CSE/IEE 598: Bio-Inspired AI and Optimization
Mini Project 3

Tanmay Khandait
Student ID : 1219385830
CIDSE, Arizona State University
tkhandai@asu.edu

March 16, 2022

1 Introduction

Multi-Objective Optimization is an optimization process where the goal is to simultaneously optimize (minimize/maximize) multiple objectives. The solutions to a multiobjective optimization problem, are a set of optimal, non-dominated solutions, are trade-offs between multiple objectives. Such a set is called a pareto-optimal solutions. Non-dominated solutions are such solutions where none of the objective functions can be improved further without deteriorating the value of some other objective function [1].

In this report, I have implemented the NSGA2 algorithm which is a multi-objective Genetic Algorithm (MoGA). NSGA2 has been widely used to solve multi-objective optimization problems in various fields including economics and engineering [2]. The NSGA2 algorithm efficiently converges (exactly or as close as possible) to the pareto optimal front while ensuring that there is diversity in the set of solutions. Some of the key features of the NSGA algorithms are:

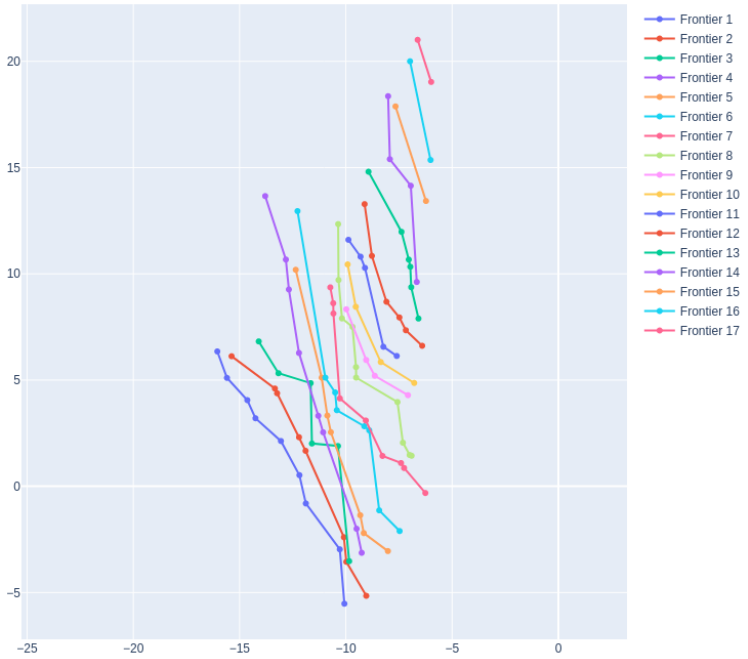


Figure 1: Fast Nondominated Sorting Approach

- **Fast Nondominated Sorting Approach**: The algorithm proposed by Deb et. al. [3] sorts the population based on their fitness based on its rank and crowding distance. First, the non-dominated solutions are obtained

and stored in a set. This set is assigned a Rank of 1, and removed from the population. The non-dominated solution in this subset of population is obtained, assigned a rank of 2 and removed from the set. This process is continued until there are no points left in the population.

Once this is done, we have a set of points belonging to various rank. Every candidate solution belonging to a particular rank is assigned a crowding distance based on its distance from its neighbors (having the same rank).

This ranking along with its crowding distance is used to determine the fitness while performing standard genetic algorithms operations. Among two points with different ranks, the solution with a lower rank is preferred. Among solutions with the same rank, the one with high crowding distance is preferred. An example is how in Figure 1

- **Elitism**: When the standard genetic algorithm operators are performed, we get a population of double the size which includes the original population from that generation along with the solutions obtained from selection, crossover and mutation. To recuce the population, the candidates with higher ranks are passed on to generation. In case there is such a rank where not all the candidate solutions can be accommodated, the ones with higher crowding distances are passed on to the next generation.

  This strategy ensures that individuals with higher ranks as well as the boundary solutions and spaced out solutions are ensured. This strategy is effective in maintaining elitism and diversity in the solution set.

The core algorithms of NSGA2 is as follows. First, the initial generation is assigned fitness based on its nondomination and crowding distance and has N candidate solutions. We then do a crowded distance based tournament selection to produce N more candidate solution. Standard SBX crossover and polynomial mutation operator. We now have a population 2N individuals. N of these are chosen based on their rank and crowding distance. The described repeated again with the new population until the termination criteria is satisfied.

The current code has been developed exactly from [3] and the experiments were replicated. In this report, the results are presented on 3 test functions, namely, Schaffer's study (SCH) [4], Kursawe's study (KUR) [5], and test example (TE) from class discussions). These test functions are presented in Table 1.

Problem	n	Variable Bounds	Type	Objective Function	Comments
SCH	1	[-1000, 1000]	Min. Min.	$f_1(x) = x^2$ $f_2(x) = (x-2)^2$	Convex
TE	1	[0,1]	Max. Max.	$f_1(x) = xsin(10\pi x)$ $f_2(x) = 2.5xcos(3\pi x)$	Non Convex, disconnected
KUR	3	[-5, 5]	Min. Min.	$f_1(x) = \sum_{n=1}^{n-1}(10\exp(-0.2\sqrt{x_i^2 + X_{i+1}^2}))$ $f_2(x) = \sum_{n=1}^{n}(\lvert x_i\rvert^{0.8} + 5sin(x_i^3))$	Non Convex

