Mini-Project 4

Appendix

1. **Introduction**
2. **Methods**
3. **Results and Discussion**
4. **Source Code**
5. **References**

Introduction

Finding all the possible optima is beneficial in physical systems where attaining the global optima is infeasible or it is advantageous to know other alternatives. Niching techniques facilitate canonical GA to perform multi-modal optimization. The niching capability of restricted tournament selection is demonstrated by comparing results of multi-modal optimization using a canonical GA with and without RTS.

Methods

The objective function is $f1(x) = x\sin(10\pi x)+1$. It can be inferred that the linearly increasing amplitude of the sinusoidal graph has many local optima in the chosen domain [-0.5,1]. Canonical GA is run for the first time without using RTS. Consequently, it is run for the second time with RTS which is performed after the crossover and mutation step. To perform RTS, each individual in the child population is compared with all the individuals from the population of the previous generation to find the individual that has the most resemblance. Fitness is compared for each individual and its most closely resembling counterpart from the previous generation, and the child is added to the population only if it has higher fitness than its counterpart. Same hyperparameters are used for running the canonical GA with and without RTS. They are as follows.

- Population size is 50 and each individual is encoded as a base-10 allele with chromosome size equal to 8.
- Tournament selection is done to obtain a parent population. The tournament size is 25 and the parent population size is 6.
- Crossover and mutation probability is 0.4.
- The GA is run for 10,000 generations.

Results

From the following plots, it can be inferred that the individuals are converging only to the global maximum when RTS is not used. In contrast, when RTS is used, we can see many individuals converging to all the local maxima.

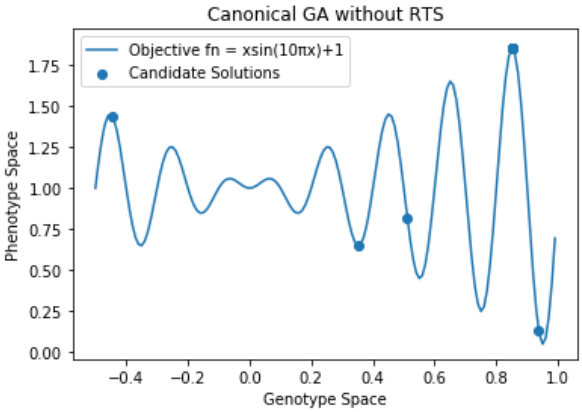

Fig. 1

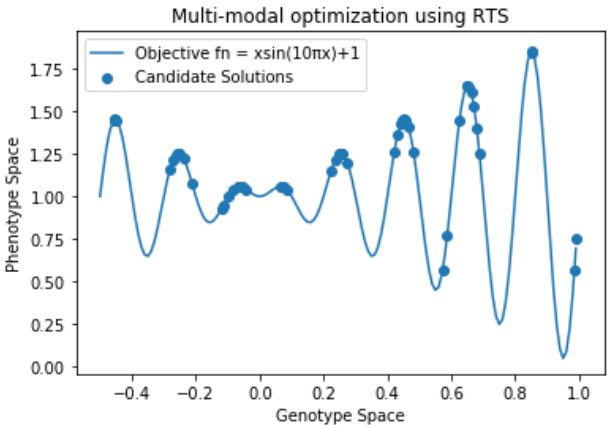

Fig. 2

Discussion

It is not very clear from the plot in fig. 1 that the population converged to the global maximum, but 45 out of 50 individuals are present at the global maximum implying convergence whereas in fig. 2 the population converges to all the local maxima. From plots in fig. 1 and fig. 2, the niching capability of RTS is evident, which is advantageous for multi-modal optimization.

