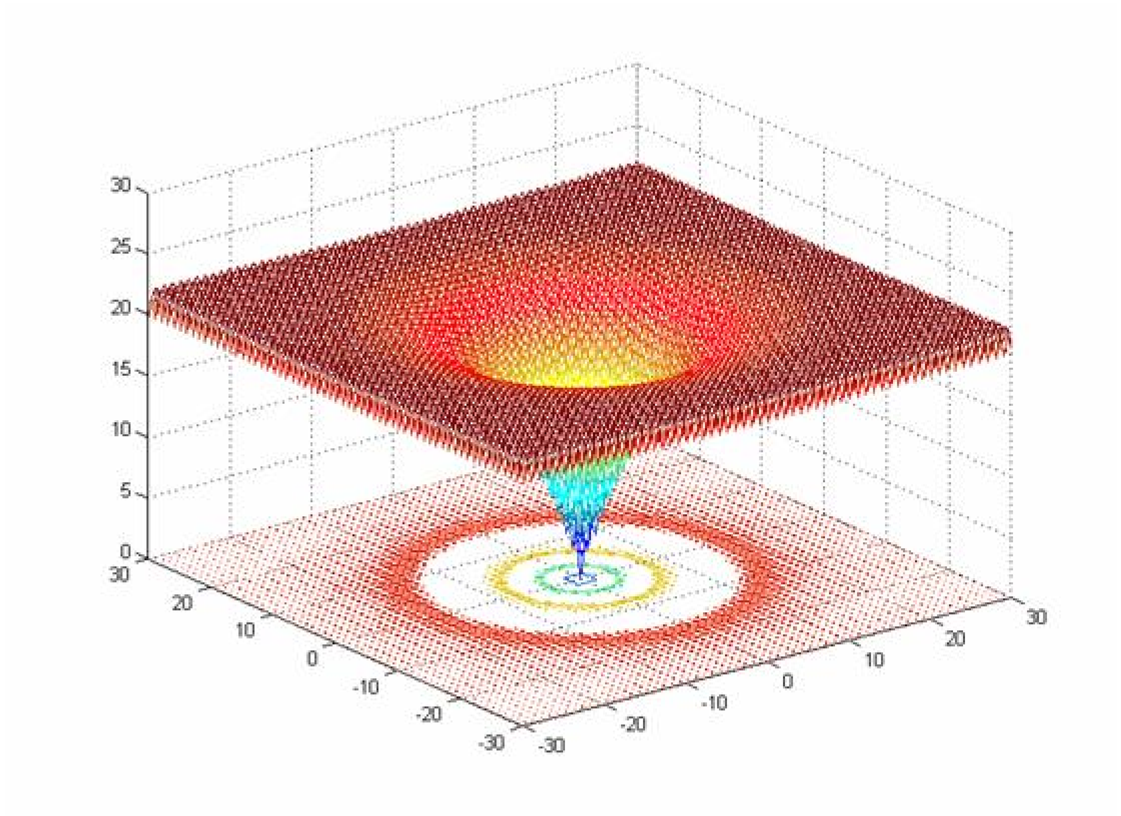


Figure 5.3: Two-dimensional Ackley function (image source)

Public Attributes:

- *global_optimum* – the problem input that produces the optimum output. Here, this corresponds to [0, 0, ..., 0].

class `inspyred.benchmarks.`**`Griewank`**(*dimensions=2*)

Defines the Griewank benchmark problem.

This class defines the Griewank global optimization problem. This is a highly multimodal minimization problem with numerous, wide-spread, regularly distributed local minima. It is defined as follows:

$$f(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1$$

Here, n represents the number of dimensions and $x_i \in [-600, 600]$ for $i = 1, ..., n$.

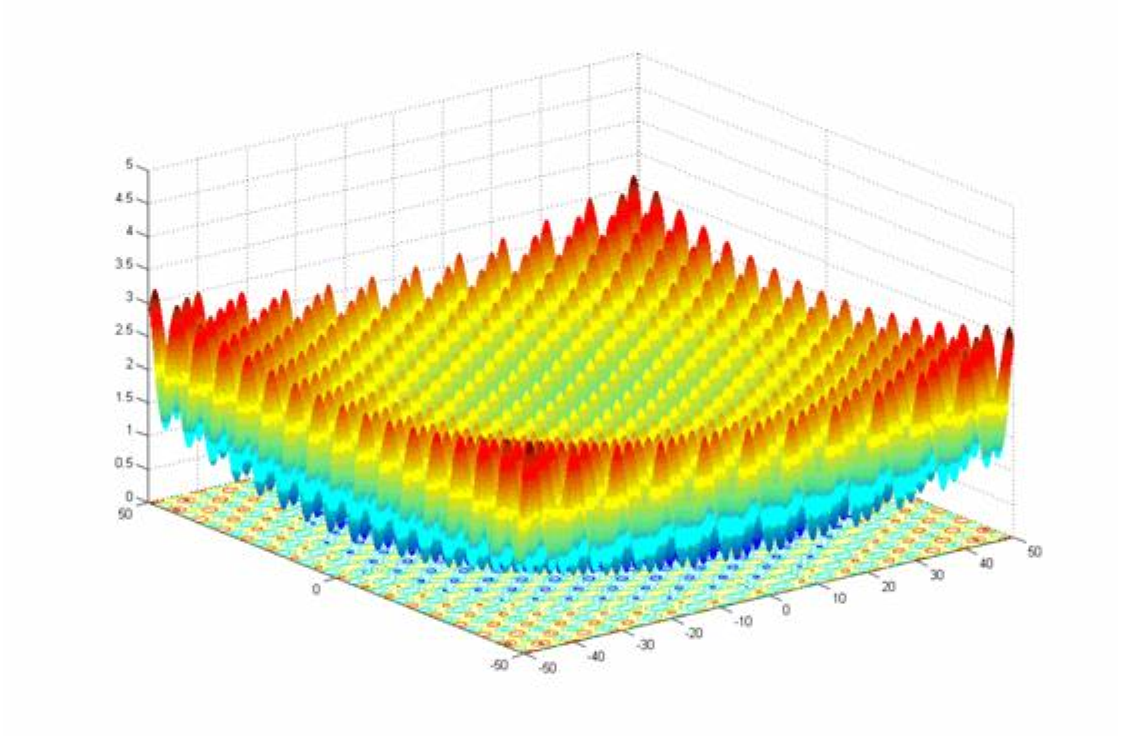


Figure 5.4: Two-dimensional Griewank function (image source)

Public Attributes:

- *global_optimum* – the problem input that produces the optimum output. Here, this corresponds to [0, 0, ..., 0].

class `inspyred.benchmarks.`**`Rastrigin`**(*dimensions=2*)

Defines the Rastrigin benchmark problem.