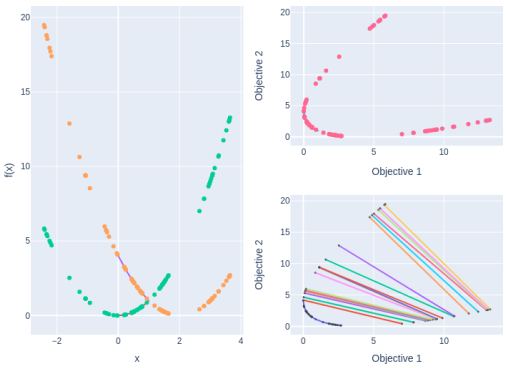
Table 1: Optimization Problems

2 Schaffer's Study (SCH)

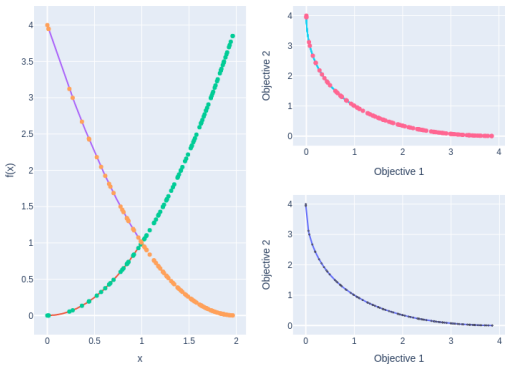
The Schaffer's study is 1 variable problem with minimization of both the objective functions as defined in Table 1. We show three set of plots during different generations where each set of plot has the solution set from that generation plotted on the objective function, the solution set overlapped with the plot between two objectives and the various pareto frontiers of that solution set. We can observe that the solutions are spread out on the frontier and we an also validate that the final population has solutions that minimize both the objectives. This result is similar to that from the reference paper [3].

3 Test Example (TE)

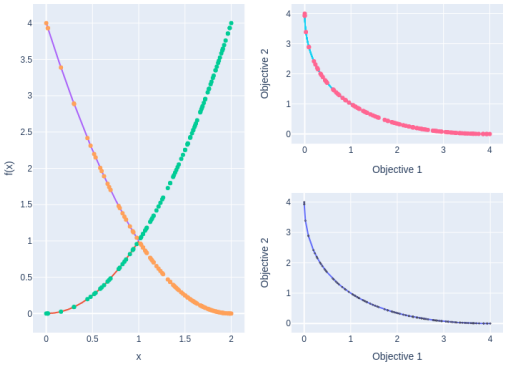
The test example in 1 variable problem with maximization of both the objective functions as defined in Table 1. We show three set of plots during different generations where each set of plot has the solution set from that generation plotted on the objective function, the solution set overlapped with the plot between two objectives and the various pareto frontiers of that solution set. We can observe that the solutions are spread out on the frontier and we an also validate that the final population has solutions that maximized both the objectives.



(a) Generation 4



(b) Generation 50



(c) Generation 100

Figure 2: Plots for Schaffer's Study[4]. For each sub figure, the plot on the left overlaps the candidate solution and their objective values on the objective functions. The top plot on the right overlaps the objective function values on the true objective function values and bottom plot on the right shows the pareto frontiers.

3.1 Comments

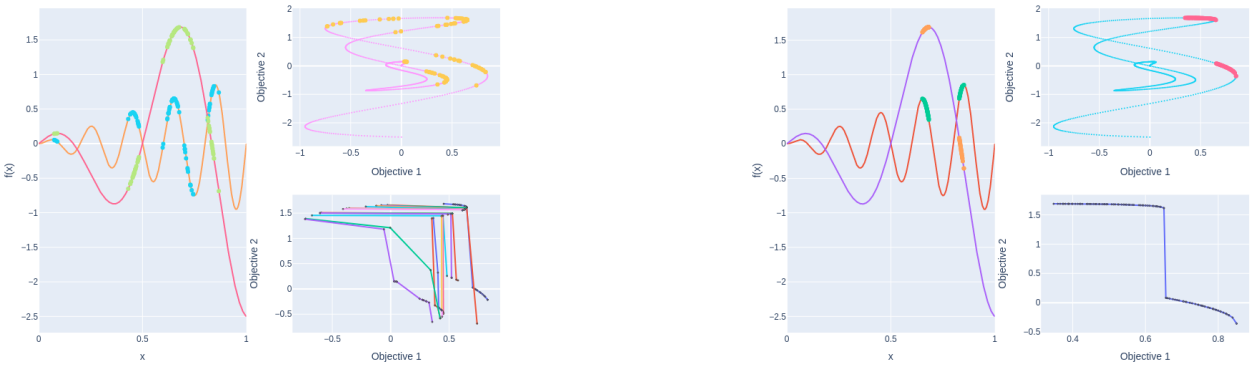
This was indeed a tough problem to solve and I had to incorporate changes, the main one being the ability to handle out-of-bound solutions. The alleles of candidate solutions which were out of bounds during the crossover and mutation were replaced with a random value from the bounds. IN my experience, this approach worked the best. Some other approached that I tried are to trim the solutions to not exceed values, but this often gave me results which were not optimal. I have to investigate why that case is, and I will do it as part of developing this tool.

4 Kursawe's Study (SCH)

The Kursawe's study is a 3 variable problem with minimization of both the objective functions as defined in Table 1. We show three set of plots during different generations where each set of plot has the solution set from that generation plotted on the objective function, the solution set overlapped with the plot between two objectives and the various pareto frontiers of that solution set. We can observe that the solutions are spread out on the frontier and we an also validate that the final population has solutions that minimize both the objectives. This result is similar to that from the reference paper [3].