Source Code

Mini-project 1 source code has been modified and used as a benchmark to assess niching capability of RTS. It is included below.

```
import numpy as np
import math
import matplotlib.pyplot as plt
#Function to calculate fitness = xsin(10πx)+1
def calc_fitness(x):
    return x*math.sin(10*math.pi*x)+1
#Function for performing base-10 encoding
def gene_encode(x, chromosome_size):
    charlist = [char for char in str(round(x,chromosome_size-2))]
    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

#Function to generate uniform population distribution across the domain
def gen_pop(pop_size,domain,chromosome_size):
    pop_list = []
    for i in range(pop_size):
        pop_list.append(gene_encode(np.random.uniform(domain[0],
domain[1]),chromosome_size))
    return pop_list

#Function to select parents for recombination
# def roulette_selection(pop_list,parent_size): #population size is equal
to number of parents selected and number of offspring
#     fitness_list = []
#     selection_list = []
#     fitness_sum = 0
#     # roulette_dict = {}
#     # roulette_sum = 0
#     for pop in pop_list:
#         pop_fitness = calc_fitness(gene_decode(pop))
#         fitness_sum += pop_fitness
#         fitness_list.append(pop_fitness)
#     for i in range(len(fitness_list)):
#         fitness_list[i] /= fitness_sum
#     while len(selection_list) != parent_size:
#         for i in range(len(pop_list)):
```

```
#             if np.random.uniform() <= fitness_list[i]:
#                 selection_list.append(pop_list[i])
#     #     prev_roulette_sum = roulette_sum
#     #     roulette_sum += fitness_list[i]
#     #     roulette_dict[(prev_roulette_sum,roulette_sum)] = pop_list[i]
#     # for i in range(parent_size):
#     #     randvar = np.random.uniform(0, 1)
#     #     for j in roulette_dict.keys():
#     #         if float(list(j)[0]) <= randvar <= float(list(j)[1]):
#     #             selection_list.append(roulette_dict[j])
#     #             break
#     return selection_list

def ts(pop_list,parent_size,ksize):
    parent_list = []
    while len(parent_list) <= parent_size:
        kpop = {}
        while len(kpop) <= ksize:
            # print('while 2')
            # print(len(kpop))
            # print(ksize)
            randint = np.random.randint(0,len(pop_list))
            key = calc_fitness(gene_decode(pop_list[randint]))
            if not key in kpop:
                kpop[key] = pop_list[randint]
            else:
                kpop[key+np.random.uniform(0,0.0000000001)] =
pop_list[randint]
        parent_list.append(kpop[max(kpop.keys())])
    return parent_list

# print('Normal pop')
# pop_list = gen_pop(50,[-2,2], 7)
# for i in pop_list:
#     print('Pop = ',i,' :: Fitness = ',calc_fitness(gene_decode(i)))
# parent_list = ts(pop_list,10,48)
# print('parent pop')
# for i in parent_list:
#     print('Pop = ',i,' :: Fitness = ',calc_fitness(gene_decode(i)))

#Function for mutating base-10 genes
#Retains population outside domain
```

```
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
    childpop_list = parent_list[:]
    while len(childpop_list) <= pop_size:
        child1 = None
        child2 = None
        a = parent_list[np.random.randint(0,len(parent_list))]
        b = parent_list[np.random.randint(0,len(parent_list))]
        ##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:
            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)
        if not (domain[0]<=gene_decode(child1)<domain[1] and
domain[0]<=gene_decode(child2)<domain[1]):
            continue
        childpop_list.append(child1)
        childpop_list.append(child2)
    return childpop_list


# GA function
```

```
def run_ga(pop_size,domain,pc,pm,chromosome_size,
generations,parent_size,ksize):
    initial_pop = gen_pop(pop_size, domain,chromosome_size)
    print(len([gene_decode(x) for x in initial_pop if
0.8<=gene_decode(x)<=.9]))
    plt.plot([x for x in
np.arange(domain[0],domain[1],0.01)],[calc_fitness(x) for x in
np.arange(domain[0],domain[1],0.01)],label='Objective fn = xsin(10πx)+1')
    plt.scatter([gene_decode(x) for x in
initial_pop],[[calc_fitness(gene_decode(x)) for x in
initial_pop]],label='initial_pop')
    plt.legend()
    plt.show()
    pop_list = initial_pop[:]
    # count = 0
    for i in range(generations):
        # count += 1
        # print(count)
        #selection
        selected_pop = ts(pop_list,parent_size,ksize)
        #cross and mut
        crossmut_pop = cross_mut(pop_size, selected_pop, domain, pc, pm)
        pop_list = crossmut_pop[:]
    return pop_list


if __name__ == "__main__":
    print('miniproject 1')
    #population size needs to be even
    population_size = 50
    ksize = int(population_size/2)
    parent_size = 6
    crossover_pc = 0.4
    mutation_pm = .4
    chromosome_size = 8
    domain = [-.5,1]
    generations = 10000
    final_poplist =
run_ga(population_size,domain,crossover_pc,mutation_pm,chromosome_size,gen
erations,parent_size,ksize)
    print(len([gene_decode(x) for x in final_poplist if
0.8<=gene_decode(x)<=.9]))
```

```
    plt.plot([x for x in
np.arange(domain[0],domain[1],0.01)],[calc_fitness(x) for x in
np.arange(domain[0],domain[1],0.01)],label='Objective fn = xsin(10πx)+1')
    plt.scatter([gene_decode(x) for x in
final_poplist],[[calc_fitness(gene_decode(x)) for x in
final_poplist]],label='Candidate Solutions')
    plt.title('Canonical GA without RTS')
    plt.xlabel('Genotype Space')
    plt.ylabel('Phenotype Space')
    plt.legend()
    plt.show()
```