This class defines the Rastrigin global optimization problem. This is a highly multimodal minimization problem where the local minima are regularly distributed. It is defined as follows:

$$f(x) = \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i) + 10)$$

Here, n represents the number of dimensions and $x_i \in [-5.12, 5.12]$ for $i = 1, ..., n$.

Public Attributes:

- *global_optimum* – the problem input that produces the optimum output. Here, this corresponds to [0, 0, ..., 0].

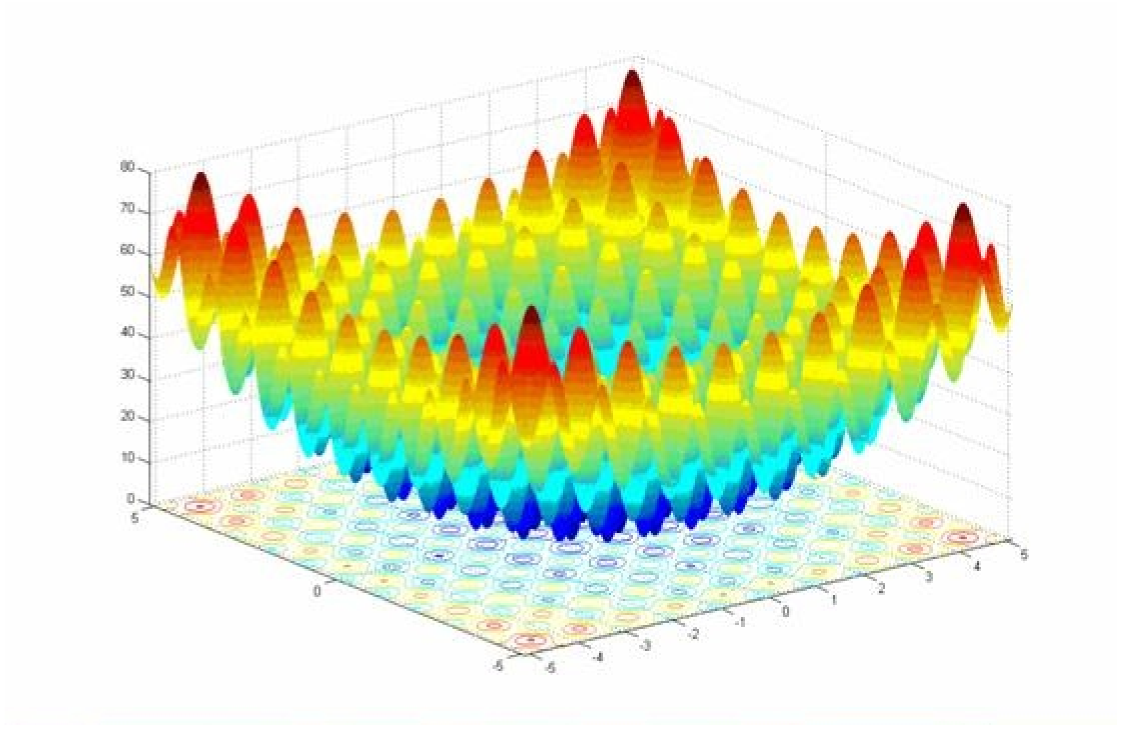


Figure 5.5: Two-dimensional Rastrigin function (image source)

class `inspyred.benchmarks.`**`Rosenbrock`**`(`*dimensions=2*`)`

Defines the Rosenbrock benchmark problem.

This class defines the Rosenbrock global optimization problem, also known as the "banana function." The global optimum sits within a narrow, parabolic-shaped flattened valley. It is defined as follows:

$$f(x) = \sum_{i=1}^{n-1}[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2]$$

Here, n represents the number of dimensions and $x_i \in [-5, 10]$ for $i = 1, ..., n$.

Public Attributes:

- *global_optimum* – the problem input that produces the optimum output. Here, this corresponds to [1, 1, ..., 1].

class `inspyred.benchmarks.`**`Schwefel`**`(`*dimensions=2*`)`

Defines the Schwefel benchmark problem.

This class defines the Schwefel global optimization problem. It is defined as follows:

$$f(x) = 418.9829n - \sum_{i=1}^{n}\left[-x_i \sin(\sqrt{|x_i|})\right]$$

Here, n represents the number of dimensions and $x_i \in [-500, 500]$ for $i = 1, ..., n$.

Public Attributes:

- *global_optimum* – the problem input that produces the optimum output. Here, this corresponds to [420.9687, 420.9687, ..., 420.9687].