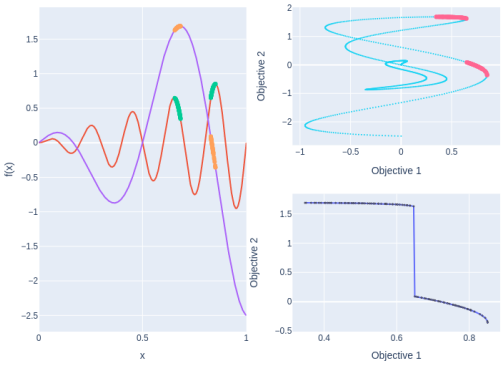
4.1 Comments

My implementation of algorithm was working correctly, however I could get to solve this problem initially. This made me investigate and tune the crossover rate and mutation rate. For this problem, I spent a lot of time tuning the parameters and running them for larger number of generations in order to replicate the results from the paper [3].



(a) Generation 0

(b) Generation 50



(c) Generation 100

Figure 3: Plots for Test Example (TE). For each sub figure, the plot on the left overlaps the candidate solution and their objective values on the objective functions. The top plot on the right overlaps the objective function values on the true objective function values and bottom plot on the right shows the pareto frontiers.

5 Discussion

In all the three problems that were being tried to solve (Table 1, we can observe that we get a diverse set of solutions in the optimal pareto front. This algorithm is interesting in how the initial effort is in reducing the ranks and maintaining diversity. Basically, the algorithms tries to find a non-dominating set of solution which has diverse set of individuals. All of these algorithms were run 10 times and the best plot was picked up. The main motivation between doing 10 runs was the ability to come up with some statistics on the solution set.

6 Future Work

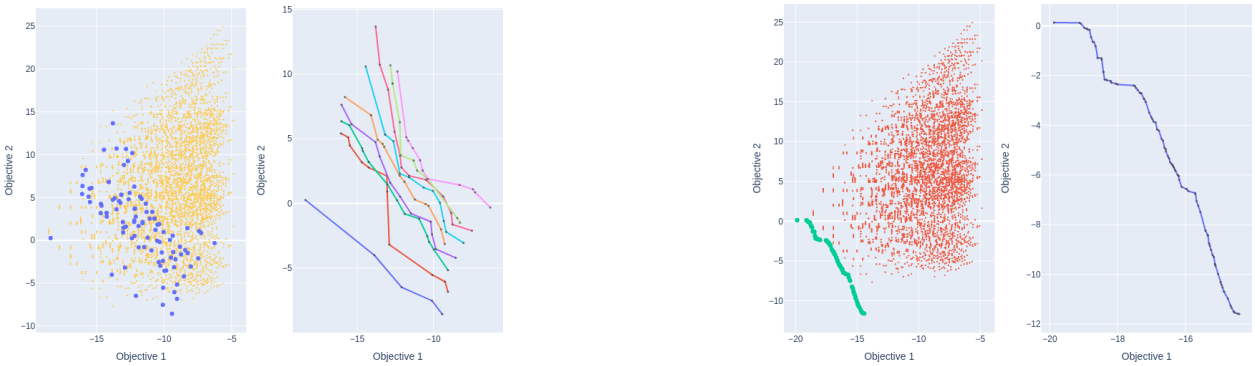
My code is very modular with easy way to define the objective functions. I plan to make a toolbox out of this code by optimizaing some of the operations in order to reduce the run time. The current status is that the code is ready but needs unit test cases and the documentation to be written. I have added features where a seed specified seed can be used in order to reproduce the results for analysis. I have used this current version for this porject and is also publicly available on my github repository at `https://github.com/DaitTan/super-funicular/tree/main/Project%202`.

References

[1] Kuang-Hua Chang. Chapter 19 - multiobjective optimization and advanced topics. In Kuang-Hua Chang, editor, *e-Design*, pages 1105–1173. Academic Press, Boston, 2015.

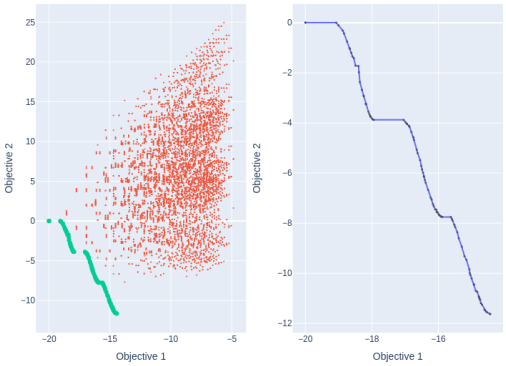
[2] Ankita Golchha and Shahana Gajala Qureshi. Non-dominated sorting genetic algorithm-ii@ a succinct survey. 2015.

[3] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Trans. Evol. Comput.*, 6:182–197, 2002.



(a) Generation 0

(b) Generation 38



(c) Generation 500

Figure 4: Plots for Kursawe's study [5]. For each sub figure, the plot on the left overlaps the objective function values on the true objective function values and the plot on the right shows the pareto frontiers.

[4] J. David Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, page 93–100, USA, 1985. L. Erlbaum Associates Inc.

[5] Frank Kursawe. A variant of evolution strategies for vector optimization. In *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, PPSN I, page 193–197, Berlin, Heidelberg, 1990. Springer-Verlag.