Source Code for mini-project 4 is included below.

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python
"""
Title: Effect of RTS on Multi-modal Optimization
Version: 1.0
Created on Sat Mar 15 2022
@author: Mahesh
"""
import numpy as np
import math
import matplotlib.pyplot as plt

#Function to calculate fitness = xsin(10πx)+1
#def calc_fitness(x):
#    return x * math.sin(10 * math.pi * x) + 1

def calc_fitness(x):
    return x*math.sin(10*math.pi*x)+1

#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.decoded_value = gene_decode(self.value)
        self.fitness = calc_fitness(self.decoded_value)
        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 ts(pop_list,parent_size,ksize):
```

```
    parent_list = []
    while len(parent_list) <= parent_size:
        # print('while 1')
        kpop = {}
        while len(kpop) <= ksize:
            # print('while 2')
            # print(len(kpop))
            # print(ksize)
            randint = np.random.randint(0,len(pop_list))
            key = pop_list[randint].fitness
            if not key in kpop:
                kpop[key] = pop_list[randint]
            else:
                kpop[key+np.random.uniform(0,0.0000000001)] =
pop_list[randint]
        parent_list.append(kpop[max(kpop.keys())])
    return parent_list

#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

def rts_selection(pop):
    distance = 99
    similar_pop = None
    decoded_pop = gene_decode(pop)
    for i in Pop.pop_list:
        gap = abs(decoded_pop - i.decoded_value)
        if gap <= distance:
            similar_pop = i
            distance = gap
    if similar_pop.fitness < calc_fitness(decoded_pop):
        del Pop.pop_list[Pop.pop_list.index(similar_pop)]
        Pop(pop)

# GA function
```

```
def run_ga(initial_pop,domain,pc,pm,generations,parent_size):
    for i in range(generations):
        parent_list = ts(initial_pop,parent_size,int(parent_size/2))
        a = parent_list[np.random.randint(0,len(parent_list))].value
        b = parent_list[np.random.randint(0,len(parent_list))].value
        while a == b:
            b = parent_list[np.random.randint(0,len(parent_list))].value
        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:
            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)
        if not (domain[0]<=gene_decode(child1)<domain[1] and
domain[0]<=gene_decode(child2)<domain[1]):
            i -= 1
            continue

        rts_selection(child1)
        rts_selection(child2)

if __name__ == "__main__":
    print('miniproject4')
    pop_size = 50
    parent_size = 6
    domain = [-.5,1]
    pc = 0.4
    pm = .4
    generations = 500*20
    chromosome_size = 8
    gen_pop(pop_size,domain,chromosome_size)
    run_ga(Pop.pop_list,domain,pc,pm,generations,parent_size)
    plt.plot([i for i in
np.arange(domain[0],domain[1],0.01)],[calc_fitness(i) for i in
np.arange(domain[0],domain[1],0.01)],label='Objective fn = xsin(10πx)+1')
```

```
    plt.scatter([i.decoded_value for i in Pop.pop_list],[i.fitness for i
in Pop.pop_list],label='Candidate Solutions')
    plt.title('Multi-modal optimization using RTS')
    plt.xlabel('Genotype Space')
    plt.ylabel('Phenotype Space')
    plt.legend()
    plt.show()
```

References

- Singh, G., & Deb, K. (2006, July). Comparison of multi-modal optimization algorithms based on evolutionary algorithms. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation* (pp. 1305-1312).