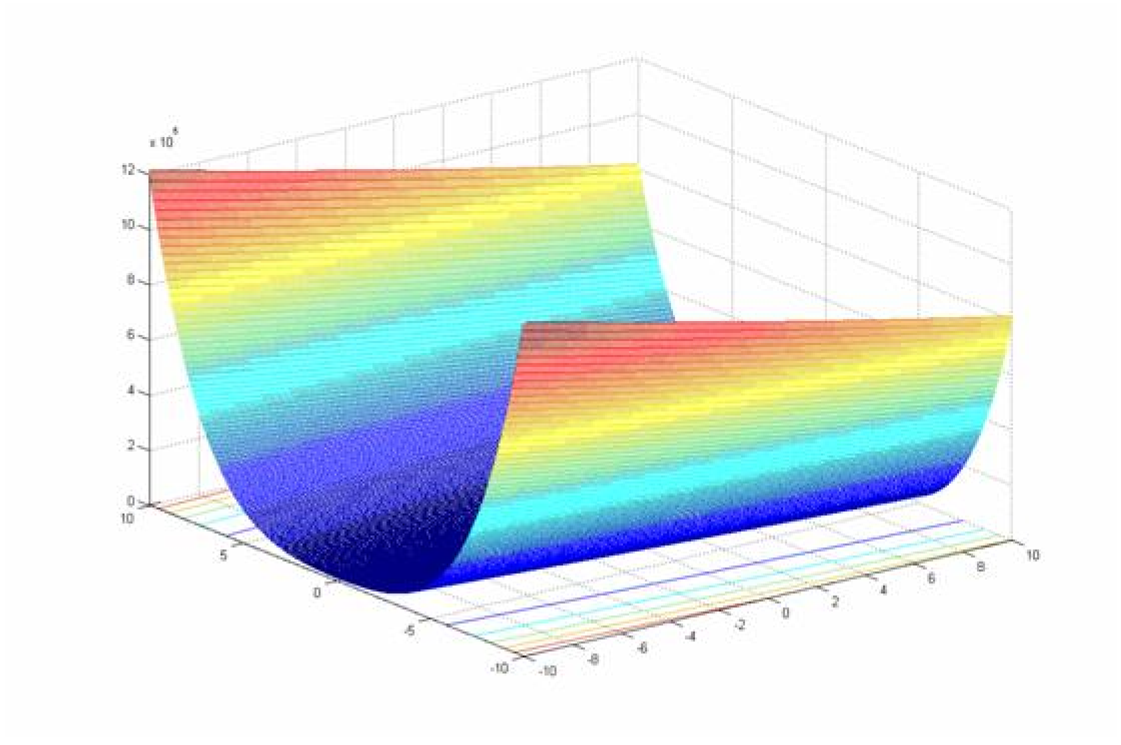


Figure 5.6: Two-dimensional Rosenbrock function (image source)

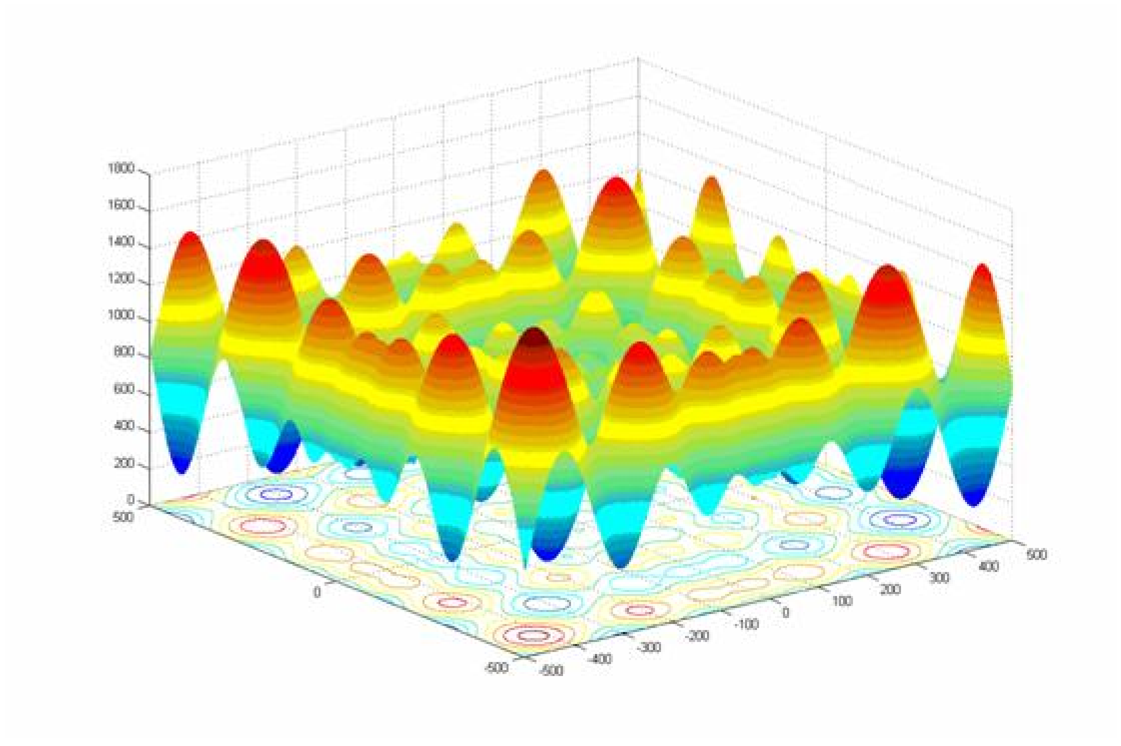


Figure 5.7: Two-dimensional Schwefel function (image source)

class `inspyred.benchmarks.`**`Sphere`**`(`*dimensions=2*`)`

Defines the Sphere benchmark problem.

This class defines the Sphere global optimization problem, also called the "first function of De Jong's" or "De Jong's F1." It is continuous, convex, and unimodal, and it is defined as follows:

$$f(x) = \sum_{i=1}^{n} x_i^2$$

Here, n represents the number of dimensions and $x_i \in [-5.12, 5.12]$ for $i = 1, ..., n$.

