Mini-Project 3

Appendix

Introduction

In this mini-project, extension of a simple GA is done to perform a multi-objective optimization using Pareto ranking. By selecting individuals with low pareto rank (lower is better) for recombination and mutation, over the generations, the algorithm filters the non-dominating solutions from the population and finds an approximation of the Pareto frontier.

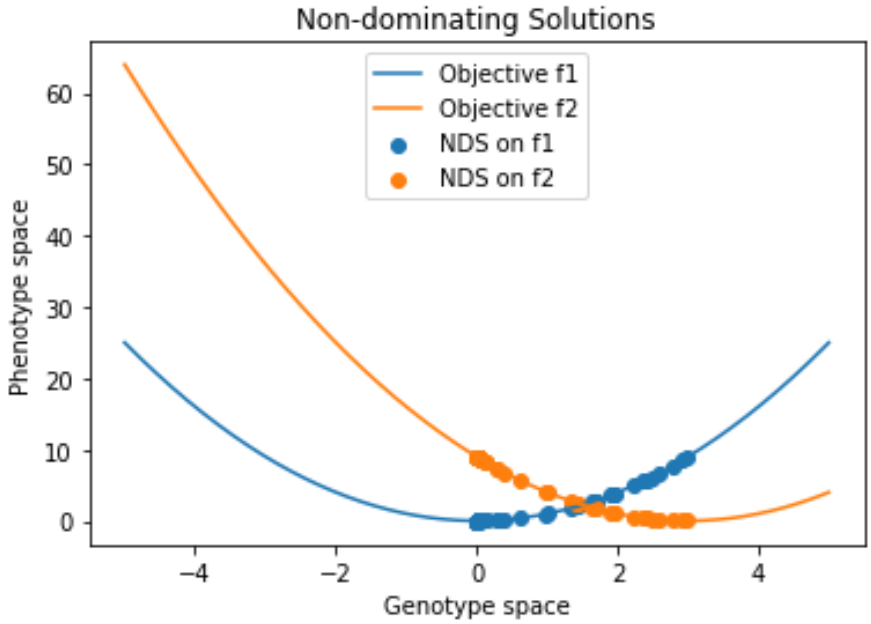
Methods

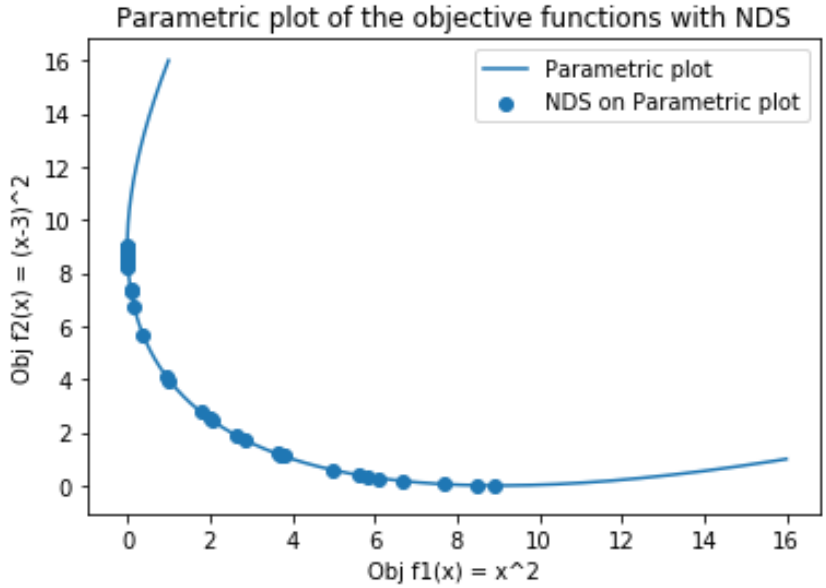
The objective functions are f1(x) = x^2 and f2(x) = (x-3)^2. From the objective functions' convex nature and their minima, it can be inferred that the Pareto frontier lies between 1 and 3. If the output of the algorithm is also between 1 and 3, it can be said that the algorithm correctly finds the Pareto frontier. The following steps are taken to find the Pareto frontier.

