NSGA-II

Multi-objective optimisation problem using the elitist non dominated sorting genetic algorithm (NSGA-II)

$$\min\{f_1, f_2\};$$

$$f_1 = \frac{\left[\left(\frac{x_1}{2.0}\right)^2 + \left(\frac{x_2}{4.0}\right)^2 + (x_3)^2\right]}{3.0}$$

$$f_2 = \frac{\left[\left(\frac{x_1}{2.0} - 1.0\right)^2 + \left(\frac{x_2}{4.0} - 1.0\right)^2 + (x_3 - 1.0)^2\right]}{3.0}$$

$$-4.0 \le x_1, x_2, x_3 \le 4.0$$

The algorithm was implemented in Python with DEAP library. The code is in the Appendix.

First of all, types are created using the creator and initialized using the toolbox. "FitnessMin" is created for two objective optimisation algorithm. Both weights are set to -1.0 to make it minimization. "Individual" class inherit from list with fitness attribute is created.

Decision variables are encoded using Gray coding so the individuals will be simple lists containing Booleans. “attr_bool” is registered to the toolbox, which returns 0 or 1 with equal probability. It is used to represent an attribute of genes. “individual” is registered, which returns an "Individual" class object. This function creates an individual using “initRepeat” function which repeats function “attr_bool” 10×3 times. The length of the individual is 10×3 because there are 3 decision variables where each one is represented with 10 bits. "population" also uses “initRepeat” function to repeat the registered "individual" function to create a list of population. "evaluation" is registered which calls “calcFitness” function.

“calcFitness” function calculates the fitness values. It first separates an individual into 3 decision variables using “separatevariables” function. “separatevariables” function separates an individual into three 10-bit decision variables and convert it to real numbers using “chrom2real” function. “chrom2real” function first converts 10-bit Gray code to a string, converts it to binary number using “gray_to_bin” function and turns this to an integer. Then, it scales the integer into the range between -4.0 and 4.0. After converting each decision variable into real numbers, “separatevariables” function returns an array of the converted decision variables. “calcFitness” function calculates fitness values using the decision variables.

28 individuals are initialized using the registered “population” function and each individual is evaluated using the registered “evaluate” function. The results are set to each individual as an attribute.

Table 1 shows the three initial decision variables x_1, x_2 and x_3 and fitness values f_1 and f_2 of each individual when the population is initialized randomly with random seed set to 1004.

x_1	x_2	x_3	f_1	f_2
3.5546875	-3.9140625	-0.765625	1.567541758	2.545406342
0.3359375	-0.4765625	2.3359375	1.833003998	1.243160248
0.6875	1.875	1.21875	0.607747396	0.253580729
3.4453125	3.953125	-1.1953125	1.791005452	1.780588786
-0.2578125	-1.6015625	0.515625	0.147599538	1.156714122
2.125	0.28125	2.8046875	3.00004069	1.37504069
-2.84375	3.734375	1.5	1.714441935	2.039962769
1.953125	-0.8125	3.453125	4.306335449	2.488627116
-1.1015625	3.9453125	3.9375	5.593369802	3.678005219
-0.0859375	2.890625	-0.7109375	0.343170166	1.364003499
-0.8515625	3.8671875	1.5	1.12199529	0.761318207
2.0703125	2.09375	0.0625	0.449813843	0.369084676
-2.71875	-2.2265625	1.8828125	1.900910695	2.923046112
-3.7109375	3.9453125	3.8125	6.316921234	5.354681651
1.421875	-1.7265625	2.71875	2.694449107	1.69575119
-2.5390625	-3.8046875	2.5390625	2.987758636	3.775519053
2.9453125	-0.5078125	-3.890625	5.773932139	8.470546722
1.875	-1.203125	-1.34375	0.925013224	2.396367391
3.484375	-2.953125	2.0546875	2.600672404	1.561609904
0.78125	0.0234375	-1.3125	0.625092824	2.235769908
0.625	2.3203125	-0.4140625	0.201864878	0.882854462
-3.0078125	-1.296875	-3.046875	3.883433024	8.133433024
-3.0390625	3.2734375	1.5234375	1.766516368	2.218339284
1.3046875	-3.328125	-3.90625	5.458872477	9.182830811
2.6171875	2.5703125	-3.203125	4.128444672	5.963080088
-3.7890625	2.65625	-3.21875	4.796859741	8.763005575
-0.4609375	-1.7421875	2.2109375	1.710353851	1.680405935
2.921875	3.8125	0.828125	1.242858887	0.081400553

Table 1: Initial decision variables and fitness values.