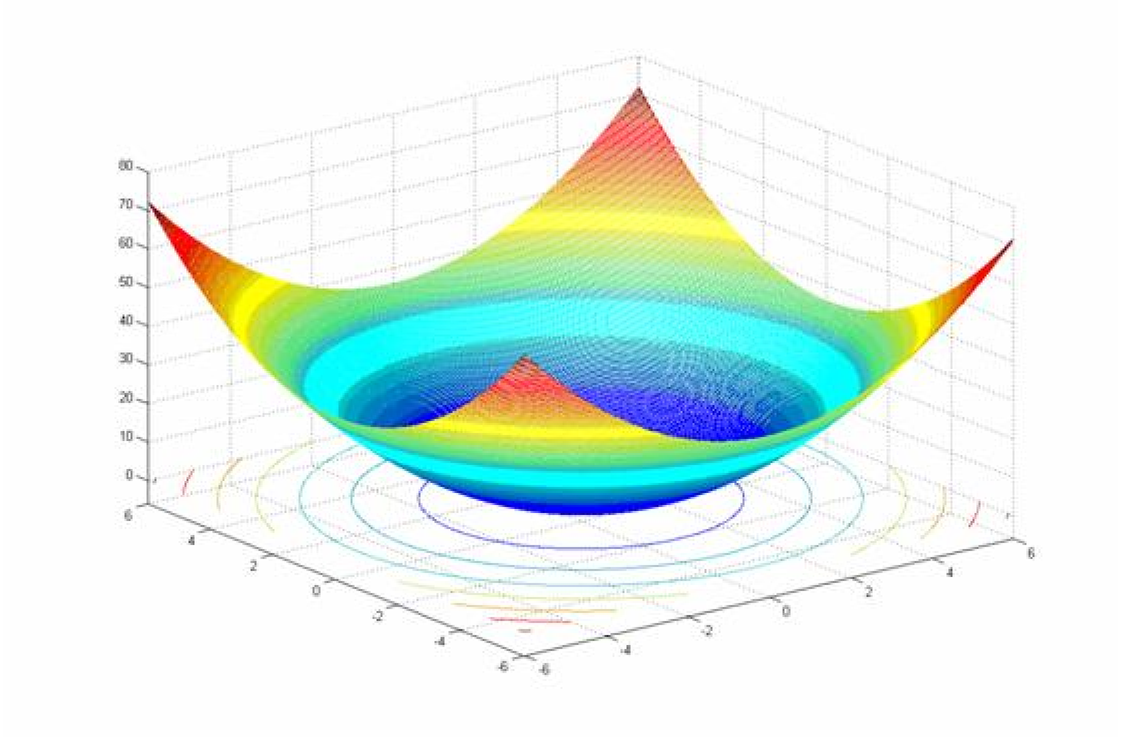


Figure 5.8: Two-dimensional Sphere function (image source)

Public Attributes:

- *global_optimum* – the problem input that produces the optimum output. Here, this corresponds to [0, 0, ..., 0].

5.3.3 Multi-Objective Benchmarks

class `inspyred.benchmarks.`**`Kursawe`**`(`*dimensions=2*`)`

Defines the Kursawe multiobjective benchmark problem.

This class defines the Kursawe multiobjective minimization problem. This function accepts an n-dimensional input and produces a two-dimensional output. It is defined as follows:

$$f_1(x) = \sum_{i=1}^{n-1} \left[-10e^{-0.2\sqrt{x_i^2 + x_{i+1}^2}} \right]$$

$$f_2(x) = \sum_{i=1}^{n} \left[|x_i|^{0.8} + 5\sin(x_i)^3 \right]$$

Here, n represents the number of dimensions and $x_i \in [-5, 5]$ for $i = 1, ..., n$.

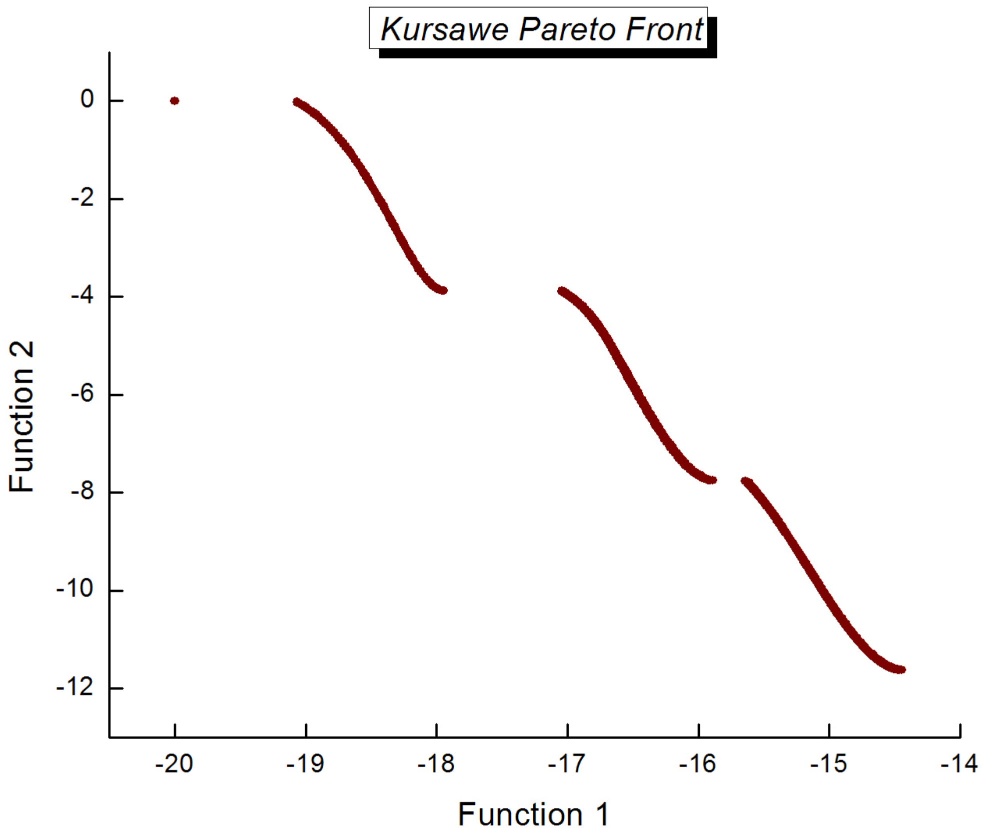


Figure 5.9: Three-dimensional Kursawe Pareto front (image source)

class `inspyred.benchmarks.`**`DTLZ1`**(*dimensions=2*, *objectives=2*)

Defines the DTLZ1 multiobjective benchmark problem.