- 100 individuals in the genotype space are encoded as base-10 digits with chromosome size equal to 7.
- An object is used to represent the individuals which keeps track of the value and also its Pareto rank.
- Pairwise comparisons are done to calculate the Pareto rank. Lower the Pareto rank for an individual, higher the number of individuals that are dominated by it.
- As a parameter, the parent population size (equal to 30) is set. After sorting the population in the ascending order of their Pareto rank, the parent population is retained and the remaining individuals are deleted.
- Parent population undergoes one-point crossover and mutation according to probability parameters for recombination (P(recombination) = 0.7) and mutation (P(mutation) = 0.001).
- If a child lies outside the domain, it is ignored.
- Selecting by Pareto rank, recombination and mutation occur in a loop for 100 generations.
- Finally, the algorithm filters the non-dominating solutions one last time and finds the approximate Pareto frontier.

Results and Discussion

From the plot of objective functions and the Non-dominated solutions (NDS), we can see that the algorithm is able to find the approximate Pareto frontier, the solution space where there is a trade-off for prioritizing optimization of one objective function relative to others. It is clear from the plot below that there are no solutions for x<0 and x>3 that are better (lower) in all the objective functions than the ones between 0 and 3. In addition, we can characterize the trade-off between the objective functions with the parametric plot and see the non-dominating solutions on it.





Source Code

Mini-project 1 source code has been modified to include selection of low Pareto ranked individuals for the purposes of mini-project 3.

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python
"""
(Citing my previous work)
Title: Simple GA
Version: 1.0
Created on Sat Feb 4 2022
@author: Mahesh
"""

"""
Title: Pareto frontier estimation using Simple GA
Version: 2.0
Created on Sat Mar 12 2022
@author: Mahesh
"""
from cProfile import label
import numpy as np
import matplotlib.pyplot as plt

#Function to calculate fitness = xsin(10πx)+1
def calc_fitness1(x):
    return x ** 2

def calc_fitness2(x):
    return (x-3)**2

#Function for performing base-10 encoding
def gene_encode(x, chromosome_size):
    charlist = [char for char in str(x)]
    if charlist[0] == '-':
        del charlist[0]
        del charlist[1]
        returnlist = [int(char) for char in charlist]
        returnlist.insert(0, -1)
```

```
    else:
        del charlist[1]
        returnlist = [int(char) for char in charlist]
        returnlist.insert(0, 1)
    while len(returnlist) != chromosome_size:
        returnlist.append(0)
    return returnlist

#Function to decode base-10 genes
def gene_decode(gene):
    try:
        if len(gene) > 0:
            val = 0
            for i in range(1,len(gene)):
                val += gene[i]/10**(i-1)
            val *= gene[0]
            return val
    except:
        return gene

class Pop:
    pop_count = 0
    pop_list = []
    def __init__(self,value):
        self.id = Pop.pop_count
        self.value = value
        self.pareto_fitness = 0
        Pop.pop_count += 1
        Pop.pop_list.append(self)
    def __repr__(self):
                                    return     'id='+str(self.id)+'     ::
value='+str(gene_decode(self.value))+' :: pr='+str(self.pareto_fitness)

#Function to generate uniform population distribution across the domain
def gen_pop(pop_size,domain,chromosome_size):
    for i in range(pop_size):
                    pop   =   round(np.random.uniform(domain[0],   domain[1]),
chromosome_size-2)
        Pop(gene_encode(pop,chromosome_size))
```

```
def pareto_selection(parent_size,bool):
    completed_comparisions = []
    for i in Pop.pop_list:
        for j in Pop.pop_list:
              if [i.id,j.id] in completed_comparisions or [j.id,i.id] in
completed_comparisions or i.id == j.id:
                continue
                                if   calc_fitness1(gene_decode(i.value))   >
calc_fitness1(gene_decode(j.value))                                    and
calc_fitness2(gene_decode(i.value)) > calc_fitness2(gene_decode(j.value)):
                i.pareto_fitness += 1
                j.pareto_fitness -= 1 #lower pareto rank is better
                completed_comparisions.append([i.id,j.id])
    Pop.pop_list.sort(key=lambda x: x.pareto_fitness)
    del Pop.pop_list[parent_size:]
    if bool:
        for i in Pop.pop_list:
            i.pareto_fitness = 0