7 Codes

7.1 Main Driver Code

```
from audioop import cross
from dataclasses import replace
import enum
from operator import le
import operator
from unicodedata import name
import numpy as np
from numpy.random import default_rng
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import itertools
import pandas as pd
from tqdm import tqdm
import plotly.graph_objects as go
import pathlib
import os
import pickle

class Objectives:
    def __init__(self, objectives):
        self.signs = np.ones((len(objectives)))
        for iterate in range(len(objectives)):
            if objectives[iterate]["type"] == "Maximize":
                self.signs[iterate] = -1

        self.objectives = objectives

    def evaluate(self, population):
        m, _ = population.shape
        sol = np.zeros((m, len(self.objectives)))

        for iterate in range(len(self.objectives)):
            sol[:, iterate] = self.signs[iterate] * self.objectives[iterate]["function"](population)

        return sol


class Frontiers:
    # class_id = itertools.count(1,1)
    def __init__(self, index):
        self.frontier_id = index
        self.points_in_frontier = []


    def add(self, serial_number):
        self.points_in_frontier.append(serial_number)


class FrontiersFactory:
    def __init__(self) -> None:
        self.counter = itertools.count(1,1)
    def create(self):
        index = next(self.counter)
        return Frontiers(index)
```

```
class NonDominatedSorting:
    def __init__(self, population):
        self.points_frontier_class = []
        frontier_factory = FrontiersFactory()
        frontier = frontier_factory.create()

        for iterate_1 in range(len(population.population)):
            for iterate_2 in range(len(population.population)):
                # print("********")
                # print(population.population[iterate_1].serial_number)
                # print(population.population[iterate_2].serial_number)
                # print("********")
                if self.dominates(population.population[iterate_1].corres_eval,
                                  population.population[iterate_2].corres_eval):

                    population.population[iterate_1].S.add(
                                        population.population[iterate_2].serial_number
                                        )

                elif self.dominates(population.population[iterate_2].corres_eval,
                                  population.population[iterate_1].corres_eval):

                    population.population[iterate_1].n += 1

            if population.population[iterate_1].n == 0:
                population.population[iterate_1].rank = 1
                frontier.add(population.population[iterate_1].serial_number)
            # print(population.population[iterate_1].S)
        all_frontiers = [frontier]
        iterate = 0
        while not len(all_frontiers[iterate].points_in_frontier)==0:
            Q = []
            for p_id in all_frontiers[iterate].points_in_frontier:
                point_p, index_p = population.fetch_by_serial_number(p_id)
                # print(list([point_p.S]))
                for q_id in list(point_p.S):
                    # print("Hh")
                    # print(q_id)
                    # print([iterate.serial_number for iterate in population.population])
                    point_q, index_q = population.fetch_by_serial_number(q_id)
                    population.population[index_q].n -= 1
                    if population.population[index_q].n == 0:
                        # print(population.population[index_q].rank)
                        population.population[index_q].rank = iterate + 2
                        # print(population.population[index_q].rank)
                        Q.append(q_id)

            # for x in all_frontiers[iterate].points_in_frontier:
            #     _, index = population.fetch_by_serial_number(x)

            #     population.population[index].frontier = all_frontiers[iterate].frontier_id

            iterate += 1
            # print("Length Q = {}".format(len(Q)))

            next_frontier = frontier_factory.create()
            for id in Q:
                next_frontier.add(id)
```

```
            all_frontiers.append(next_frontier)

        self.all_frontiers = all_frontiers[:-1]

    def crowding_distance(self, population):
        all_eval = np.array([iterate.corres_eval for iterate in population.population])

        for _, frontiers in enumerate(self.all_frontiers):
            cardinality_r = len(frontiers.points_in_frontier)
            evaluations = []
            sr_num = []
            for iterate in frontiers.points_in_frontier:
                point,_ = population.fetch_by_serial_number(iterate)
                evaluations.append(point.corres_eval)
                sr_num.append(point.serial_number)
            evaluations = np.array(evaluations)
            sr_num = np.array(sr_num)
            for objective_num in range(population.num_objectives):

                sub_evaluations = evaluations[:, objective_num]

                sort_indices = np.argsort(sub_evaluations)

                sub_evaluations = sub_evaluations[sort_indices]
                # print(sub_evaluations)
                sr_num = sr_num[sort_indices]

                _, first_index = population.fetch_by_serial_number(sr_num[0])
                _, last_index = population.fetch_by_serial_number(sr_num[-1])

                population.population[first_index].d = float('inf')
                population.population[last_index].d = float('inf')
                low_val = min(all_eval[:, objective_num])
                high_val = max(all_eval[:, objective_num])
                for i in range(1, cardinality_r-1):

                    _, index_mid = population.fetch_by_serial_number(sr_num[i])
                    # index_high_d, _ = population.fetch_by_serial_number(sr_num[i+1])
                    # population.population[index_mid].d += ((
                    #                         index_high_d.d - index_low_d.d
                    #                     )) / (high_val - low_val + 1e-30)

                    population.population[index_mid].d += (abs(
                                            sub_evaluations[i+1] - sub_evaluations[i-1]
                                        )) / (high_val - low_val + 1e-60)
                    # print((evaluations_sub[cardinality_r-1], evaluations_sub[0]))
                # print([iterate.d for iterate in population.population])
        # print(f)
        # print("******************************")

    def dominates(self, p, q):
        comparison = p < q
        if np.all(comparison):
            return True
        else:
            return False

class SolutionVecProps:
    def __init__(self, sol_vec, corres_eval, index):
```

```
        self.serial_number = index
        self.rank = 0
        self.sol_vec = sol_vec
        self.corres_eval = corres_eval
        self.S = set()
        self.n = 0
        self.d = 0
        self.frontier = -1

class SolutionVecPropsFactory:
    def __init__(self) -> None:
        self.counter = itertools.count()
    def create(self, sol_vec, corres_eval):
        index = next(self.counter)
        return SolutionVecProps(sol_vec, corres_eval, index)

class Population:
    def __init__(self, population_size, num_variables, bounds,
                 objectives, seed, defined_pop = [], generate = True):
        self.seed = seed
        self.rng = default_rng(seed)
        self.population_size = population_size
        self.num_objectives = len(objectives.objectives)
        self.num_variables = num_variables
        self.bounds = bounds
        self.objectives = objectives
        if generate == True:
            sol_vectors = self.generate_random_legal_population()
            # sol_vectors = np.array([[0.913, 2.181],
            #                         [0.599, 2.450],
            #                         [0.139, 1.157],
            #                         [0.867, 1.505],
            #                         [0.885, 1.239],
            #                         [0.658, 2.040],
            #                         [0.788, 2.166],
            #                         [0.342, 0.756]])
            evaluations = self.evaluate_objectives(sol_vectors)
            self.population = self.generate_population(sol_vectors, evaluations)
        else:
            sol_vectors = defined_pop
            evaluations = self.evaluate_objectives(sol_vectors)
            self.population = self.generate_population(sol_vectors, evaluations)


    def get_all_sol_vecs(self):
        return np.array([iterate.sol_vec for iterate in self.population])

    def get_all_evals(self):
        return np.array([iterate.corres_eval for iterate in self.population])

    def get_all_serial_numbers(self):
        return [iterate.serial_number for iterate in self.population]

    def generate_population(self, sol_vectors, evaluations):
        population = []
        solVecProp = SolutionVecPropsFactory()
        for sol_vec, corres_eval in zip(sol_vectors, evaluations):
            pointProp = solVecProp.create(sol_vec, corres_eval)
            population.append(pointProp)
```

```
        return population

    def fetch_by_serial_number(self, target):
        for iterate, pop in enumerate(self.population):
            if pop.serial_number == target:
                return pop, iterate


    def generate_random_legal_population(self):
        population = self.rng.random((self.population_size, self.num_variables))
        for iterate in range(self.num_variables):
            lower_b = self.bounds[iterate][0]
            upper_b = self.bounds[iterate][1]
            population[:, iterate] = (population[:, iterate] * (upper_b - lower_b)) + lower_b

        return population

    def plotPopulation(self):
        fig = go.Figure()
        evaluations = self.get_all_evals()
        point_caption = (["Point {}".format(i) for i in self.get_all_serial_numbers()])
        fig.add_trace(go.Scatter(
            x = 1*evaluations[:,0],
            y = 1*evaluations[:,1],
            mode = "markers",
            text = point_caption
        ))
        fig.update_layout(
            width = 800,
            height = 800,
            title = "fixed-ratio axes"
        )
        fig.update_yaxes(
            range = [-12, 2],
            scaleanchor = "x",
            scaleratio = 1,
        )
        fig.update_xaxes(
            range = [-20,-14],
            scaleanchor = "x",
            scaleratio = 1,
        )
        fig.show()

    def plotPopulationwithFrontier(self):
        fig = go.Figure()
        all_frontiers = NonDominatedSorting(self)
        print(len(all_frontiers.all_frontiers))
        for rank, frontiers in enumerate(all_frontiers.all_frontiers):
            evaluations = []
            sr_num = []
            for iterate in frontiers.points_in_frontier:
                point,_ = self.fetch_by_serial_number(iterate)
                evaluations.append(point.corres_eval)
                sr_num.append(point.serial_number)
            evaluations = np.array(evaluations)
            df = pd.DataFrame(dict(
                    x = 1*evaluations[:,0],
                    y = 1*evaluations[:,1],
```

```
            ))
        print(df)
        point_caption = (["Point {}".format(i) for i in sr_num])
        fig.add_trace(go.Scatter(
        x = df.sort_values(by="x")["x"],
        y = df.sort_values(by="x")["y"],
        mode = "markers+lines",
        text = point_caption,
        name = "Frontier {}".format(rank + 1)
        ))

    fig.update_layout(
        width = 800,
        height = 800,
        title = "fixed-ratio axes"
    )
    fig.update_yaxes(
        range = [-12, 2],
        scaleanchor = "x",
        scaleratio = 1,
    )
    fig.update_xaxes(
        range = [-20,-14],
        scaleanchor = "x",
        scaleratio = 1,
    )
    fig.show()

def evaluate_objectives(self, sol_vectors):
    # population = self.population
    return self.objectives.evaluate(sol_vectors)

def thanos_kill_move(self):

    ranks = [iterate.rank for iterate in self.population]
    cd = [-1*iterate.d for iterate in self.population]
    serial_num = [iterate.serial_number for iterate in self.population]
    combined_array = np.array([ranks, cd, serial_num]).T.tolist()
    combined_array = np.array(sorted(combined_array, key = operator.itemgetter(0,1)))
    # print(combined_array)
    # print(f)
    selected_indices = combined_array[0:self.population_size,-1].astype(int)

    survivors = []
    for iterate in selected_indices:
        point, _ = self.fetch_by_serial_number(iterate)

        survivors.append(point.sol_vec)


    survivors = np.array(survivors)
    # for survivor in survivors:

    #     for iterate,j in enumerate(self.bounds):
    #         if survivor[iterate] < j[0]:
    #             survivors[iterate][0] = j[0]
    #         if survivor[iterate] > j[1]:
    #             survivors[iterate][1] = j[1]
    # print(survivors)
```

```
        # print(f)

        return Population(self.population_size, self.num_variables,
                          self.bounds, self.objectives, self.seed+3,
                          survivors, False)


    def find_bounds(self, ranks, target):
        lower_bound = self.binarySearch(ranks, target,True)
        upper_bound = self.binarySearch(ranks, target,False, lower_bound)
        return lower_bound, upper_bound

    def find_upper_bound(self, ranks, target):
        upper_bound = self.binarySearch(ranks, target,False)
        return upper_bound

    def binarySearch(self, inp, target, lowerBound, start = 0):
        left = start
        right = len(inp) - 1

        while left <= right:
            mid = (left + right) // 2
            # print(left, mid, right)
            # print(inp[left], inp[mid], inp[right])
            if inp[mid] == target:
                if inp[mid] == target:
                    if lowerBound:
                        if mid == 0:
                            return mid
                        elif inp[mid-1] != target:
                            return mid
                        else:
                            right = mid
                    else:
                        if mid == len(inp)-1:
                            return mid
                        elif inp[mid+1] != target:
                            return mid
                        else:
                            left = mid + 1

            elif inp[mid] < target:
                left = mid + 1

            else:
                right = mid - 1


class GARoutine:
    def __init__(self, population, seed) -> None:
        self.rng = default_rng(seed)

        self.sol_vec = population.get_all_sol_vecs()
        self.crowding_distance = np.array(
                          [iterate.d for iterate in population.population]
                      )
        self.rank = np.array(
                          [iterate.rank for iterate in population.population]
                      )
```

```
        self.size = len(population.population)
        self.bounds = population.bounds
        # print(self.sol_vec)
        # print(self.crowding_distance)
        # print(self.rank)

    def crowded_binary_tournament_selection(self, withReplacement = False):
        # print(self.size)
        size = self.size
        chosen = []
        for _ in range(2):

            tournament_draw = self.rng.choice(size, size = size, replace = withReplacement)
            # print(tournament_draw)
            if size % 2 == 0:
                for iterate in range(0, size, 2):
                    chosen.append(self.choose(tournament_draw[iterate], tournament_draw[iterate+1]))

            else:
                raise ValueError("Population Size should be Even integer.")
        # print(chosen)
        winners = []
        for i in chosen:
            winners.append(self.sol_vec[i, :])

        return np.array(winners)

    def choose(self, parent_1_index, parent_2_index):

        if self.rank[parent_1_index] != self.rank[parent_2_index]:
            rank_p1 = self.rank[parent_1_index]
            rank_p2 = self.rank[parent_2_index]
            return parent_1_index if rank_p1 < rank_p2 else parent_2_index
        elif self.crowding_distance[parent_1_index] != self.crowding_distance[parent_2_index]:
            cd_p1 = self.crowding_distance[parent_1_index]
            cd_p2 = self.crowding_distance[parent_2_index]
            return parent_1_index if cd_p1 > cd_p2 else parent_2_index
        else:
            toss = self.rng.random()
            return parent_1_index if toss < 0.5 else parent_2_index

    def sbx_crossover_operator(self, sol_vec, crossover_prob, p_curve_param, withReplacement = False):
        crossover_couples = self.rng.choice(self.size, size = self.size, replace = withReplacement)
        offsprings = []

        if self.size%2 == 0:
            for iterate in range(0, self.size, 2):
                offspring_1, offspring_2 = self.generate_offspring_from_SBX(sol_vec, crossover_couples[i
                                                    crossover_couples[iterate + 1],
                                                    p_curve_param, crossover_prob)
                offsprings.append(offspring_1)
                offsprings.append(offspring_2)
        else:
            raise ValueError("Population Size should be Even integer.")

        return np.array(offsprings)


    def generate_offspring_from_SBX(self, sol_vec, p1_index, p2_index, p_curve_param, crossover_prob):
```

```
        biased_toss = self.rng.random()
        if biased_toss <= crossover_prob:
            beta = self.calculate_beta(p_curve_param, biased_toss)
            # print("Crossover Took Place: {}, {}".format(biased_toss, crossover_prob))
            p1 = sol_vec[p1_index,:]
            p2 = sol_vec[p2_index, :]
            child_1 = 0.5 * ((p1 + p2) + beta*(p1-p2))
            child_2 = 0.5 * ((p1 + p2) - beta*(p1-p2))
            # print(child_1)
            # print(child_2)
            # print(f)

            offsprings_1 = child_1
            offsprings_2 = child_2

            for iterate in range(len(self.bounds)):

                if offsprings_1[iterate] < self.bounds[iterate][0] or offsprings_1[iterate] > self.bound

                    offsprings_1[iterate] = self.bounds[iterate][0] + self.rng.random() * (self.bounds[i



            for iterate in range(len(self.bounds)):

                if offsprings_2[iterate] < self.bounds[iterate][0] or offsprings_2[iterate] > self.bound

                    offsprings_2[iterate] = self.bounds[iterate][0] + self.rng.random() * (self.bounds[i

        else:
            offsprings_1 = sol_vec[p1_index,:]
            offsprings_2 = sol_vec[p2_index,:]

        return offsprings_1, offsprings_2

    def calculate_beta(self, p_curve_param, toss):
        # toss = self.rng.random()
        if toss <= 0.5:
            beta = (2*toss)**(1/(p_curve_param + 1))
        else:
            beta = (1/(2*(1-toss)))**(1/(p_curve_param + 1))

        return beta

    def polynomial_mutation_operator(self, sol_vec, bounds, mutation_prob, p_curve_param_mutation):
        offsprings = []

        bound_length = []
        for b in bounds:
            bound_length.append(b[1] - b[0])
        bound_length = np.array(bound_length)

# future edit needed here
        for iterate in range(self.size):
            biased_toss = self.rng.random()

            if biased_toss <= mutation_prob:
                delta_bar = self.calculate_delta_bar(p_curve_param_mutation, biased_toss)
                # mut_offspring = []
```

```
                # print(bound_length)
                mut_offspring = sol_vec[iterate,:] + ((bound_length) * delta_bar)
                for iterate in range(len(self.bounds)):

                    if mut_offspring[iterate] < self.bounds[iterate][0] or mut_offspring[iterate] > self

                        mut_offspring[iterate] = self.bounds[iterate][0] + self.rng.random() * (self.bou
                # print(f)
                # for iterate_2, b in bounds:

                #     if gen < b[]
                #     mut_offspring =

                    offsprings.append(mut_offspring)
            else:
                offsprings.append(sol_vec[iterate,:])


        return np.array(offsprings)


    def calculate_delta_bar(self, p_curve_param, toss):
        # toss = self.rng.random()
        if toss <= 0.5:
            delta_bar = ((2*toss)**(1/(p_curve_param + 1))) - 1
        else:
            delta_bar = 1 - ((2*(1-toss))**(1/(p_curve_param + 1)))

        return delta_bar
```