This class defines the DTLZ1 multiobjective minimization problem taken from (Deb et al., "Scalable Multi-Objective Optimization Test Problems." CEC 2002, pp. 825–830). This function accepts an n-dimensional input and produces an m-dimensional output. It is defined as follows:

$$f_1(\vec{x}) = \frac{1}{2}x_1 \dots x_{m-1}(1 + g(\vec{x_m}))$$
$$f_i(\vec{x}) = \frac{1}{2}x_1 \dots x_{m-i}(1 + g(\vec{x_m}))$$
$$f_m(\vec{x}) = \frac{1}{2}(1 - x_1)(1 + g(\vec{x_m}))$$
$$g(\vec{x_m}) = 100\left[|\vec{x_m}| + \sum_{x_i \in \vec{x_m}} \left((x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5))\right)\right]$$

Here, n represents the number of dimensions, m represents the number of objectives, $x_i \in [0, 1]$ for $i = 1, ..., n$, and $\vec{x_m} = x_m x_{m+1} \dots x_n$.

The recommendation given in the paper mentioned above is to provide 4 more dimensions than objectives. For instance, if we want to use 2 objectives, we should use 6 dimensions.

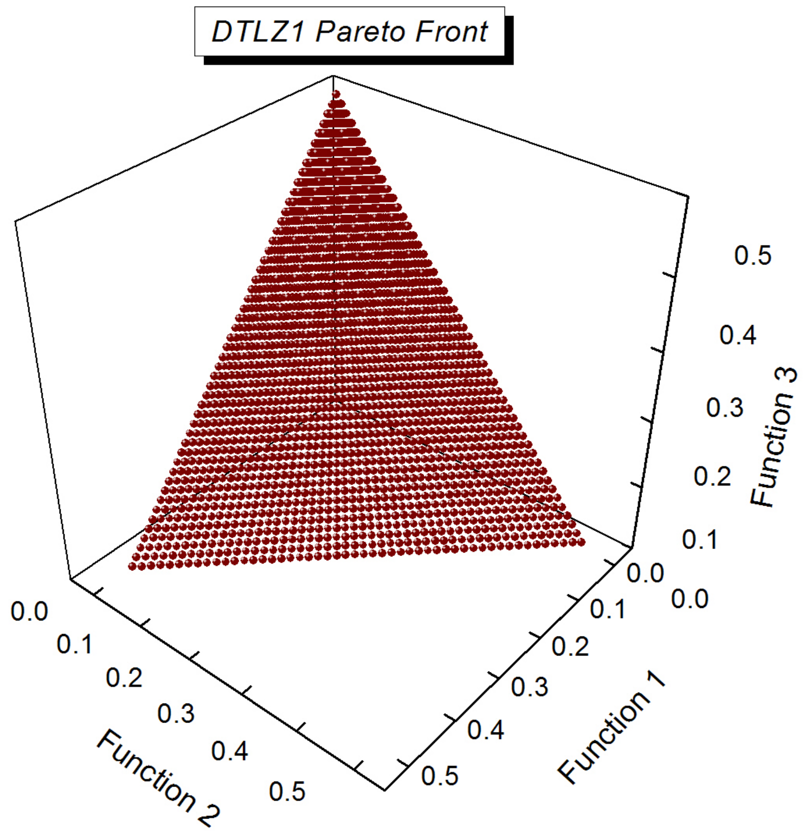


Figure 5.10: Three-dimensional DTLZ1 Pareto front (image source)

global_optimum()

Return a globally optimal solution to this problem.

This function returns a globally optimal solution (i.e., a solution that lives on the Pareto front). Since there are many solutions that are Pareto-optimal, this function randomly chooses one to return.

class inspyred.benchmarks.**DTLZ2**(*dimensions=2*, *objectives=2*)

Defines the DTLZ2 multiobjective benchmark problem.

This class defines the DTLZ2 multiobjective minimization problem taken from (Deb et al., "Scalable Multi-Objective Optimization Test Problems." CEC 2002, pp. 825–830). This function accepts an n-dimensional input and produces an m-dimensional output. It is defined as follows:

$$\begin{aligned}
f_1(\vec{x}) &= (1 + g(\vec{x_m}))\cos(x_1\pi/2)\cos(x_2\pi/2)\ldots\cos(x_{m-2}\pi/2)\cos(x_{m-1}\pi/2) \\
f_i(\vec{x}) &= (1 + g(\vec{x_m}))\cos(x_1\pi/2)\cos(x_2\pi/2)\ldots\cos(x_{m-i}\pi/2)\sin(x_{m-i+1}\pi/2) \\
f_m(\vec{x}) &= (1 + g(\vec{x_m}))\sin(x_1\pi/2) \\
g(\vec{x_m}) &= \sum_{x_i \in \vec{x_m}} (x_i - 0.5)^2
\end{aligned}$$

Here, n represents the number of dimensions, m represents the number of objectives, $x_i \in [0, 1]$ for $i = 1, ..., n$, and $\vec{x_m} = x_m x_{m+1} \dots x_n$.

The recommendation given in the paper mentioned above is to provide 9 more dimensions than objectives. For instance, if we want to use 2 objectives, we should use 11 dimensions.

`global_optimum``()`

Return a globally optimal solution to this problem.

This function returns a globally optimal solution (i.e., a solution that lives on the Pareto front). Since there are many solutions that are Pareto-optimal, this function randomly chooses one to return.

class `inspyred.benchmarks.`**`DTLZ3`**`(`*dimensions=2*, *objectives=2*`)`

Defines the DTLZ3 multiobjective benchmark problem.

