Methods and Results of Using an Island model and a Genetic Algrithom to solve the 6 standard test functions

Robert Goes

***Abstract*— Analysis of two different genetic algorithms being used to solve the 8-Queens problem, one using permutation to represent the chess board, and the other using a set. The algorithm using the permutation converged on a correct solution the fastest.**

I. METHODS

A. Island Model

I use a 5 island representation, with each island being a population of 200 individuals with 20 additional migrant individuals. Migrant individuals were determined by selecting randomly, and copying over to the new population. Migration occurred from $1 \rightarrow 2 \rightarrow ... \rightarrow 5 \rightarrow 1$. The migration interval used was 20, and the algorithm was run for 1000 generations.

B. Genetic Algorithm (GeneticAlg)

A special option was created, so when the migration interval was -1 no migration would occur. The island model was run with 1 island and a interval of -1 to make it the equivalent genetic algorithm.

C. Selection

5 individuals are randomly selected from the population. The two most fit individuals are selected as parents.

D. Crossover

Single point crossover was used. A crossover point was selected from (0,29), non-inclusive, from a uniform random distribution. Child A would get the portion of parent A, not including the crossover point, and the rest from Parent B. Child B likewise.

E. Mutation

For each gene in a individual, a random value between 0 and 1 is calculated. If this value is less or equal to 0.2, that gene will be mutated. The amount the gene is mutated is a number from the Gaussian distribution with a mean 0 and a standard deviation of 0.09*abs(upper bound of gene - lower bound). If the value of the gene is above the upper bound, the same absolute value will be subtracted until it is within the bounds. The same is the case if it is below the lower bound.

F. Elitism

The 4 most fit individuals from the population are directly copied over to the next generation.

G. Fitness Sharing(sharing)

Inside the fitness function, for a individual with a fitness k, the number of individuals within the range (k-10,k+10) non-inclusive are counted. A fitness penalty of 0.2*count is added to the individuals fitness.

H. Crowding(dist)

When a new individual is created, after mutation, it's euclidean distance to all the individuals in the new generation is calculated. If this value is less then 5, it is not added to the new population.

II. RESULTS

The results graphed below are for the values given above. One trial was conducted for each function with each of the listed features turned on only when listed. The two unimodel functions, rosenbrock and sphere did not appear to share any significant similarities as to which option was being used as to speed of convergence.In every function, crowding(dist) appeared to have a significant affect on the rate of convergence for the average. This effect wasn't consistent for the best fitness. Because of elitism, the best fitness always improved, except in the case of the algorithms using sharing, because when more individuals would become close to the best, it's fitness would increase in value. Most functions rapidly approached a fitness low fitness value within 200 generations, then kept steady, except in the case of the schwefel function, where it improved it's fitness, but did not get anywhere close to 0. In the rosenbrock function, crowding(dist) performed worse the any other method.

III. ADDITIONAL OBSERVATIONS

While determining what mutation rate, larger changes in genes when mutating produced a much slower convergence. Also, a scaled sharing method was attempted, where the distance where the penality was assesed, and by how much, was changed with average fitness of the population. This created extreme varation in the best fitness of the algorithms using sharing, so it was removed. Having 4 elites, rather then 2, also seemed to increase the rate the best converged to a low value, but most likely reduced the diversity in the population, leading to the algorithms never converging to zero.

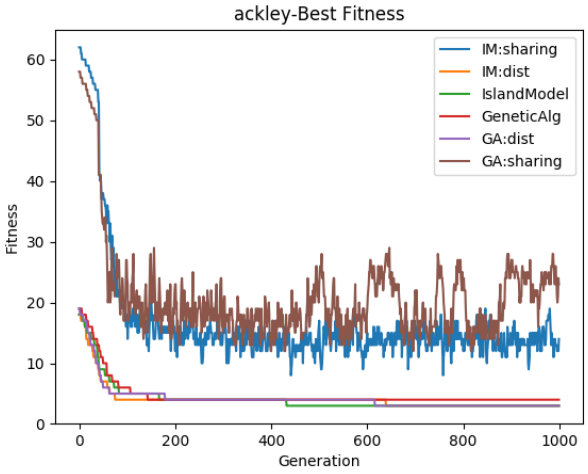

Fig. 1. Fitness of best individual in entire generation, In the case of the island model, it is the single best individual of all the islands.