“efficientNondominatedSort” sorts individuals in a population using the efficient non-dominated sorting. It returns a list of fronts with individuals assigned. It first sorts the individuals from best to worst according to f_1 and assigns the first individual to front 1. It compares the next individual with individuals from the last individual in front 1. If it is not dominated by any of solutions in front 1, it is appended to the front 1. If it is dominated, repeat comparing with individuals in the next front and so on.

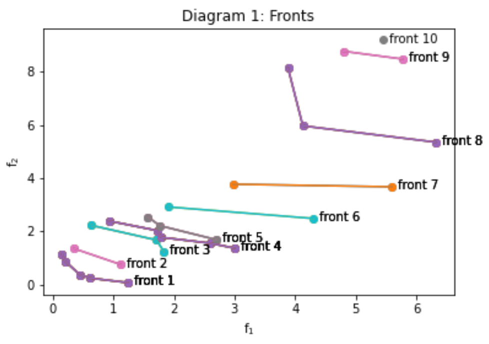
Table 2 shows the fitness values f_1 and f_2 and a front number of each individual in the initial population after sorting using the efficient non-dominated sorting.

f_1	f_2	Front number
0.147599538	1.156714122	1
0.201864878	0.882854462	1
0.449813843	0.369084676	1
0.607747396	0.253580729	1
1.242858887	0.081400553	1
0.343170166	1.364003499	2
1.12199529	0.761318207	2
0.625092824	2.235769908	3
1.710353851	1.680405935	3

1.833003998	1.243160248	3
0.925013224	2.396367391	4
1.714441935	2.039962769	4
1.791005452	1.780588786	4
2.600672404	1.561609904	4
3.00004069	1.37504069	4
1.567541758	2.545406342	5
1.766516368	2.218339284	5
2.694449107	1.69575119	5
1.900910695	2.923046112	6
4.306335449	2.488627116	6
2.987758636	3.775519053	7
5.593369802	3.678005219	7
3.883433024	8.133433024	8
4.128444672	5.963080088	8
6.316921234	5.354681651	8
4.796859741	8.763005575	9
5.773932139	8.470546722	9
5.458872477	9.182830811	10

Table 2: Fitness values and front numbers after the efficient non-dominated sorting.

Diagram 1 shows the individuals in the fronts after the sorting in the objective space.



The worst objective value in f_1:

$$f_1^* = \max\{f_1^i\}, i = 1, \dots, 28$$

$$f_1^* = 6.316921234130859$$

The worst objective value in f_2:

$$f_2^* = \max\{f_2^i\}, i = 1, \dots, 28$$

$$f_2^* = 9.182830810546875$$

The crowding distance of each individual in the initial population was calculated and assigned using "assignCrowdingDist" function.

This function takes a front as an input. It first assigns infinite to the first and last individual in the front. Then, it calculates the distance between two neighbours for each objective. The distances are divided by the distance between the minimum and maximum individuals and the number of objectives for normalization and averaging, respectively. Finally, the sum of the distances for every objective is assigned to each individual as an attribute.

Table 3 shows the fitness values f_1 and f_2, front number and crowding distance of all individuals in the initial population.

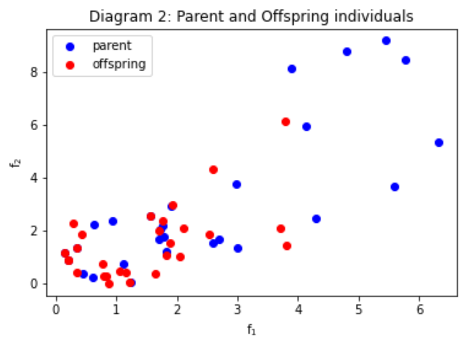
f_1	f_2	Front number	Crowding distance
0.147599538	1.156714122	1	inf
0.201864878	0.882854462	1	0.504197173
0.449813843	0.369084676	1	0.477890705
0.607747396	0.253580729	1	0.495802827
1.242858887	0.081400553	1	inf
0.343170166	1.364003499	2	inf
1.12199529	0.761318207	2	inf
0.625092824	2.235769908	3	inf
1.710353851	1.680405935	3	1
1.833003998	1.243160248	3	inf
0.925013224	2.396367391	4	inf
1.714441935	2.039962769	4	0.510130216
1.791005452	1.780588786	4	0.447728785
2.600672404	1.561609904	4	0.489869784
3.00004069	1.37504069	4	inf
1.567541758	2.545406342	5	inf
1.766516368	2.218339284	5	1
2.694449107	1.69575119	5	inf
1.900910695	2.923046112	6	inf
4.306335449	2.488627116	6	inf
2.987758636	3.775519053	7	inf
5.593369802	3.678005219	7	inf
3.883433024	8.133433024	8	inf
4.128444672	5.963080088	8	1
6.316921234	5.354681651	8	inf
4.796859741	8.763005575	9	inf
5.773932139	8.470546722	9	inf
5.458872477	9.182830811	10	inf

Table 3: Objective values, front numbers and crowding distances after the efficient non-dominated sorting.

"mate" and "mutate" functions are registered to the toolbox. "mate" calls "cxUniform" function with independent probability for each attribute to be exchanged set to 0.9 and "mutate" calls "mutFlipBit" function with independent probability for each attribute to be flipped set to 1.0/30 where 30 is the length of chromosome.

First, select 28 individuals from the initial population using "selTournamentDCD" function and clone it. Select a pair of individuals from the selected population and perform uniform crossover using the registered "mate" function. Then, each individual is mutated using the registered "mutate" function. After the crossover and mutation, the individuals are evaluated to update these fitness values.

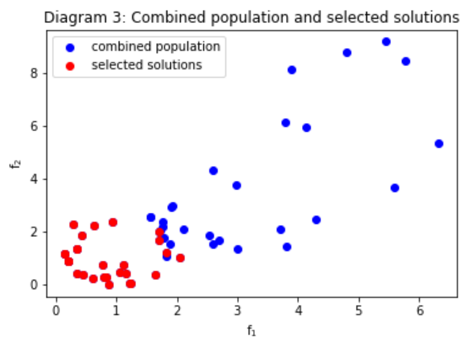
Diagram 2 shows the plot of the initial population (parents) and the population after the crossover and mutation (offspring individuals) in the objective space.



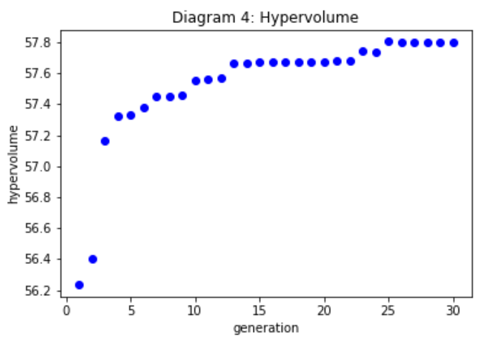
After combining the 28 offspring individuals with 28 parent individuals, 28 individuals are selected using "select" function registered to the toolbox which calls “NSGA_II” function.

NSGA_II function first sorts the individuals using “efficientNondominatedSort” function. After the sorting, it loops through the fronts from first to last and assigns crowding distance to each individual in the fronts until the number of individuals is over the number of individuals to select, which is 28 this time. The last front is sorted according to the crowding distance and individuals in it are selected from the first until the total number of individuals is the number of individuals to select. Finally, the function returns the selected individuals.

The diagram 3 shows the combined population and selected solutions in the objective space. The selected solutions are highlighted in red.



The above steps are repeated for 30 times. The diagram 4 shows the plot of hypervolume over the 30 generations. $(f_1{}^*, f_2{}^*)$ obtained earlier is used as the reference point to calculate the hypervolume using “hypervolume” function from DEAP library.



For two-objective minimization optimization problem, when there are n solutions in a pareto front, $(f_1(i), f_2(i)), \ldots\ (f_1(n), f_2(n));\ i = 1, \ldots, n$, and a reference point (r_1, r_2), the hypervolume can be obtained with the following equation.

$$\begin{aligned} h &= \left(r_1 - f_1(1)\right) \times \left(r_2 - f_2(1)\right) \\ &+ \left(r_1 - f_1(2)\right) \times \left(f_2(1) - f_2(2)\right) \\ &+ \cdots + \left(r_1 - f_1(n)\right) \times \left(f_2(n-1) - f_2(n)\right) \\ &= r_1 r_2 - r_2 f_1(1) - r_1 f_2(n) \\ &+ \sum_{i=1}^{n} f_1(i) f_2(i) - \sum_{i=1}^{n-1} f_1(i+1) f_2(i) \end{aligned}$$

When comparing two solutions, the difference in hypervolume can be obtained with the following equation.

$$\begin{aligned} h - h' &= \sum_{i=1}^{n} f_1(i) f_2(i) - \sum_{i=1}^{n} f_1{}'(i) f_2{}'(i) \\ &+ \sum_{i=1}^{n-1} f_1{}'(i+1) f_2{}'(i) - \sum_{i=1}^{n-1} f_1(i+1) f_2(i) \\ &+ r_2\left(f_1'(1) - f_1(1)\right) + r_1\left(f_2{}'(n) - f_2(n)\right) \end{aligned}$$

If we assume $r_1 = r_2 = r$, and there are two pareto fronts with solutions $p_1 = \{(0.1,0.9), (0.3,0.7), (0.5,0.5), (0.7,0.3), (0.9,0.1)\ \}$ and $p_1 = \{(0,1), (0.25,0.75), (0.5,0.5), (0.75,0.25), (1,0)\}$, the difference in hypervolume between these two pareto fronts is $0.2r - 0.425$. Thus, hypervolume of p_1 is larger than the hypervolume of p_2 when $r > 2.125$, these pareto fronts have the same hypervolume when $r = 2.125$ and the hypervolume of p_1 is smaller than p_2 when $r < 2.125$. It shows that an algorithm can be better or worse depending on the reference point when comparing solution sets obtained by different algorithms.

Appendix

Python code

(The efficient non-dominated sorting function is highlighted in yellow)

```
import random
import math
import numpy
from sympy.combinatorics.graycode import gray_to_bin
import matplotlib.pyplot as plt
from itertools import chain
from operator import attrgetter
from prettytable import PrettyTable
from deap import base
from deap.benchmarks.tools import hypervolume
from deap import creator
from deap import tools

# Create two objectives minimizing fitness class called "FitnessMin"
creator.create("FitnessMin", base.Fitness, weights=(-1.0, -1.0))
# Create an Individual class inherit from list with fitness attribute
creator.create("Individual", list, fitness=creator.FitnessMin)

random.seed(1004)

NGEN = 30
popSize = 28

# define function to calculate two objective values f1 and f2
def calcFitness(individual, table=None):
    sep=separatevariables(individual)
    x1=sep[0]
    x2=sep[1]
    x3=sep[2]
    f1=((x1/2.0)**2+(x2/4.0)**2+x3**2.0)/3.0
    f2=((x1/2.0-1.0)**2+(x2/4.0-1.0)**2+(x3-1.0)**2.0)/3.0
    if (table):
        table.add_row([x1,x2,x3,f1,f2])
    return f1,f2

# define function to convert gray code to real number
def chrom2real(c):
    indasstring=''.join(map(str, c))
    degray=gray_to_bin(indasstring)
    numasint=int(degray, 2) # convert to int from base 2 list
    numinrange=-4.0+8.0*numasint/2**10
    return numinrange

# define function to separate an individual into 3 decision variables
def separatevariables(v):
    sep = []
    for i in range (0,3):
        sep.append(chrom2real(v[i*10:i*10+10]))
    return sep

# define function to select individuals using efficient elitist non-dominated sorting genetic algorithm
def NSGA_II(individuals, k):
    fronts = efficientNondominatedSort(individuals)
    n = 0
    chosen = []
    for i, front in enumerate(fronts,1):
        if (k > n):
            assignCrowdingDist(front)
            chosen.append(front)
            n += len(front)
        else:
            break

    pop = list(chain(*chosen[:-1]))
    k = k - len(pop)
    if k > 0:
        sorted_front = sorted(chosen[-1], key=attrgetter("fitness.crowding_dist"), reverse=True)
        pop.extend(sorted_front[:k])
    return pop

# define function to sort individuals using efficient non-dominated sorting
def efficientNondominatedSort(individuals):
    individuals.sort(key=lambda x: x.fitness.values[0])
    fronts = [[individuals[0]]]

    for ind in individuals[1:]:
        for front in fronts:
            dominate = False
            for sol in reversed(front):
                if sol.fitness.dominates(ind.fitness):
                    dominate = True
                    break
            if not dominate:
                front.append(ind)
                break
        if dominate:
            fronts.append([ind])

    return fronts

# define function to assign crowding distance to each individual in a front
def assignCrowdingDist(individuals):
    distances = [0.0] * len(individuals)
    crowd = [(ind.fitness.values, i) for i, ind in enumerate(individuals)]
    nobj = len(individuals[0].fitness.values)

    for i in range(nobj):
        crowd.sort(key=lambda element: element[0][i])
        distances[crowd[0][1]] = float("inf")
        distances[crowd[-1][1]] = float("inf")
        if crowd[-1][0][i] == crowd[0][0][i]:
            continue
        norm = nobj * float(crowd[-1][0][i] - crowd[0][0][i])
        for prev, cur, next in zip(crowd[:-2], crowd[1:-1], crowd[2:]):
            distances[cur[1]] += (next[0][i] - prev[0][i]) / norm

    for i, dist in enumerate(distances):
        individuals[i].fitness.crowding_dist = dist

# register functions to the toolbox
toolbox = base.Toolbox()
# attr_bool function which returns 0 or 1 with equal probability
toolbox.register("attr_bool", random.randint, 0, 1)
# individual function to generate an individual consisting of 10*3 attr_bool elements
# 10*3 = 10 bits * 3 decision variables
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, 10*3)
# population function to generate a list of individuals
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
# evaluate function which calls calcFitness
toolbox.register("evaluate", calcFitness)
```

```
# mate function using uniform crossover at a probability of 0.9
toolbox.register("mate", tools.cxUniform, indpb=0.9)
# mutate function using flip bit mutation at a probability of 1.0/30
# the probability is set to be inversely proportional to the length of the chromosome (3*10)
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/30)
toolbox.register("select", NSGA_II)

pop = toolbox.population(n=popSize)
table = PrettyTable(['x1','x2','x3','f1','f2'])

# Evaluate the intial individuals
fitnesses = toolbox.map(toolbox.evaluate, pop, [table]*len(pop))
for ind, fit in zip(pop, fitnesses):
    ind.fitness.values = fit

print(table)

# sort initial individuals using efficient non-dominated sorting
fronts = efficientNondominatedSort(pop)
table = PrettyTable(['f1','f2','front number'])
plt.title('Diagram 1: Fronts')
plt.xlabel(r'f$_1$')
plt.ylabel(r'f$_2$')
for i, front in enumerate(fronts,1):
    for ind in front:
        table.add_row([ind.fitness.values[0],ind.fitness.values[1],i])
        plt.plot([ind.fitness.values[0] for ind in front],[ind.fitness.values[1] for ind in front])
        plt.scatter([ind.fitness.values[0] for ind in front],[ind.fitness.values[1] for ind in front])
        plt.text(front[-1].fitness.values[0]+0.1,front[-1].fitness.values[1]-0.1, "front %d" %i)
print(table)
plt.savefig('d1.png')
plt.show()

# calculate the worst objective values in f1 and f2
f1max = numpy.max([ind.fitness.values[0] for ind in pop])
f2max = numpy.max([ind.fitness.values[1] for ind in pop])
print('f1*: ',f1max)
print('f2*: ',f2max)

table = PrettyTable(['f1','f2','front number','crowding distance'])
for i, front in enumerate(fronts,1):
#     calculate crowding distance of each individual in each front
    assignCrowdingDist(front)
    for ind in front:
        table.add_row([ind.fitness.values[0],ind.fitness.values[1],i,ind.fitness.crowding_dist])
print(table)

pop = list(chain(*fronts[:]))

# select individuals for reproduction using tournament selection
offspring = tools.selTournamentDCD(pop, len(pop))
offspring = [toolbox.clone(ind) for ind in offspring]

#make pairs of all (even,odd) in offspring
for ind1, ind2 in zip(offspring[::2], offspring[1::2]):
#     perform crossover using the registered mate function
    toolbox.mate(ind1, ind2)
#     perform mutation using the registered mutate function
    toolbox.mutate(ind1)
    toolbox.mutate(ind2)
#     remove the previously calculated fitness values
    del ind1.fitness.values, ind2.fitness.values
```

```
# Evaluate the individuals with an invalid fitness
fitnesses = toolbox.map(toolbox.evaluate, offspring)
for ind, fit in zip(offspring, fitnesses):
    ind.fitness.values = fit

parent = numpy.array([ind.fitness.values for ind in pop])
child = numpy.array([ind.fitness.values for ind in offspring])

plt.title('Diagram 2: Parent and Offspring individuals')
plt.xlabel(r'f$_1$')
plt.ylabel(r'f$_2$')
plt.scatter(parent[:,0], parent[:,1], c="b", label='parent')
plt.scatter(child[:,0], child[:,1], c="r", label='offspring')
plt.legend()
plt.savefig('d2.png')
plt.show()

# select 28 individuals from combined population of parents and offspring individuals using the registered select function
pop = toolbox.select(pop + offspring, popSize)
combine = numpy.concatenate((parent, child))
selected = numpy.array([ind.fitness.values for ind in pop])
plt.title('Diagram 3: Combined population and selected solutions')
plt.xlabel(r'f$_1$')
plt.ylabel(r'f$_2$')
plt.scatter(combine[:,0], combine[:,1], c="b", label='combined population')
plt.scatter(selected[:,0], selected[:,1], c="r", label='selected solutions')
plt.legend()
plt.savefig('d3.png')
plt.show()

# calculate the hypervolume of the first generation using (f1max, f2max) as a reference point
hv = [hypervolume(pop,[f1max,f2max])]

# repeat the generational process
for gen in range(2, NGEN+1):
    offspring = tools.selTournamentDCD(pop, len(pop))
    offspring = [toolbox.clone(ind) for ind in offspring]

    for ind1, ind2 in zip(offspring[::2], offspring[1::2]):
        toolbox.mate(ind1, ind2)
        toolbox.mutate(ind1)
        toolbox.mutate(ind2)
        del ind1.fitness.values, ind2.fitness.values

    fitnesses = toolbox.map(toolbox.evaluate, offspring)
    for ind, fit in zip(offspring, fitnesses):
        ind.fitness.values = fit

    pop = toolbox.select(pop + offspring, popSize)
    hv.append(hypervolume(pop, [f1max, f2max]))


plt.title('Diagram 4: Hypervolume')
plt.xlabel('generation')
plt.ylabel('hypervolume')
plt.scatter(range(1,NGEN+1), hv, c="b")
plt.savefig('d4.png')
plt.show()
```