This class defines the DTLZ3 multiobjective minimization problem taken from (Deb et al., "Scalable Multi-Objective Optimization Test Problems." CEC 2002, pp. 825–830). This function accepts an n-dimensional input and produces an m-dimensional output. It is defined as follows:

$$\begin{aligned}
f_1(\vec{x}) &= (1 + g(\vec{x_m})) \cos(x_1\pi/2) \cos(x_2\pi/2) \dots \cos(x_{m-2}\pi/2) \cos(x_{m-1}\pi/2) \\
f_i(\vec{x}) &= (1 + g(\vec{x_m})) \cos(x_1\pi/2) \cos(x_2\pi/2) \dots \cos(x_{m-i}\pi/2) \sin(x_{m-i+1}\pi/2) \\
f_m(\vec{x}) &= (1 + g(\vec{x_m})) \sin(x_1\pi/2) \\
g(\vec{x_m}) &= 100 \left[|\vec{x_m}| + \sum_{x_i \in \vec{x_m}} \left((x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5)) \right) \right]
\end{aligned}$$

Here, n represents the number of dimensions, m represents the number of objectives, $x_i \in [0, 1]$ for $i = 1, ..., n$, and $\vec{x_m} = x_m x_{m+1} \dots x_n$.

The recommendation given in the paper mentioned above is to provide 9 more dimensions than objectives. For instance, if we want to use 2 objectives, we should use 11 dimensions.

`global_optimum``()`

Return a globally optimal solution to this problem.

This function returns a globally optimal solution (i.e., a solution that lives on the Pareto front). Since there are many solutions that are Pareto-optimal, this function randomly chooses one to return.

class `inspyred.benchmarks.`**`DTLZ4`**`(`*dimensions=2*, *objectives=2*, *alpha=100*`)`

Defines the DTLZ4 multiobjective benchmark problem.

This class defines the DTLZ4 multiobjective minimization problem taken from (Deb et al., "Scalable Multi-Objective Optimization Test Problems." CEC 2002, pp. 825–830). This function accepts an n-dimensional input and produces an m-dimensional output. It is defined as follows:

$$\begin{aligned}
f_1(\vec{x}) &= (1 + g(\vec{x_m})) \cos(x_1^\alpha\pi/2) \cos(x_2^\alpha\pi/2) \dots \cos(x_{m-2}^\alpha\pi/2) \cos(x_{m-1}^\alpha\pi/2) \\
f_i(\vec{x}) &= (1 + g(\vec{x_m})) \cos(x_1^\alpha\pi/2) \cos(x_2^\alpha\pi/2) \dots \cos(x_{m-i}^\alpha\pi/2) \sin(x_{m-i+1}^\alpha\pi/2) \\
f_m(\vec{x}) &= (1 + g(\vec{x_m})) \sin(x_1^\alpha\pi/2) \\
g(\vec{x_m}) &= \sum_{x_i \in \vec{x_m}} (x_i - 0.5)^2
\end{aligned}$$

Here, n represents the number of dimensions, m represents the number of objectives, $x_i \in [0, 1]$ for $i = 1, ..., n$, $\vec{x_m} = x_m x_{m+1} \dots x_n$, and $\alpha = 100$.

The recommendation given in the paper mentioned above is to provide 9 more dimensions than objectives. For instance, if we want to use 2 objectives, we should use 11 dimensions.

`global_optimum``()`

Return a globally optimal solution to this problem.

This function returns a globally optimal solution (i.e., a solution that lives on the Pareto front). Since there are many solutions that are Pareto-optimal, this function randomly chooses one to return.

class `inspyred.benchmarks.`**`DTLZ5`**`(`*dimensions=2*, *objectives=2*`)`

Defines the DTLZ5 multiobjective benchmark problem.

This class defines the DTLZ5 multiobjective minimization problem taken from (Deb et al., "Scalable Multi-Objective Optimization Test Problems." CEC 2002, pp. 825–830). This function accepts an n-dimensional input and produces an m-dimensional output. It is defined as follows:

$$\begin{aligned}
f_1(\vec{x}) &= (1 + g(\vec{x_m}))\cos(\theta_1\pi/2)\cos(\theta_2\pi/2)\ldots\cos(\theta_{m-2}\pi/2)\cos(\theta_{m-1}\pi/2)\\
f_i(\vec{x}) &= (1 + g(\vec{x_m}))\cos(\theta_1\pi/2)\cos(\theta_2\pi/2)\ldots\cos(\theta_{m-i}\pi/2)\sin(\theta_{m-i+1}\pi/2)\\
f_m(\vec{x}) &= (1 + g(\vec{x_m}))\sin(\theta_1\pi/2)\\
\theta_i &= \frac{\pi}{4(1 + g(\vec{x_m}))}(1 + 2g(\vec{x_m})x_i)\\
g(\vec{x_m}) &= \sum_{x_i \in \vec{x_m}} (x_i - 0.5)^2
\end{aligned}$$

Here, n represents the number of dimensions, m represents the number of objectives, $x_i \in [0, 1]$ for $i = 1, ..., n$, and $\vec{x_m} = x_m x_{m+1} \ldots x_n$.

The recommendation given in the paper mentioned above is to provide 9 more dimensions than objectives. For instance, if we want to use 2 objectives, we should use 11 dimensions.

`global_optimum``()`

Return a globally optimal solution to this problem.

This function returns a globally optimal solution (i.e., a solution that lives on the Pareto front). Since there are many solutions that are Pareto-optimal, this function randomly chooses one to return.

class `inspyred.benchmarks.`**`DTLZ6`**`(`*dimensions=2*, *objectives=2*`)`

Defines the DTLZ6 multiobjective benchmark problem.