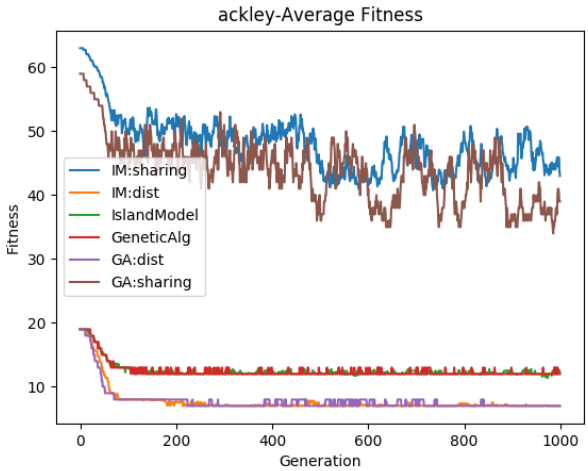

Fig. 2. Average Fitness of all individual's in entire generation.

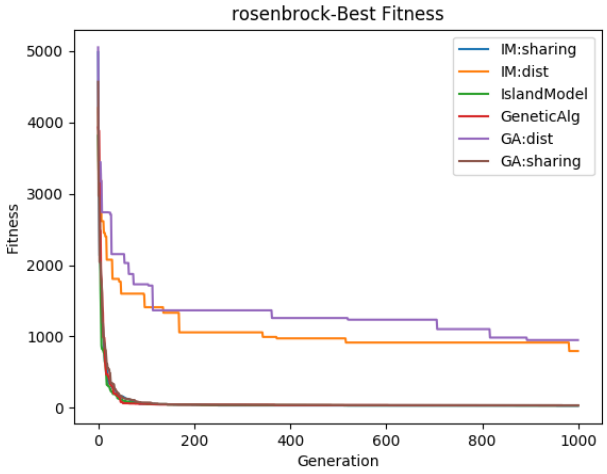

Fig. 3. Fitness of best individual in entire generation, In the case of the island model, it is the single best individual of all the islands.

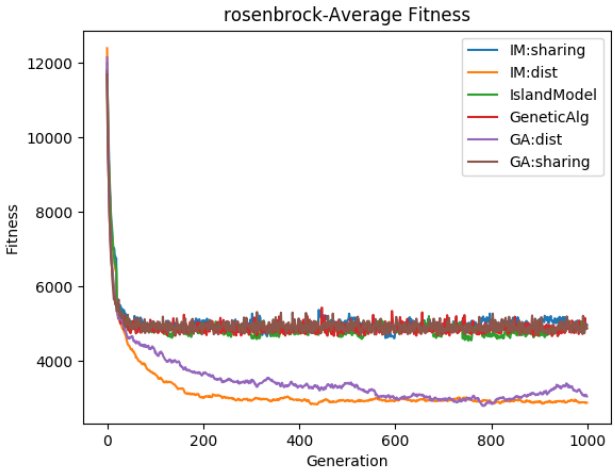

Fig. 4. Average Fitness of all individual's in entire generation.

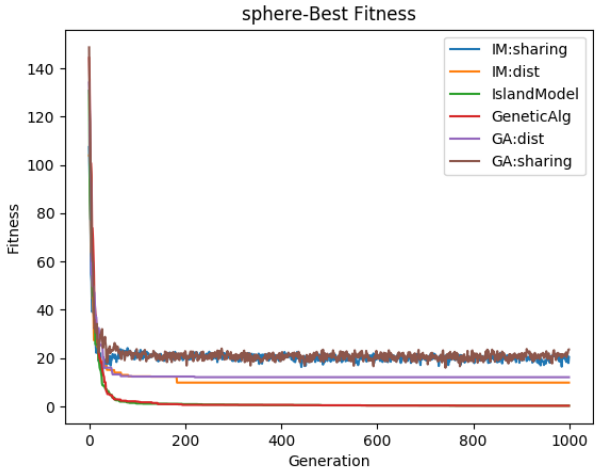

Fig. 5. Fitness of best individual in entire generation, In the case of the island model, it is the single best individual of all the islands.

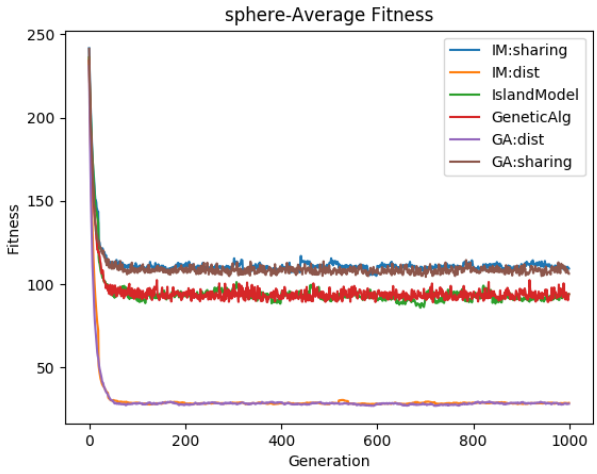

Fig. 6. Average Fitness of all individual's in entire generation.

1.png

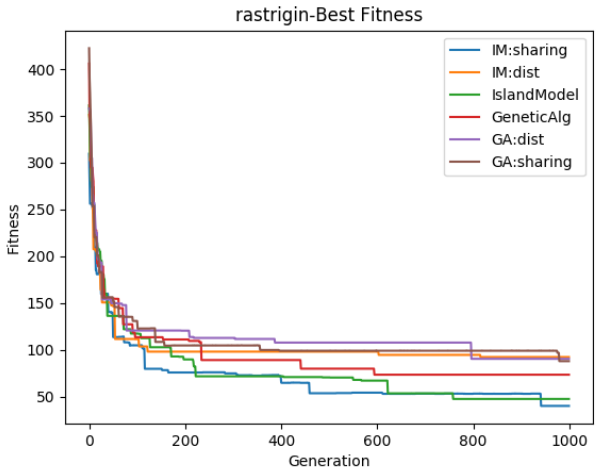

Fig. 7. Fitness of best individual in entire generation, In the case of the island model, it is the single best individual of all the islands.

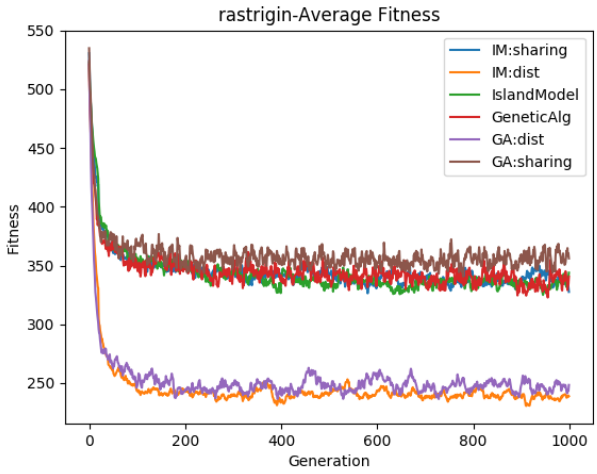

Fig. 8. Average Fitness of all individual's in entire generation.

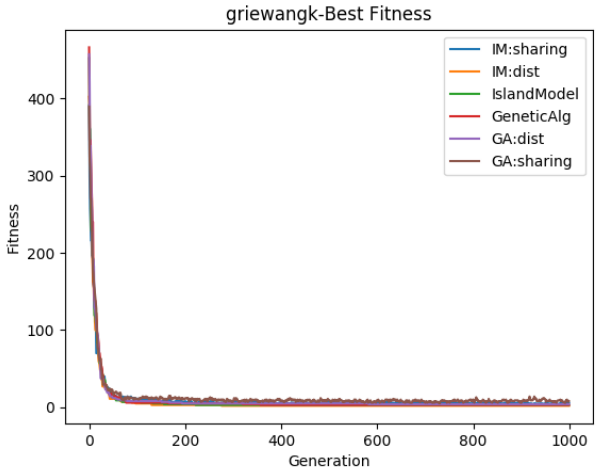

Fig. 9. Fitness of best individual in entire generation, In the case of the island model, it is the single best individual of all the islands.

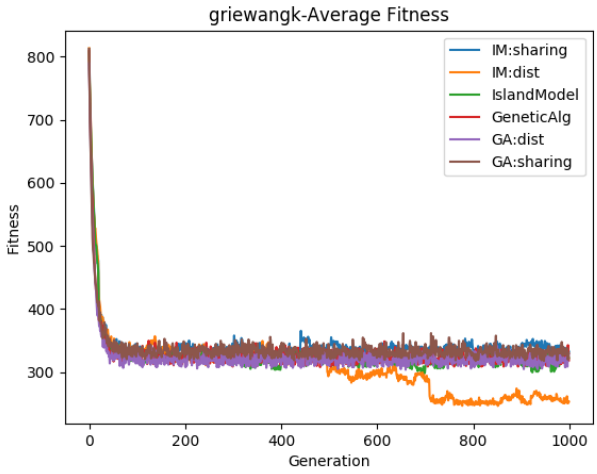

Fig. 10. Average Fitness of all individual's in entire generation.

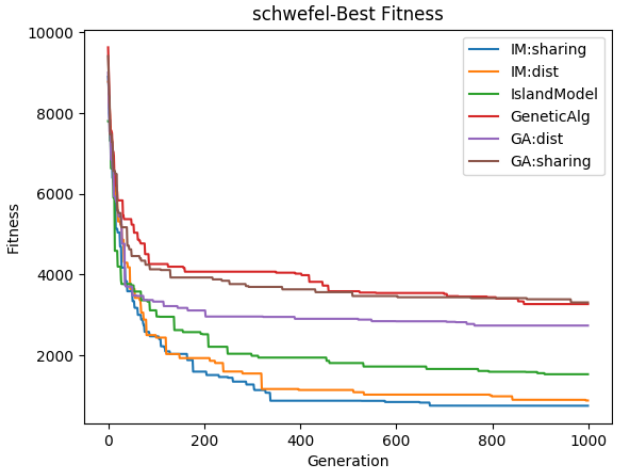

Fig. 11. Fitness of best individual in entire generation, In the case of the island model, it is the single best individual of all the islands.

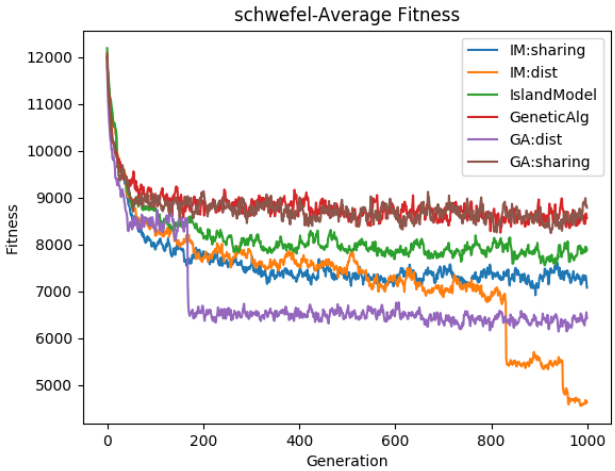

Fig. 12. Average Fitness of all individual's in entire generation.

1 Source Code

```
import matplotlib.pyplot as plt
import numpy as np
import random
from multiprocessing import Pool
debug_mig = False
debug_mutate = False
debug_select = False
debug_elite = False

#global storage for plot labeling
plot_num = 0

#min dist
min_dist = 5
#fitness share range (proximety to other fitness +- the value)
share_range = 10
share_factor = 0.2
#number of elites to copy over
elites_size = 4
#size of individuals for tournment
tournment_size = 5
#probability of a signle gene being mutated
mutation_rate = 0.2
#distrabution of random numbers to assign
mutation_scale = 0.09
#class represents a full island model ga

def make_chart(best_fits, labels, generations, title):
    axis = np.arange(generations)
    count = 0
    for line in best_fits:
        plt.plot(axis, line, label = labels[count])
        count += 1
    plt.xlabel("Generation")
    plt.ylabel("Fitness")

    plt.title(title)
    plt.legend()
    plt.savefig(title)
   # Clear the current axes.
    plt.cla()
    # Clear the current figure.
    plt.clf()
    # Closes all the figure windows.
```

```
    plt.close('all')



class GA:
    def __init__(self, flags, fitness, bounds, precision, pop_size, num_islands, mig_size):
        #dict containing flags(for use in ga project2)
        self.flags = flags
        #fitness function to be used
        self.input_fitness = fitness
        #bounds of random values to create in tuple
        self.bounds = bounds
        #precision
        self.precision = precision
        #size of each poplulation(without migration)
        self.pop_size = pop_size
        #number of islands to create
        self.num_islands = num_islands

        #size of each migration
        self.mig_size = mig_size

        self.islands = [Poplulation(flags, fitness,bounds,precision,pop_size,mig_size) for i

    def run(self, generations, mig_interval):
        #make sure our current best is allways better then the value
        best_fit = np.zeros(generations)
        best_fit = best_fit + 10000000

        avg_fit = np.zeros(generations)
        worst_fit = np.zeros(generations)

        current = 0
        while current < generations:
            num = 0
            #if migration is negative, then we are running without islands.
            if current % mig_interval == 0 and mig_interval >= 0:
                for i in range(0,self.num_islands):
                    if debug_mig:
                        print("Migrating from {} to {}.".format(i,(i+1)%self.num_islands))
                    self.islands[i].migrate(self.islands[(i+1)%self.num_islands])
            for island in self.islands:
                island.make_new_gen()

                #reccord the data for plotting
                if island.best_fit() < best_fit[current]:
```

```
                    best_fit[current] = island.best_fit()
                if avg_fit[current] == 0:
                    avg_fit[current] = island.avg_fit()
                else:
                    avg_fit[current] = (island.avg_fit()+avg_fit[current])/2
                if island.worst_fit() > worst_fit[current]:
                    worst_fit[current] = island.worst_fit()

            current += 1
        return (best_fit,avg_fit, worst_fit)




class Poplulation:
    def __init__(self, flags, fitness, bounds, precision, pop_size, mig_size):
        self.flags = flags
        #fitness function to be used
        self.input_fitness = fitness

        #bounds of random values to create in tuple
        self.bounds = bounds
        #precision
        self.precision = precision
        #size of each poplulation(without migration)
        self.pop_size = pop_size
        #size of each migration
        self.mig_size = mig_size

        #initialize the population
        self.curr = np.random.rand(31*(pop_size+mig_size))
        self.curr = self.curr.reshape(pop_size+mig_size, 31)
        self.curr = abs(bounds[0]-bounds[1])*self.curr +bounds[0]
        self.curr = np.round(self.curr,decimals=self.precision)
        self.calc_fitness()
        self.curr = np.round(self.curr,decimals=self.precision)


    def calc_fitness(self):
        #make a new function wrap supplyed fitness function
        def apply_fit(a):
            a[30] = self.input_fitness(np.copy(a[0:30]))
            #print("a[30] was {} and now is {}".format(prev, a[30]))
            return a
        np.apply_along_axis(apply_fit, 1, self.curr)
        if self.flags.get("sharing",False):
```

```
        fit_vals = np.copy(self.curr[:,30])
        def apply_sharing(a):
            lower = a[30] - share_range
            upper = a[30] + share_range
            count = np.sum((fit_vals > lower )& (fit_vals < upper))
            #remove 1 because it will allways count itself
            count = share_factor*(count -1)
            #print("Calculated count of {} for {}".format(count,a[30]))
            a[30] = a[30] + count
            return a
        np.apply_along_axis(apply_sharing, 1, self.curr)



#migrates from a to b
def migrate(self, b):
    #actually uses destination migration size, so populations could have diffrent numbe
    mig_index = np.random.choice(self.pop_size,b.mig_size)
    if debug_mig:
        print("Index's of population members to migrate:")
        print(mig_index)
        print("Migrating invidiuals:")
        print(self.curr[mig_index])
    b.curr[b.pop_size:] = self.curr[mig_index]

def select(self):
    select_index = np.random.choice(self.pop_size+self.mig_size,tournment_size)
    parents_fit = self.curr[select_index,30]
    parents = parents_fit.argsort()
    if debug_select:
        print("Parents, parents_fit, and select indexs-----")
        print(parents)
        print(parents_fit)
        print(select_index)
        print("-------------------")
    #return the two best parents out of the choice
    return select_index[parents[0]], select_index[parents[1]]

def make_new_gen(self):
    ##select half the number of childern needed as parents, and also index into mig size

    #make a new array representing the next genration
    self.new = np.copy(self.curr)

    #current members added
    current = 0
```

```
        target = self.pop_size

        #perform elitism
        all_fit = self.curr[0:self.pop_size,30]
        elite_index = all_fit.argsort()[0:elites_size]
        self.new[0:elites_size] = self.curr[elite_index]
        current += elites_size
        if debug_elite:
            print(self.curr[elite_index,30])
            print(self.new[:10,30])
            print("best gene")
            print(self.curr[elite_index[0]])

        #fill up the population
        while current < target:
            a_index,b_index = self.select()
            current += 1
            child_a , child_b = self.crossover(self.curr[a_index][:],self.curr[b_index][:])
            self.mutate(child_a)
            self.mutate(child_b)
            if self.flags.get("dist",False):
                if current< target and self.distance(child_a,self.new,current) >= min_dist:
                    self.new[current] = child_a
                    current = current+1
                if current < target and self.distance(child_b,self.new,current) >= min_dist:
                    self.new[current] = child_b
                    current = current+1
            else:
                if current< target:
                    self.new[current] = child_a
                    current = current+1
                if current < target:
                    self.new[current] = child_b
                    current = current+1




        self.curr[:] = self.new
        self.curr = np.round(self.curr,decimals=self.precision)
        self.calc_fitness()
        self.curr = np.round(self.curr,decimals=self.precision)

    def crossover(self, a , b):
        #crossover points
        #1-29 since don't want to randomly copy parents
```

```
            pa = random.randint(1,29)
            child_a = np.copy(a)
            child_a[pa:] = b[pa:]
            child_b = np.copy(b)
            child_b[pa:] = a[pa:]
            if debug_mutate:
                print("Child a: {}".format(a))
                print("Child b: {}".format(b))
            return (child_a,child_b)

        def mutate(self, a):
            #calc all 30 random numbers to add first

            diff = abs(self.bounds[1]-self.bounds[0])/2
            randoms = np.random.normal(0, mutation_scale*diff,30)
            for i in range(0,30):
                if random.uniform(0, 1) <= mutation_rate:
                    if debug_mutate:
                        print("Mutation occured!")
                    a[i] += randoms[i]
                    while a[i] > self.bounds[1]:
                        a[i] = a[i] - abs(self.bounds[1])
                    while a[i] < self.bounds[0]:
                        a[i] = a[i] + abs(self.bounds[0])
                    if debug_mutate:
                        print("----new a[{}] = {} and mutated by {}".format(i,a[i],randoms[i]))

        def distance(self, a, pop,current):
            def dist(b):
                return np.sqrt(np.sum((a[0:30]-b)**2))
            return np.amin(np.apply_along_axis(dist, 1, np.copy(pop[:current,0:30])))
        def best_fit(self):
            fits = self.curr[:,30]
            return round(min(fits),self.precision)
        def avg_fit(self):
            fits = self.curr[:,30]
            return round(np.average(fits),self.precision)
        def worst_fit(self):
            fits = self.curr[:,30]
            return round(max(fits),self.precision)




def test_fit(a):
```

```
    return abs(a.sum())

#use bounds (-5.12,5.12) precision 2, 0's at all 0
def sphere(a):
    return np.sum((a**2))

#use bounds (-2.038,2.048) precision 3, 0's at all 1
def rosenbrock(a):
    a_1 = np.copy(a[0:-1])
    a_2 =  np.copy(a[1:])
    return np.sum(100*(a_2-(a_1**2))**2+(a_1-1)**2)

#use bounds (-5.12,5.12) precision 2 all 0's solution
def rastrigin(a):
    return 300+np.sum(a**2-10*np.cos(2*np.pi*a))

#use bounds (-512.03,511.97) with precision 4 and x all -420.9687 is 0
def schwefel(a):
    return 418.9829*30+np.sum(a*np.sin(np.sqrt(np.absolute(a))))

#use bounds (-30,30) with precision 0?? 0's = 0
def ackley(a):
    return 20+np.e-20*np.exp(-0.2*np.sqrt(np.sum(a**2)/30))-np.exp(np.sum(np.cos(2*np.pi*a)/

#use bounds(-600,600) with precision 0 and all 0's = 0
def griewangk(a):
    b = np.arange(1,31)
    return 1 + np.sum(a**2)/4000-np.prod(np.cos(a/np.sqrt(b)))


"""
#test sphere
test_sph = np.zeros(30)
print(sphere(test_sph))

#test rosenbrock
test_rose = np.zeros(30)+1
print(rosenbrock(test_rose))

#test rastrigin
test_rast = np.zeros(30)
print(rastrigin(test_rast))

#test schwefel
test_schwe = np.zeros(30) -420.9687
print(schwefel(test_schwe))
```

```
#result is 0.0003818351233348949 and the function is correct, rounding errors?


#test ackley
test_ackl = np.zeros(30)
print(ackley(test_ackl))
#result is effectivly 0.... in given precision it is.

#test griewangk
test_grie = np.zeros(30)
print(griewangk(test_grie))



"""
#(self, flags, fitness, bounds, precision, pop_size, num_islands, mig_size)
#test GA  with 2 islands, pop size of 2, migrant size of 2
function_list = [(sphere, (-5.12,5.12), 2),
                 (rosenbrock,(-2.038,2.048),3),
                 (rastrigin,(-5.12,5.12),2),
                 (schwefel, (-512.03,511.97), 4),
                 (ackley,(-30,30),0),
                 (griewangk,(-600,600),0)]
num_generations = 1000
def proccess_function(function):
    print("+{} Starting".format(function[0].__name__))
    test_1 = GA({"sharing":True,"dist":False},function[0],function[1],function[2],200,5,20)
    test_2 = GA({"sharing":False,"dist":True},function[0],function[1],function[2],200,5,20)
    test_3 = GA({"sharing":False,"dist":False},function[0],function[1],function[2],200,5,20)
    test_4 = GA({"sharing":False,"dist":False},function[0],function[1],function[2],200,1,0)
    test_5 = GA({"sharing":False,"dist":True},function[0],function[1],function[2],200,1,0)
    test_6 = GA({"sharing":True,"dist":False},function[0],function[1],function[2],200,1,0)
    lines = []
    lines_best = []
    lines_avg = []
    lines_worst = []
    lines.append(test_1.run(num_generations,20))
    lines.append(test_2.run(num_generations,20))
    lines.append(test_3.run(num_generations,20))
    lines.append(test_4.run(num_generations,-1))
    lines.append(test_5.run(num_generations,-1))
    lines.append(test_6.run(num_generations,-1))
    lines_best = [part[0] for part in lines]
    lines_avg = [part[1] for part in lines]
    lines_worst = [part[2] for part in lines]
    labels = ["IM:sharing","IM:dist","IslandModel","GeneticAlg","GA:dist","GA:sharing"]
```

```
    make_chart(lines_best,labels,num_generations,"{}-Best Fitness".format(function[0].__name
    make_chart(lines_avg,labels,num_generations,"{}-Average Fitness".format(function[0].__na
    make_chart(lines_worst,labels,num_generations,"{}-Worst Fitness".format(function[0].__na
    print("-{} Complete".format(function[0].__name__))
    return 1


with Pool(6) as p:
    print(p.map(proccess_function,function_list))
```

2 Output

```
python3 ga.py
+sphere Starting
+rosenbrock Starting
+rastrigin Starting
+schwefel Starting
+ackley Starting
+griewangk Starting
-schwefel Complete
-griewangk Complete
-ackley Complete
-rastrigin Complete
-sphere Complete
-rosenbrock Complete
[1, 1, 1, 1, 1, 1]
```