7.2 Code for Replicating SCH Test Function

```
import numpy as np
import pathlib

from driver import Objectives, Population, GARoutine, NonDominatedSorting
import pickle
from tqdm import tqdm
from numpy.random import default_rng

# def obj_1(pop):
#     x = pop[:, 0]
#     return x

# def obj_2(pop):
#     y = pop[:,1:]
#     # print(np.sum(y,1))
#     # print(f)
#     g = 1 + (9/29) * (np.sum(y,1))
#     h = 1 - (pop[:,0]/g)**2

#     return 1*g*h

# def obj_1(pop):
#     x = pop[:, 0]

#     return -1 * (x * (np.sin(10 * np.pi * x)))

# def obj_2(pop):
```

```
#     x = pop[:, 0]
#     return -1 * (2.5 * x * (np.cos(3 * np.pi * x)))

def obj_1(pop):
    x = pop[:, 0]

    return x**2

def obj_2(pop):
    x = pop[:, 0]
    return (x-2)**2


objective_1 = {}
objective_1["type"] = "Minimize"
objective_1["function"] = obj_1

objective_2 = {}
objective_2["type"] = "Minimize"
objective_2["function"] = obj_2


objectives_list = [objective_1, objective_2]
obj_name = "SCH"




objectives = Objectives(objectives_list)


population_size = 100
num_variables = 1
bounds = [[-1000, 1000]]*1

crossover_prob = 0.9
mutation_prob = 0.15

p_curve_param = 20
p_curve_param_mutation = 20

num_generations = 100
num_runs = 1




base_path = pathlib.Path()
result_directory = base_path.joinpath(obj_name)
result_directory.mkdir(exist_ok=True)

for run in tqdm(range(num_runs)):
    seed = 123458 + run
    rng = default_rng(seed)
    run_folder = result_directory.joinpath(obj_name + "_Run_"+str(run) + "_seed_"+str(seed))
    run_folder.mkdir(exist_ok=True)

    pop = Population(population_size, num_variables, bounds, objectives, rng.integers(1e16))
```

```
    file_name = obj_name + "_Run_" + str(run) +"_initial_pop.pkl"
    f = open(run_folder.joinpath(file_name), "wb")
    pickle.dump(pop,f)
    f.close()


    for generation in tqdm(range(num_generations)):


        fds = NonDominatedSorting(pop)
        fds.crowding_distance(pop)


        ga = GARoutine(pop, rng.integers(1e16))
        sel_pop = ga.crowded_binary_tournament_selection()
        crossover_offsprings = ga.sbx_crossover_operator(sel_pop, crossover_prob, p_curve_param)
        mut_offspring = ga.polynomial_mutation_operator(crossover_offsprings, bounds, mutation_prob, p_c

        new_sol_vecs = np.vstack((np.array(pop.get_all_sol_vecs()), mut_offspring))


        temp_extended_pop = Population(population_size, num_variables, bounds, objectives, rng.integers(
        fds = NonDominatedSorting(temp_extended_pop)
        fds.crowding_distance(temp_extended_pop)

        new_pop = temp_extended_pop.thanos_kill_move()
        pop = new_pop

        file_name = obj_name + "_Gen_" + str(generation) + "_Run_" + str(run) +".pkl"
        f = open(run_folder.joinpath(file_name), "wb")
        pickle.dump(pop.get_all_sol_vecs(),f)
        f.close()
```

7.2.1 Code for Replicating TE Test Function

```
import numpy as np
import pathlib

from driver import Objectives, Population, GARoutine, NonDominatedSorting
import pickle
from tqdm import tqdm
from numpy.random import default_rng

# def obj_1(pop):
#     x = pop[:, 0]
#     return x

# def obj_2(pop):
#     y = pop[:,1:]
#     # print(np.sum(y,1))
#     # print(f)
#     g = 1 + (9/29) * (np.sum(y,1))
#     h = 1 - (pop[:,0]/g)**2

#     return 1*g*h

def obj_1(pop):
    x = pop[:, 0]
```

```
    return -1 * (x * (np.sin(10 * np.pi * x)))

def obj_2(pop):
    x = pop[:, 0]
    return -1 * (2.5 * x * (np.cos(3 * np.pi * x)))



objective_1 = {}
objective_1["type"] = "Minimize"
objective_1["function"] = obj_1

objective_2 = {}
objective_2["type"] = "Minimize"
objective_2["function"] = obj_2


objectives_list = [objective_1, objective_2]
obj_name = "test_ex"




objectives = Objectives(objectives_list)


population_size = 100
num_variables = 1
bounds = [[0,1]]*1

crossover_prob = 0.9
mutation_prob = 0.15

p_curve_param = 20
p_curve_param_mutation = 20

num_generations = 100
num_runs = 1




base_path = pathlib.Path()
result_directory = base_path.joinpath(obj_name)
result_directory.mkdir(exist_ok=True)

for run in tqdm(range(num_runs)):
    seed = 123458 + run
    rng = default_rng(seed)
    run_folder = result_directory.joinpath(obj_name + "_Run_"+str(run) + "_seed_"+str(seed))
    run_folder.mkdir(exist_ok=True)

    pop = Population(population_size, num_variables, bounds, objectives, rng.integers(1e16))

    file_name = obj_name + "_Run_" + str(run) +"_initial_pop.pkl"
    f = open(run_folder.joinpath(file_name), "wb")
    pickle.dump(pop,f)
    f.close()
```

```
    for generation in tqdm(range(num_generations)):


        fds = NonDominatedSorting(pop)
        fds.crowding_distance(pop)


        ga = GARoutine(pop, rng.integers(1e16))
        sel_pop = ga.crowded_binary_tournament_selection()
        crossover_offsprings = ga.sbx_crossover_operator(sel_pop, crossover_prob, p_curve_param)
        mut_offspring = ga.polynomial_mutation_operator(crossover_offsprings, bounds, mutation_prob, p_c

        new_sol_vecs = np.vstack((np.array(pop.get_all_sol_vecs()), mut_offspring))


        temp_extended_pop = Population(population_size, num_variables, bounds, objectives, rng.integers(
        fds = NonDominatedSorting(temp_extended_pop)
        fds.crowding_distance(temp_extended_pop)

        new_pop = temp_extended_pop.thanos_kill_move()
        pop = new_pop

        file_name = obj_name + "_Gen_" + str(generation) + "_Run_" + str(run) +".pkl"
        f = open(run_folder.joinpath(file_name), "wb")
        pickle.dump(pop.get_all_sol_vecs(),f)
        f.close()
```

7.3 Code for Replicating KUR Test Function

```
import numpy as np
import pathlib

from driver import Objectives, Population, GARoutine, NonDominatedSorting
import pickle
from tqdm import tqdm
from numpy.random import default_rng

def obj_1(pop):
    x1 = pop[:, 0]
    x2 = pop[:, 1]
    x3 = pop[:, 2]
    return -10 * (np.exp(-0.2 * np.sqrt(x1**2 + x2**2)) + np.exp(-0.2 * np.sqrt(x2**2 + x3**2)))

def obj_2(pop):
    x1 = pop[:, 0]
    x2 = pop[:, 1]
    x3 = pop[:, 2]
    return (np.abs(x1)**0.8 + (5 * np.sin(x1**3))) + (np.abs(x2)**0.8 + (5 * np.sin(x2**3))) + (np.abs(x

objective_1 = {}
objective_1["type"] = "Minimize"
objective_1["function"] = obj_1

objective_2 = {}
objective_2["type"] = "Minimize"
objective_2["function"] = obj_2
```

```
objectives_list = [objective_1, objective_2]
obj_name = "KUR"



objectives = Objectives(objectives_list)


population_size = 100
num_variables = 3
bounds = [[-5,5]]*3

crossover_prob = 0.9
mutation_prob = 0.15

p_curve_param = 10
p_curve_param_mutation = 10

num_generations = 500
num_runs = 1



base_path = pathlib.Path()
result_directory = base_path.joinpath(obj_name)
result_directory.mkdir(exist_ok=True)

for run in tqdm(range(num_runs)):
    seed = 123458 + run
    rng = default_rng(seed)
    run_folder = result_directory.joinpath(obj_name + "_Run_"+str(run) + "_seed_"+str(seed))
    run_folder.mkdir(exist_ok=True)

    pop = Population(population_size, num_variables, bounds, objectives, rng.integers(1e16))

    file_name = obj_name + "_Run_" + str(run) +"_initial_pop.pkl"
    f = open(run_folder.joinpath(file_name), "wb")
    pickle.dump(pop,f)
    f.close()


    for generation in tqdm(range(num_generations)):


        fds = NonDominatedSorting(pop)
        fds.crowding_distance(pop)


        ga = GARoutine(pop, rng.integers(1e16))
        sel_pop = ga.crowded_binary_tournament_selection()
        crossover_offsprings = ga.sbx_crossover_operator(sel_pop, crossover_prob, p_curve_param)
        mut_offspring = ga.polynomial_mutation_operator(crossover_offsprings, bounds, mutation_prob, p_c

        new_sol_vecs = np.vstack((np.array(pop.get_all_sol_vecs()), mut_offspring))


        temp_extended_pop = Population(population_size, num_variables, bounds, objectives, rng.integers(
```

```
        fds = NonDominatedSorting(temp_extended_pop)
        fds.crowding_distance(temp_extended_pop)

        new_pop = temp_extended_pop.thanos_kill_move()
        pop = new_pop

        file_name = obj_name + "_Gen_" + str(generation) + "_Run_" + str(run) +".pkl"
        f = open(run_folder.joinpath(file_name), "wb")
        pickle.dump(pop.get_all_sol_vecs(),f)
        f.close()

pop.plotPopulation()
```