This class defines the DTLZ6 multiobjective minimization problem taken from (Deb et al., "Scalable Multi-Objective Optimization Test Problems." CEC 2002, pp. 825–830). This function accepts an n-dimensional input and produces an m-dimensional output. It is defined as follows:

$$\begin{aligned}
f_1(\vec{x}) &= (1 + g(\vec{x_m}))\cos(\theta_1\pi/2)\cos(\theta_2\pi/2)\ldots\cos(\theta_{m-2}\pi/2)\cos(\theta_{m-1}\pi/2)\\
f_i(\vec{x}) &= (1 + g(\vec{x_m}))\cos(\theta_1\pi/2)\cos(\theta_2\pi/2)\ldots\cos(\theta_{m-i}\pi/2)\sin(\theta_{m-i+1}\pi/2)\\
f_m(\vec{x}) &= (1 + g(\vec{x_m}))\sin(\theta_1\pi/2)\\
\theta_i &= \frac{\pi}{4(1 + g(\vec{x_m}))}(1 + 2g(\vec{x_m})x_i)\\
g(\vec{x_m}) &= \sum_{x_i \in \vec{x_m}} x_i^{0.1}
\end{aligned}$$

Here, n represents the number of dimensions, m represents the number of objectives, $x_i \in [0, 1]$ for $i = 1, ..., n$, and $\vec{x_m} = x_m x_{m+1} \ldots x_n$.

The recommendation given in the paper mentioned above is to provide 9 more dimensions than objectives. For instance, if we want to use 2 objectives, we should use 11 dimensions.

`global_optimum``()`

Return a globally optimal solution to this problem.

This function returns a globally optimal solution (i.e., a solution that lives on the Pareto front). Since there are many solutions that are Pareto-optimal, this function randomly chooses one to return.

class `inspyred.benchmarks.`**`DTLZ7`**`(`*dimensions=2*, *objectives=2*`)`

Defines the DTLZ7 multiobjective benchmark problem.

This class defines the DTLZ7 multiobjective minimization problem taken from (Deb et al., "Scalable Multi-Objective Optimization Test Problems." CEC 2002, pp. 825–830). This function accepts an n-dimensional input and produces an m-dimensional output. It is defined as follows:

$$
\begin{aligned}
f_1(\vec{x}) &= x_1 \\
f_i(\vec{x}) &= x_i \\
f_m(\vec{x}) &= (1 + g(\vec{x_m}))h(f_1, f_2, \ldots, f_{m-1}, g) \\
g(\vec{x_m}) &= 1 + \frac{9}{|\vec{x_m}|} \sum_{x_i \in \vec{x_m}} x_i \\
h(f_1, f_2, \ldots, f_{m-1}, g) &= m - \sum_{i=1}^{m-1} \left[\frac{f_1}{1+g}(1 + \sin(3\pi f_i)) \right]
\end{aligned}
$$

Here, n represents the number of dimensions, m represents the number of objectives, $x_i \in [0, 1]$ for $i = 1, ..., n$, and $\vec{x_m} = x_m x_{m+1} \ldots x_n$.

The recommendation given in the paper mentioned above is to provide 19 more dimensions than objectives. For instance, if we want to use 2 objectives, we should use 21 dimensions.

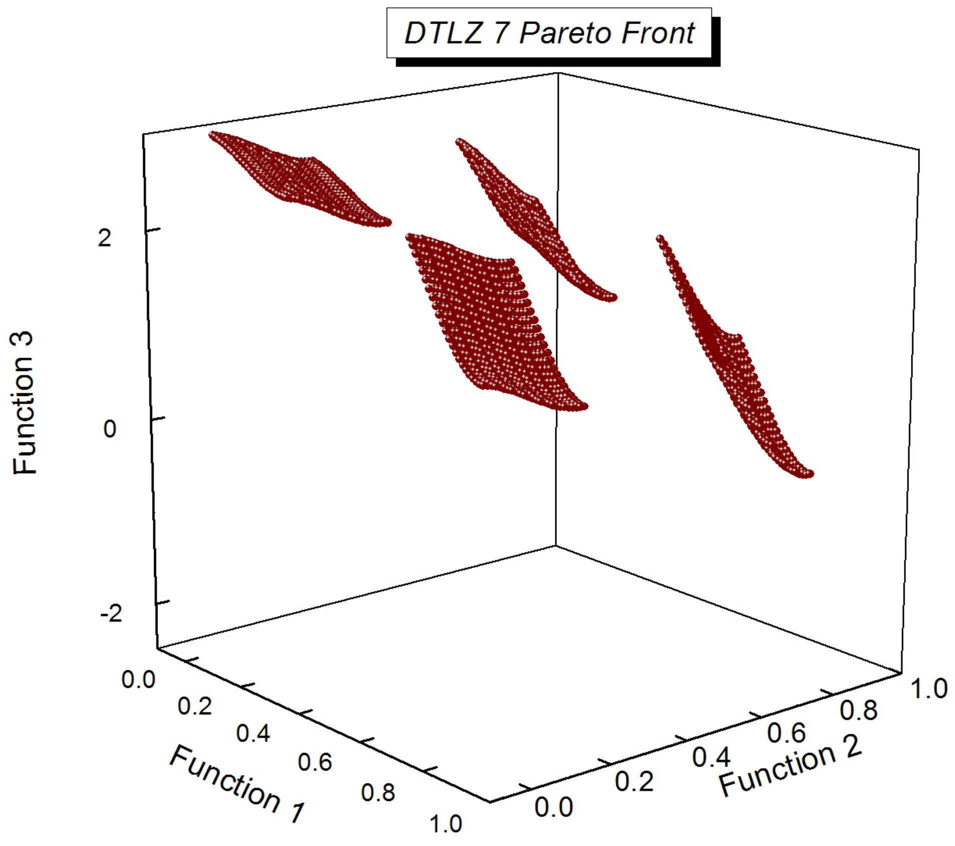


Figure 5.11: Three-dimensional DTLZ7 Pareto front (image source)

global_optimum()
Return a globally optimal solution to this problem.

This function returns a globally optimal solution (i.e., a solution that lives on the Pareto front). Since there are many solutions that are Pareto-optimal, this function randomly chooses one to return.

5.3.4 Discrete Optimization Benchmarks

class `inspyred.benchmarks.`**`Knapsack`**(*capacity*, *items*, *duplicates=False*)

Defines the Knapsack benchmark problem.