#Function for mutating base-10 genes
def mutation(pop):
    for gene in range(len(pop)):
        if gene == 0:
            if np.random.uniform(0,1) >= 0.5:
                pop[0] *= -1
        else:
            pop[gene] = int(10*np.random.uniform(0,1))
    return pop

#Function to perform recombination and mutation
def cross_mut(pop_size, parent_list, domain, pc, pm):
    # single function for crossover and mutation
    # one-point crossover when encoding as array for base-10 genes
    while len(Pop.pop_list) != pop_size:
        child1 = None
        child2 = None
        a = parent_list[np.random.randint(0,len(parent_list))].value
        b = parent_list[np.random.randint(0,len(parent_list))].value
        ##one-point crossover
        pc_randvar = np.random.random()
```

```
          crossover_point = np.random.randint(0,len(a)+2) #crossover point
is chosen randomly for every parent pair selection
        if pc_randvar >= pc and a != b:
            child1 = a[:crossover_point] + b[crossover_point:]
            child2 = b[:crossover_point] + a[crossover_point:]
        else:
            child1 = a[:]
            child2 = b[:]
        if np.random.random() >= pm:
            child1 = mutation(child1)
        if np.random.random() >= pm:
            child2 = mutation(child2)
        #Removing population outside domain
                  if  not  (domain[0]<=gene_decode(child1)<domain[1]  and
domain[0]<=gene_decode(child2)<domain[1]):
            continue
        Pop(child1)
        Pop(child2)

# GA function
def run_ga(initial_pop,domain,pc,pm,generations,parent_size):
    pop_size = len(initial_pop)
    for i in range(generations):
        #ranking and selection
        pareto_selection(parent_size,True)
        #cross and mut
        cross_mut(pop_size, Pop.pop_list, domain, pc, pm)

if __name__ == "__main__":
    pop_size = 100
    parent_size = 30
    domain = [-5,5]
    pc = .7
    pm = .001
    generations = 100
    chromosome_size = 7
    gen_pop(pop_size,domain,chromosome_size)
    run_ga(Pop.pop_list,domain,pc,pm,generations,parent_size)
    nd_list = Pop.pop_list[:]
    for i in Pop.pop_list:
```

```
        for j in Pop.pop_list:
            if i.id == j.id:
                continue
                                if   calc_fitness1(gene_decode(i.value))   >
calc_fitness1(gene_decode(j.value))                                    and
calc_fitness2(gene_decode(i.value)) > calc_fitness2(gene_decode(j.value)):
                try:
                    nd_list.remove(i)
                except:
                    continue
    decoded_pop = [gene_decode(x.value) for x in nd_list]
       plt.scatter(decoded_pop,[calc_fitness1(x)  for  x  in  decoded_pop],
label='NDS on f1')
       plt.scatter(decoded_pop,[calc_fitness2(x)  for  x  in  decoded_pop],
label='NDS on f2')
    plt.title('Non-dominating Solutions')
    plt.xlabel('Genotype space')
    plt.ylabel('Phenotype space')
    #plt.show()
    genotype_space = np.arange(domain[0],domain[1],0.001)
                plt.plot(genotype_space,[calc_fitness1(x)    for    x    in
genotype_space],label='Objective f1')
                plt.plot(genotype_space,[calc_fitness2(x)    for    x    in
genotype_space],label='Objective f2')
    plt.legend(loc='upper center')
    plt.show()
      plt.plot([x**2  for  x  in  np.arange(-1,4,0.001)],[(x-3)**2  for  x  in
np.arange(-1,4,0.001)],label='Parametric plot')
        plt.scatter([x**2  for  x  in  decoded_pop],[(x-3)**2  for  x  in
decoded_pop], label='NDS on Parametric plot')
    plt.title('Parametric plot of the objective functions with NDS')
    plt.xlabel('Obj f1(x) = x^2')
    plt.ylabel('Obj f2(x) = (x-3)^2')
    plt.legend()
    plt.show()
```