This class defines the Knapsack problem: given a set of items, each with a weight and a value, find the set of items of maximal value that fit within a knapsack of fixed weight capacity. This problem assumes that the `items` parameter is a list of (weight, value) tuples. This problem is most easily defined as a maximization problem, where the total value contained in the knapsack is to be maximized. However, for the evolutionary computation (which may create infeasible solutions that exceed the knapsack capacity), the fitness is either the total value in the knapsack (for feasible solutions) or the negative difference between the actual contents and the maximum capacity of the knapsack.

If evolutionary computation is to be used, then the `generator` function should be used to create candidates. If ant colony optimization is used, then the `constructor` function creates candidates. The `evaluator` function performs the evaluation for both types of candidates.

Public Attributes:

- *capacity* – the weight capacity of the knapsack
- *items* – a list of (weight, value) tuples corresponding to the possible items to be placed into the knapsack
- *components* – the set of `TrailComponent` objects constructed from the `items` parameter
- *duplicates* – Boolean value specifying whether items may be duplicated in the knapsack (i.e., False corresponds to 0/1 Knapsack)
- *bias* – the bias in selecting the component of maximum desirability when constructing a candidate solution for ant colony optimization (default 0.5)

`constructor`(*random*, *args*)

Return a candidate solution for an ant colony optimization.

`evaluator`(*candidates*, *args*)

Return the fitness values for the given candidates.

`generator`(*random*, *args*)

Return a candidate solution for an evolutionary computation.

class `inspyred.benchmarks.`**`TSP`**(*weights*)

Defines the Traveling Salesman benchmark problem.

This class defines the Traveling Salesman problem: given a set of locations and their pairwise distances, find the shortest route that visits each location once and only once. This problem assumes that the `weights` parameter is an n-by-n list of pairwise distances among n locations. This problem is treated as a maximization problem, so fitness values are determined to be the reciprocal of the total path length.

In the case of typical evolutionary computation, a candidate solution is represented as a permutation of the n-element list of the integers from 0 to n-1. In the case of ant colony optimization, a candidate solution is represented by a list of `TrailComponent` objects which have (source, destination) tuples as their elements and the reciprocal of the weight from source to destination as their values.

If evolutionary computation is to be used, then the `generator` function should be used to create candidates. If ant colony optimization is used, then the `constructor` function creates candidates. The `evaluator` function performs the evaluation for both types of candidates.

Public Attributes:

- *weights* – the two-dimensional list of pairwise distances
- *components* – the set of `TrailComponent` objects constructed from the `weights` attribute, where the element is the (source, destination) tuple and the value is the reciprocal of its `weights` entry
- *bias* – the bias in selecting the component of maximum desirability when constructing a candidate solution for ant colony optimization (default 0.5)

`constructor`(*random*, *args*)
    Return a candidate solution for an ant colony optimization.

`evaluator`(*candidates*, *args*)
    Return the fitness values for the given candidates.

`generator`(*random*, *args*)
    Return a candidate solution for an evolutionary computation.

CHAPTER

SIX

TROUBLESHOOTING

Given the flexibility of the inspyred library, along with the inherent stochasticity of the algorithms, it can be difficult to track down errors that will inevitably arise. This chapter provides some suggestions that may make the process easier.

6.1 Always Store the Seed

Every inspyred algorithm requires that a pseudo-random number generator (PRNG) object be passed to it. This allows users to make use of different PRNGs if more sophisticated random number generation is desired (as long as it implements the relevant methods of Python's `Random` class). This also means that the PRNG must be seeded prior to its passing to an inspyred algorithm. This seed should always be printed (preferably to a file) in case the exact run of the algorithm needs to be duplicated. If an error occurs in a given run, it can be restarted by providing the same seed. The following code provides an example:

```
import random
import time
my_seed = int(time.time())
seedfile = open('randomseed.txt', 'w')
seedfile.write('{0}'.format(my_seed))
seedfile.close()
prng = random.Random()
prng.seed(my_seed)
```

6.2 Use and Consult the Logs

All inspyred algorithms provide detailed debugging data using Python's core `logging` module. This can be enabled by adding the following code to the `main` or calling scope:

```
import logging
logger = logging.getLogger('inspyred.ec')
logger.setLevel(logging.DEBUG)
file_handler = logging.FileHandler('inspyred.log', mode='w')
file_handler.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)
```

Consulting the log file will often reveal which component is misbehaving or behaving unexpectedly.

6.3 Choose Operators that Work Together

The inspyred library gives users freedom to combine operators in almost any way they choose. However, this freedom means that the library will be unable to alert the user when a particular combination of operators produces a nonsensical algorithm. Remember that the operators must work together. For example, tournament selection may be employed to select 20 individuals from a population of 100. Then Gaussian mutation may be used to create 20 offspring. Finally, generational replacement may create the next generation from those offspring. The library will allow this, even though it means that, for at least the first generation, the population size is not constant. (It drops from 100 to 20.) The reason such a thing is allowed is because there may be a need for an algorithm that requires a non-constant population size. The inspyred library does not restrict any such algorithm. It is up to the user to ensure that all components work together to achieve the desired ends. (As stated previously, consulting the log file can help determine whether operators are combined correctly.)

6.4 Converting from ecspy

Users of the ecspy library, the precursor to inspyred, may want to modify their code to use the new library instead. The best advice in such a case would be to find a similar example from the inspyred documentation and use it as a basis for the existing code. The problem-specific generator and evaluator functions will probably not need to be changed (depending on their level of separation from the library itself), but the library functionality will probably change around them. Custom operators will similarly probably need only minor changes, if any.

However, the pre-defined operators have been both modified and expanded. Existing keyword arguments should be checked carefully against the inspyred documentation to ensure that they are correct. An incorrectly named keyword argument will pass through unnoticed by the algorithm, and the default value will be used. For instance, in ecspy, a keyword argument for `tournament_selection` was *tourn_size*. In inspyred, it has been more clearly named *tournament_size*. If code from ecspy is used without modification, then the tournament selection will use 2 (the default) as the tournament size, regardless of the setting for *tourn_size*. In all cases, consult the documentation to ensure that the appropriate keyword arguments are used.

CHAPTER

SEVEN

INDICES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX