Table of Contents

Item 1 - Population Based Search

Population is generated assuring that there is distance difference $x > 6$ between each other chromosome. (See Code Snippet 20, Statistics 4) This allows quicker convergence.
I have implemented GA's reproduction by retaining 40% of highest-fitness population as potential parents and choosing each other chromosome with 30% probability (See Code Snippet 2). Increasing both numbers would increase exploitation, but at the start of search we want exploration, hence we give priority to new, mutated chromosomes. Keeping each other solution with a 30% probability allows us to balance exploitation with exploration by giving lower fitness solutions a chance to make it to the next generation.
We produce 2 children from 2 parents by using Roulette Wheel parent selection method. After evaluation of different parent selection techniques. (See Statistics 1)
Parent selection is important as it prevents one fit solution from taking over the whole population, although we want to avoid premature convergence, hence we favour exploration with healthy level of exploitation and that's what Roulette Wheel selection offers to us.
Considering crossover methods (See Statistics 2), they have performed similarly, although uniform crossover with probability $p = 0.6$ (See Code Snippet 3) showed best average validation accuracy.
We put emphasis on mutation as it is the main bit that allows us to escape local minima. I have tried several mutation techniques (See Statistics 3), although highly stochastic techniques such as swap (See Code Snippet 4) or random mutation (See Code Snippet 5) provide bad results as they lack a systematic approach. We want to move towards global optima, hence approaches like offset mutation (See Code Snippet 6) or distance mutation (See Code Snippet 7) perform better. Distance mutation incorporates differential techniques, stochasticity by finding a difference between chromosomes i and j then adding that difference to another chromosome k, generating random offset r and adding it on chromosome i whilst subtracting it from chromosome j.
If ANN fails to converge to global optima after 10 iterations, we boost exploration by increasing the lower and upper bound of a random offset that we add/subtract to individual genes. (See Code Snippet 1) Although, most of the times ANN reaches accuracy $x > 70\%$ after 5 iterations of GA.
We add an extra layer of PSO (See Code Snippet 8) that acts as a "safety net" to ensure that our network will converge. Adding PSO allows us to avoid situation where GA is stuck in the wrong path, as PSO takes completely different approach to mutation of individual chromosomes. We stop executing PSO if it has not improved after 5 iterations and validation accuracy is over 75%
Although, it is a heuristic approach, hence we cannot guarantee that our ANN will converge to global optima 100% of the time.
After 20 generations of GA we start increasing exploitation slowly. (See Code Snippet 18)

Item 2 - Memetic Algorithm

DE requires some sort of local search, as it highly depends on population and is not systematic. Performing local search every generation would be simply too expensive, hence I have decided to perform local search on chromosomes that have been improved by DE. Local search is executed as well with 20% probability if DE have not improved initial best fitness after 18 generations, which allows to improve best solutions. (See Code Snippet 9) I have used Simulated Annealing algorithm.
DE is oriented in chromosomes with distance $x \in [4, 50]$ from current solution, but with SE we want to exploit even closer chromosomes that are in distance range $[0, 4]$.
When generating a neighbour, we check if we have already generated a neighbour with the same distance from the current chromosome, if that is the case, we generate once again. All the generated neighbours are stored in tabu list which prevents us from taking too similar chromosomes. (See Code Snippet 10) Distance is rounded to floating point precision starting from 1 and is increased gradually, as search targets closer and closer solutions.
With cooling we decrease neighbour range which accordingly decreases distance between new and current chromosomes, tailoring search systematically towards narrower search space.
We start by choosing most recently accepted solutions, but if SA have not improved for 3 iterations, we randomly choose one of 3 best fitness individuals that SA have produced (See Code Snippet 11).

Former allows quicker recovery from lower fitness locality. After we have selected one of the Top3 solutions for 10 times, we start selecting either global or local best solutions with 50% probability. (See Code Snippet 16)
When mutating, we subtract the vector if gene is cropped to one (0.99999999999) and we do it 50% of the times otherwise. We add the vector if the gene is cropped down to 0 and we do it 50% of the times otherwise. Such approach prevents certain genes being stuck at boundary values after they get cropped during mutation. To bring more variety into the search, we subtract another smaller mutation vector, which has $x \in [4,6]$ times smaller range, where x is picked randomly. (See Code Snippet 12)

Item 3 - Differential Search

After generations of GA and PSO, our ANN should have converged, therefore, DE has to be more tailored towards exploitation instead of exploration.
We improve regular DE mutation (adding the weighted difference of two chromosomes onto the third) by performing random offset mutation on the resulting chromosome. (See Code Snippet 15) Random offset mutation is performed on a random number of genes within boundaries.
This allows us to be more independent of the current generation and avoid DE barely improving solutions when the generation is dominated by similar chromosomes.
When Canberra distance (See Code Snippet 13) between chromosomes is low, our differential mutation would not make any improvements, although if distance $x > 10$ then we will explore the current locality without wondering off into different locality. To keep distance between chromosomes low, we use random offset mutation bounds within $[-0.05, 0.005]$ and we select mutation factor $m = 0.01$.
DE thrives if the population is diverse, hence we shrink random mutation boundaries every 5 DE generations. Alongside, we reduce upper boundary on the amount of genes that could be mutated. This allows us to explore more diverse solutions while shortening our search horizons to increase the likelihood of finding the local maxima.
The problem with DE is that is highly sensitive to the contents of the population, population can be dominated by lower fitness individuals which will prevent from us from finding the local optima. To counter the issue we refill population with the best accuracy solutions every 5 generations. (See Code Snippet 14) Refilling ratio starts from 50%, but is increased by 10% every 5 generations, which encourages DE to differentiate on higher fitness individuals gradually.
Mutation can go slightly off, hence we reject solutions that are too far away from the best fitness solution, to keep exploiting solutions within distance $x \in [0, 50]$. (See Code Snippet 17)

Item 1 – Performance Measure

Number of search iterations were controlled with a loop. (See Code Snippet 19) CSV file is produced with all the relevant performance metrics with each search instance. (See Statistics 5) Test accuracy is used as performance measure as its tests ANN on a dataset with 10000 unseen examples.

Item 2 – Descriptive Results

Modified algorithm provided consistent results as its 1st and 3rd quartiles are really close to the mean, especially compared to the unmodified version. Mean is way higher than the original, lowest accuracy is a bit above original's mean accuracy. (See Statistics 7) We have achieved maximum test accuracy of 79.18% whereas original one was 75.77%. Overall, almost every trial produced higher results than the original's maximum. Modified version does not have downfall in accuracy and it successfully converges 100% of the times. (See Statistics 6)

Item 3 – Comparison Statistics

Because test data from both algorithms is unpaired, I have used Mann-Whitney U test which allows us to determine if two independent samples were selected from sets having the same distribution. From which, we can derive if modified algorithm changes the solution distribution. It gave z-score of -3.2127 and p-value of 0.00132, since $p < 0.5$ we can say that both algorithms are statistically significantly different. (See Statistics 8) Which implies that randomly picked accuracy from modified algorithm is highly likely to be higher than the one from unmodified.

Code Snippets

```
if i > meta['ga_convergence_iter'] and self.best_val_accuracy < meta['convergence_fitness']:
    self.ga_increased_explr += 1
    meta['random_threshold_lower'] += 0.08
    meta['random_threshold_upper'] += 0.05
```

Code Snippet 1

```
# Keep a fraction of population as parents.
retain_length = int(len(population) * meta['retain'])
parents = population[:retain_length]

# For those that we aren't keeping, randomly keep some anyway.
for individual in population[retain_length:]:
    if meta['random_select'] > random():
        parents.append(individual)
```

Code Snippet 2

```
def uniform_crossover(self, dad, mom, p_crossover):
    """..."""
    child = gen_rand_chromosome(self.num_chrom_param)
    # Uniform crossover - we assign probability to each gene.
    for i in range(len(child)):
        rand = random()
        if rand < p_crossover:
            child[i] = dad[i]
        else:
            child[i] = mom[i]
    return child
```

Code Snippet 3

```python
def swap_mutation(self, chrom, no_of_genes):
    # Assure that we have even amount of genes.
    if no_of_genes % 2 == 1:
        no_of_genes += 1
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, no_of_genes)
    a,b = split_list(genes)

    for i,j in zip(a, b):
        temp = chrom[j]
        chrom[j] = chrom[i]
        chrom[i] = temp
```

Code Snippet 4

```python
def mutate_random(self, chrom, no_of_genes):
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, no_of_genes)
    for gene in genes:
        chrom[gene] = random()
```

Code Snippet 5

```python
def mutate_offset(self, chrom, no_of_genes, offset):
    """..."""
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, no_of_genes)
    for gene in genes:
        chrom[gene] = chrom[gene] + offset
    # Force boundaries on new mutated chromosome.
    np.clip(chrom, 0, 0.99999999999, out=chrom)
```

Code Snippet 6

```python
def mutate_distance(self, chrom, lower_bound, upper_bound):
    i = 0
    j = 1
    while j < len(chrom) - 1:
        difference = abs(chrom[i] - chrom[j])
        chrom[j + 1] += difference
        r_offset = rand.uniform(lower_bound, upper_bound)
        chrom[i] += r_offset
        chrom[j] -= r_offset
        j += 1
        i += 1
    # Force boundaries on a new mutated chromosome.
    np.clip(chrom, 0, 0.99999999999, out=chrom)
```

Code Snippet 7

```
def pso(self, swarm):
    best_fitness = 0
    best_chrom = None
    i = 0
    while i < meta['pso_iterations']:
        if i > meta['pso_iteration_threshold'] and best_fitness > meta['pso_fitness_threshold']:
            if self.best_val_accuracy == self.best_ga_accuracy:
                return
        print("PSO iteration %d best fitness: %d" % (i, best_fitness))
        # Iterate through every particle (solution) in the swarm.
        for j in range(meta['pop_count']):
            swarm[j].update_fitness(self)
            if swarm[j].fitness > best_fitness:
                best_chrom = swarm[j].chrom
                best_fitness = swarm[j].fitness
        # cycle through swarm and update velocities and position
        for j in range(meta['pop_count']):
            swarm[j].update_velocity(best_chrom)
            swarm[j].update_position()
        i += 1

    self.update_best(best_fitness, best_chrom)
```

Code Snippet 8

```
def de_execute_sa(self, chrom, fitness):
    if fitness > self.best_val_accuracy:
        solution = self.sa_search(Solution(chrom, fitness))
        chrom = solution.chrom
        fitness = solution.fitness
    elif self.de_not_improving_count > meta['de_non_improvement_t'] and random() > meta['de_prob_sa']:
        solution = self.sa_search(Solution(chrom, fitness))
        chrom = solution.chrom
        fitness = solution.fitness
    return chrom, fitness
```

Code Snippet 9

```
# Function for creating a list of neighbours
def create_neighbours(self, a_chromosome, num_chrom_params, neighbour_range, floating_point):
    neighbours = []
    iterations = 0
    # Scan through number of neighbours.
    i = 0
    while i < meta['sa_num_neighbours']:
        iterations += 1
        # Create neighbour
        a_neighb = self.create_a_neighbour(a_chromosome, neighbour_range, num_chrom_params)
        best_chrom = self.best_chromosomes[0].chrom
        distance_from_best = self.canberra_distance(a_neighb, best_chrom)
        # Round to 5 floating point digits as otherwise the change is too insignificant.
        distance_from_best = round(distance_from_best, floating_point)
        # Append neighbour if distance does not differ too much and chrom is not in tabu dict.
        if distance_from_best not in self.sa_tabu_dict:
            neighbours.append(a_neighb)
            self.sa_tabu_dict[distance_from_best] = a_neighb
            i += 1
        else:
            # It might be that our mutation is too small, hence slowly increase rounding number.
            if random() < 0.4:
                floating_point += 1
                print(' - Increased floating precision to %d points' % floating_point)
```

Code Snippet 10

```
if not_improving_count >= meta['sa_non_improvement_threshold']:
    # Randomly select TOP3 solutions from the current SA population.
    if self.sa_best_selected_count < 10:
        solutions = sorted(solutions, key=lambda x: x.fitness, reverse=True)
        r_index = rand.randint(0, 2)
        print('\n ******** solutions[%d] selected due to non-improving search.'
              % r_index)
        current = solutions[r_index]
```

Code Snippet 11

```python
# Function for creating one neighbour.
def create_a_neighbour(self, a_chromosome, neighbour_range, num_chrom_params):
    # Create mutation vector.
    mutat_vec = (np.random.rand(num_chrom_params) * neighbour_range)
    r_divisor = rand.randint(4, 6)
    smaller_vec = (np.random.rand(num_chrom_params) * (neighbour_range / r_divisor))
    mutat_vec -= smaller_vec
    new_chromosome = copy.deepcopy(a_chromosome)
    for i, mut in enumerate(mutat_vec):
        if new_chromosome[i] == 0.99999999999:
            new_chromosome[i] -= mut
        elif new_chromosome[i] == 0 or random() > 0.5:
            new_chromosome[i] += mut
        else:
            new_chromosome[i] -= mut
    # Force boundaries on new chromosome.
    np.clip(new_chromosome, 0, 0.99999999999, out=new_chromosome)
    return new_chromosome
```

Code Snippet 12

```python
def canberra_distance(self, chrom_a, chrom_b):
    return distance.canberra(chrom_a, chrom_b)
```

Code Snippet 13

```python
# If DE improved, reset the counter.
if self.best_val_accuracy != initial_best:
    self.de_not_improving_count = 0
    initial_best = self.best_val_accuracy
# Increase exploitation and refill population every 5 generations.
if j == 5:
    self.de_accuracy_list.append(self.best_val_accuracy)
    self.fill_population(meta['de_1st_half_fill_factor'])
    meta['de_1st_half_fill_factor'] += 0.1
    meta['de_r_offser_lower'] += 0.0002
    meta['de_r_offser_upper'] -= 0.0005
    j = 0
```

Code Snippet 14

```python
def mutate_differential(self, chrom_index):
    """..."""
    # Pick chromosomes a, b, c at random
    candidates = list(range(0, meta['pop_count']))
    # Must be distinct from the chromosome that we are mutating.
    candidates.remove(chrom_index)
    r_indices = sample(candidates, 3)

    chrom_a = self.population[r_indices[0]]
    chrom_b = self.population[r_indices[1]]
    chrom_c = self.population[r_indices[2]]

    # Add the weighted difference of two chromosomes onto the third.
    mutated_chrom = chrom_a - chrom_b
    mutated_chrom = (mutated_chrom * meta['mut_factor']) + chrom_c
    np.clip(mutated_chrom, 0, 0.99999999999, out=mutated_chrom)

    # Mutate random offset.
    chrom_copy = copy.deepcopy(mutated_chrom)
    number_of_genes = rand.randint(meta['de_r_gene_lower'], meta['de_r_gene_upper'])
    self.mutate_random_offset(
        mutated_chrom, number_of_genes, meta['de_r_offser_lower'], meta['de_r_offser_upper']
    )
    distance = self.canberra_distance(mutated_chrom, chrom_copy)
    print('     ----> Offset mutation distance from differential: %.2f' % distance)

    np.clip(mutated_chrom, 0, 0.99999999999, out=mutated_chrom)
    return mutated_chrom
```

Code Snippet 15

```python
# When we exceeded 10, start selecting either local or global best individual.
else:
    if random() > 0.5:
        print('\n ******** self.best_chromosomes[0] selected due to non-improving search.')
        current = self.best_chromosomes[0]
    else:
        solutions = sorted(solutions, key=lambda x: x.fitness, reverse=True)
        current = solutions[0]
        print('\n ******** solutions[0] selected due to non-improving search.')
```

Code Snippet 16

```python
distance = self.canberra_distance(child, current)
# Check if new chromosome is not too far away.
while distance > meta['de_max_distance']:
    print('     ----> Distance (%.2f) is too large, mutating chromosome once again' % distance)
    mutated = self.mutate_differential(j)
    child = self.uniform_crossover(mutated, current, meta['p_crossover_de'])
    distance = self.canberra_distance(child, current)
```

Code Snippet 17

```
if i > meta['ga_exploitation_iter']:
    meta['random_threshold_lower'] -= 0.01
    meta['random_threshold_upper'] -= 0.005
    if not exploitation:
        meta['p_crossover_ga'] = 0.45
        exploitation = True
```

Code Snippet 18

```
for i in range(TEST_ITERATIONS):
    test_case = TestCase()
    test_case.search()
    if test_case.sa_iterations != 0:
      test_case.average_sa_improvement /= test_case.sa_iterations
    # Round the accuracy list.
    for j,_ in enumerate(test_case.de_accuracy_list):
        test_case.de_accuracy_list[j] = round(test_case.de_accuracy_list[j], 2)
```

Code Snippet 19

```
def initialize_population(self):
    print('Initializing population ...')
    a_rand_chrom = gen_rand_chromosome(self.num_chrom_param)
    self.population.append(a_rand_chrom)
    sum = 0
    i = 1
    mutation_factor = 0.05
    # VEctor is going to be added on random chromosomes as they tend to fall into same range.
    # ... and then we can't generate chromosomes with different distance.
    random_vector = np.random.rand(self.num_chrom_param) + mutation_factor

    distance_list = []
    while i < meta['pop_count']:
        a_rand_chrom = gen_rand_chromosome(self.num_chrom_param)
        a_rand_chrom = a_rand_chrom * random_vector
        np.clip(a_rand_chrom, 0, 0.99999999999, out=a_rand_chrom)
        distance = self.canberra_distance(self.population[0], a_rand_chrom)
        for dist in distance_list:
            difference = abs(dist - distance)
            if difference < meta['pop_dist_threshold']:
                mutation_factor += 0.001
                if random() > 0.5:
                    random_vector = np.random.rand(self.num_chrom_param) + mutation_factor
                else:
                    random_vector = np.random.rand(self.num_chrom_param) - mutation_factor
                break
        else:
            print(' - Distance between pop[0] and pop[%d]: %.2f' % (i, distance))
            distance_list.append(distance)
            self.population.append(a_rand_chrom)
            sum += distance
            i += 1
```

Code Snippet 20

Mutation Techniques

Mutate Offset

```
def mutate_offset(self, chrom, no_of_genes, offset):
    """
    Mutates given chromosome by randomly mutating specified number of genes.
    We add offset on selected genes.
    """
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, no_of_genes)
    for gene in genes:
        chrom[gene] = chrom[gene] + offset
    # Force boundaries on new mutated chromosome.
    np.clip(chrom, 0, 0.99999999999, out=chrom)
```

Swap Mutation

```
def swap_mutation(self, chrom, no_of_genes):
    # Assure that we have even amount of genes.
    if no_of_genes % 2 == 1:
        no_of_genes += 1
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, no_of_genes)
    a,b = split_list(genes)
```

Mutate Distance

```
def mutate_distance(self, chrom, lower_bound, upper_bound):
    i = 0
    j = 1
    while j < len(chrom) - 1:
        difference = abs(chrom[i] - chrom[j])
        chrom[j + 1] += difference
        r_offset = rand.uniform(lower_bound, upper_bound)
        chrom[i] += r_offset
        chrom[j] -= r_offset
        j += 1
        i += 1
    # Force boundaries on a new mutated chromosome.
    np.clip(chrom, 0, 0.99999999999, out=chrom)
```

Mutate Random

```
def mutate_random(self, chrom, no_of_genes):
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, no_of_genes)
    for gene in genes:
        chrom[gene] = random()
```

Mutate Random Factor

```
def mutate_random_factor(self, chrom, no_of_genes, lower_bound, upper_bound):
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param - 1))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, no_of_genes)
    for gene in genes:
        r_factor = rand.uniform(lower_bound, upper_bound)
        chrom[gene] = chrom[gene] * r_factor
    # Force boundaries on a new mutated chromosome.
    np.clip(chrom, 0, 0.99999999999, out=chrom)
```

Mutate Random Offset Factor

```
def mutate_random_offset_factor(self, chrom, g_mutation_count, lower_bound, upper_bound):
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param - 1))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, g_mutation_count)

    for gene in genes:
        r_offset = rand.uniform((lower_bound * -1), upper_bound)
        r_factor = rand.uniform(lower_bound, upper_bound)
        chrom[gene] = (chrom[gene] + r_offset) * r_offset

    # Force boundaries on a new mutated chromosome.
    np.clip(chrom, 0, 0.99999999999, out=chrom)
```

Mutate Random Offset

```
def mutate_random_offset(self, chrom, g_mutation_count, lower_bound, upper_bound):
    """
    Mutation applied in Simulated Annealing algorithm.
    Applies random offset on given number of genes to be mutated.
    """
    # Every gene is a possible candidate for mutation.
    candidates = list(range(0, self.num_chrom_param - 1))
    # Randomly pick a specified amount of genes to be mutated.
    genes = sample(candidates, g_mutation_count)

    for gene in genes:
        r_offset = rand.uniform(lower_bound, upper_bound)
        chrom[gene] = chrom[gene] + r_offset

    # Force boundaries on a new mutated chromosome.
    np.clip(chrom, 0, 0.99999999999, out=chrom)
```

Reproduction Methods

Tournament Selection

```
def tournament_selection(self, population):
    tournament = rand.sample(population, meta['tournament_size'])
    # Sort population by fitness (descending), pick out the best chromosome.
    tournament = sorted(tournament, key=lambda x: x.fitness, reverse=True)
    return tournament[0]
```

Roulette Wheel Selection

```
def roulette_w_selection(self, population):
    fitness_sum = sum(chrom.fitness for chrom in population)
    probs = [(x.fitness / fitness_sum) for x in population]
    for i, prob in enumerate(probs):
        if random() < prob:
            return population[i]
    return population[0]
```

Stochastic Universal Sampling

```
def SUS(self, population, number_of_parents):
    """..."""
    total_fitness = sum(chrom.fitness for chrom in population)
    point_distance = total_fitness / number_of_parents
    start_point = rand.uniform(0, point_distance)
    points = [start_point + i * point_distance for i in range(number_of_parents)]

    # Set prevents from adding the same individual as both mom and dad.
    parents = set()
    while len(parents) < number_of_parents:...

    return list(parents)
```

Crossover Methods

Uniform Crossover

```
def uniform_crossover(self, dad, mom, p_crossover):
    """..."""
    child = gen_rand_chromosome(self.num_chrom_param)
    # Uniform crossover - we assign probability to each gene.
    for i in range(len(child)):...
    return child
```

Single Point Crossover (1PTX)

```
def single_point_crossover(self, dad, mom):
    r = rand.randrange(1, len(dad) - 1)

    girl = np.concatenate([dad[:r], mom[r:]])
    boy = np.concatenate([mom[:r], dad[r:]])
    return girl, boy
```

Two Point Crossover (2PTX)

```
def two_point_crossover(self, dad, mom):
    # Select two random points
    p = sample(range(len(dad)), 2)
    # I am not sure if sample returns sorted list hahah.
    p = sorted(p)

    girl = np.concatenate([dad[:p[0]], mom[p[0]:p[1]], dad[p[1]:]])
    boy = np.concatenate([mom[:p[0]], dad[p[0]:p[1]], mom[p[1]:]])
    return girl, boy
```

Ring Crossover

```
def ring_crossover(self, dad, mom):
    """..."""
    ring = np.concatenate([dad,mom])
    cutting_point = rand.randrange(len(ring))

    girl = gen_rand_chromosome(self.num_chrom_param)
    i = 0
    j = cutting_point
    while i < len(dad):...

    boy = gen_rand_chromosome(self.num_chrom_param)
    i = 0
    j = cutting_point
    while i < len(dad):...

    return girl, boy
```

Heuristic Crossover

```
def heuristic_crossover(self, dad, mom):
    """..."""
    if self.get_fitness(dad) > self.get_fitness(mom):...
    else:...

    girl = best_parent + random() * (best_parent - worst_parent)
    boy = best_parent
    return girl, boy
```

Statistics

GA performance with different parent selection techniques (20 trials, validation accuracy)

Random	Roulette Wheel	SUS	Tournament Selection
74.8	78.3	74.6	77.4
75.6	77.8	75.4	76.5
72.9	72.7	78	77.5
76.7	79.5	71.3	79
73.1	73	74.9	74.4
75	78.7	75.9	76.7
76.1	72.2	78.9	73.2
70.9	77.8	76.9	72.7
72.1	75.9	72.3	73.3
77.3	76.4	71.8	73.1
69.6	74	73.4	72.7
75.6	77.3	78	73.6
71.5	75.8	76.4	79.3
78.3	76.4	75	73.8
78.6	77.5	75.9	73.7
74	77.3	73.6	76.9
72.2	76.7	74.8	74.9
75.3	73.1	76.5	74.9
73.8	74.5	74	80
74.2	75	74.8	71.8
AVERAGE			
74.38	75.995	75.12	75.27

Statistics 1

GA performance with different crossover methods (20 trials, validation accuracy)

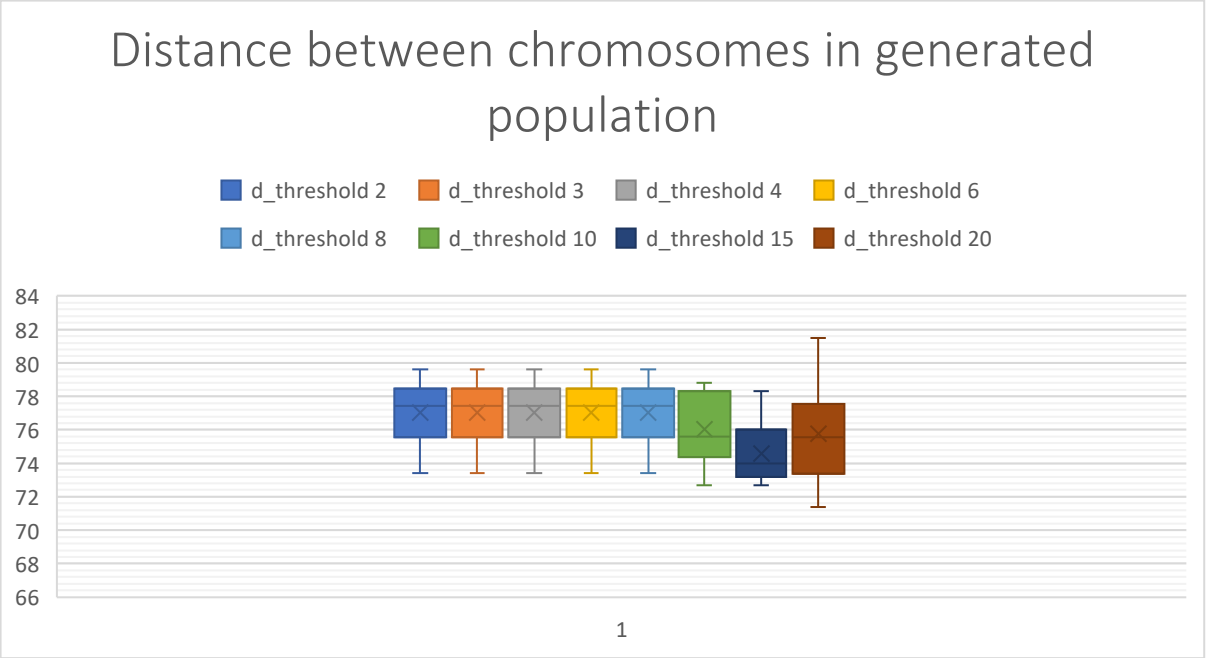
1PTX	2PTX	Heuristic	Ring	Uniform
73.9	72.4	74.9	73.2	78.3
76.2	76.7	74.4	77.2	77.8
76.5	74.6	70.7	73.1	72.7
72.5	73.9	79.1	73.1	79.5
73	71.3	72.4	77.5	73
77.3	71.7	76.7	74.4	78.7
72.6	78.3	71	75.7	72.2
75.7	73.8	74.8	71	77.8
75.2	70.8	74.7	74.1	75.9
72	70.2	74.8	76.7	76.4
72.8	74	71.3	73.5	74
74.2	77.2	78.9	76.5	77.3
76	72.1	77.6	74.4	75.8
78.6	74.5	77	74.5	76.4
72.7	75	78.6	75.3	77.5
74.2	73.6	76.8	78.4	77.3
76	77.1	71.9	76.3	76.7
73.6	75.9	75.1	75.2	73.1
78.8	76.6	74.4	72.8	74.5
77.3	73.6	73.1	73.5	75
AVERAGE				
74.955	74.165	74.91	74.82	75.995

Statistics 2

GA performance with different mutation techniques (20 trials, validation accuracy)

mutate_distance	mutate_offset	mutate_r_f	mutate_r_o_f	mutate_random	random_offset	swap
73.8	77.1	72.9	57.8	46.5	61.5	12.1
73	74.8	72.7	65.1	70	58.8	58
77.7	77.3	76.6	60.3	56.9	52.3	19
76.3	72	77.2	63.2	41.7	76	10.9
78.6	76.1	69.5	58.1	43.2	64.7	61.2
79.3	77	72.5	67.6	56.8	62.2	27.3
79	76.8	63.2	67.9	44.9	64.7	13
75.6	75.7	66.2	69.7	36.9	58.2	25.7
77	75.5	72.5	56.1	21.2	62.5	65.3
78.3	74	78.8	61.2	27.5	60.5	11.1
69.3	76.9	76.8	75.8	27.2	50.4	50.7
74.8	71.2	64.2	65	69.1	63.4	62.2
74	75.7	77.9	59.8	31.5	57.1	51.4
77.4	77.7	76.4	62.6	50.4	62.9	34.5
76.3	76.5	71.1	64.6	67.4	66.7	64.2
78.4	74.6	71.8	53.4	55.6	61.5	18.9
75.4	73.8	74.1	61.5	31.3	68	53
75.5	70.8	70.8	68.7	65.3	49.6	55.8
75.5	72.6	70.1	63.3	38.3	53.2	61.3
71.8	69.7	68.8	50.3	12.6	66.2	23.4
AVERAGE						
75.85	74.79	72.205	62.6	44.715	61.02	38.95

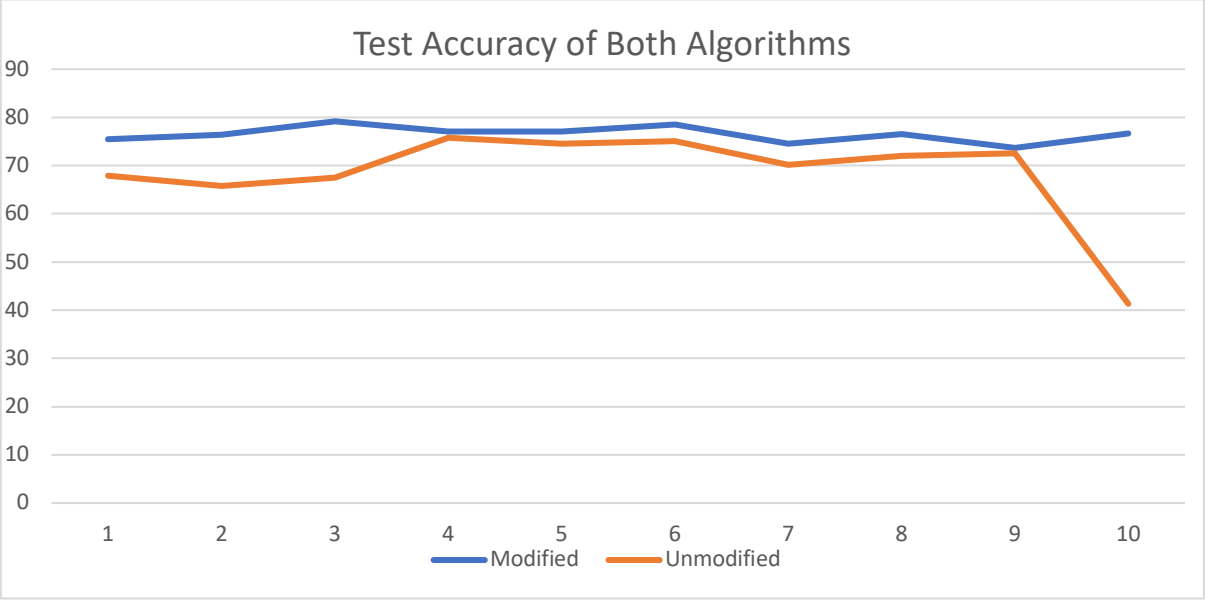
Statistics 3



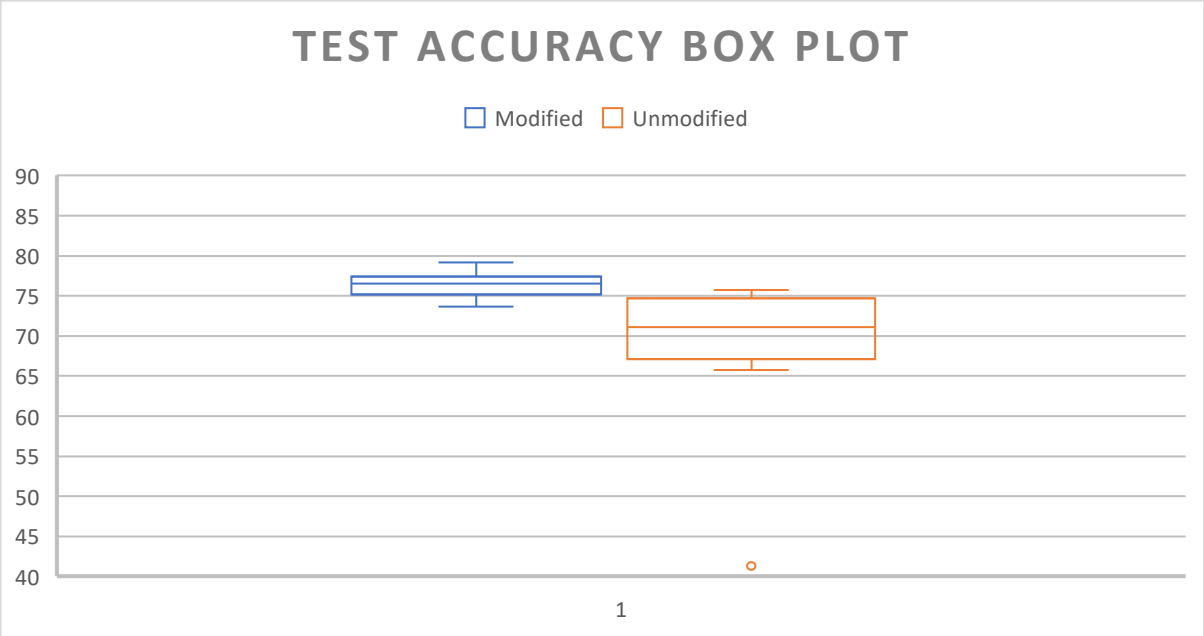
Statistics 4

ID	Final (Test) Accurac	Accuracy after GA	GA forced explr	PSO improvement	Best PSO fitness	PSO iterations	DE improvement	DE accuracy	DE accuracy list	Average SA improve	SA alternative chrom	SA count	Time elapsed	AVG pop. distance	Total pop. distance
0	75.43	77.3	0	0	77.3	6	1.423	78.4	[78.1; 78.1; 78.1; 78	2.504	162	4	301.49	180.01	1800.13
1	76.46	76.5	0	0.915	77.2	15	0.389	77.5	[77.2; 77.2; 77.2; 77	4.043	163	4	469.57	177.04	1770.37
2	79.18	77.6	0	0	77.6	6	1.16	78.5	[77.6; 77.6; 77.6; 78	0.641	41	1	168.77	178.6	1786
3	77.09	75.8	0	0.66	76.3	15	0.524	76.7	[76.4; 76.4; 76.4; 76	0	82	2	219.14	182.15	1821.51
4	77.08	75.3	0	0	75.3	8	1.992	76.8	[75.3; 75.3; 75.3; 76	2.949	158	4	292.58	178.88	1788.78
5	78.48	80.3	0	0	80.3	6	0.747	80.9	[80.3; 80.7; 80.7; 80	4.28	159	4	302.51	179.84	1798.43
6	74.5	78.8	0	0	78.8	6	1.015	79.6	[78.8; 78.8; 78.8; 79	3.388	160	4	302.64	178.4	1784.03
7	76.49	77.1	0	0	77.1	6	0.778	77.7	[77.6; 77.6; 77.6; 77	2.297	159	4	283.96	177.96	1779.63
8	73.67	77.8	0	0	77.8	6	1.414	78.9	[78.9; 78.9; 78.9; 78	3.107	160	4	290	175.88	1758.83
9	76.6	76.3	0	0	76.3	6	1.442	77.4	[76.7; 77.4; 77.4; 77	0.658	164	4	307.06	183.08	1830.84

Statistics 5



Statistics 6



Statistics 7

Significance Level:

- ○ 0.01
- ◉ 0.05

1 or 2-tailed hypothesis?:

- ○ One-tailed
- ◉ Two-tailed

Result Details

Sample 1
Sum of ranks: 62
Mean of ranks: 6.2
Expected sum of ranks: 105
Expected mean of ranks: 10.5
U-value: 93
Expected *U*-value: 50

Sample 2
Sum of ranks: 148
Mean of ranks: 14.8
Expected sum of ranks: 105
Expected mean of ranks: 10.5
U-value: 7
Expected *U*-value: 50

Sample 1 & 2 Combined
Sum of ranks: 210
Mean of ranks: 10.5
Standard Deviation: 13.2288

Result 1 - U-value

The *U*-value is 7. The critical value of *U* at $p < .05$ is 23. Therefore, the result is significant at $p < .05$.

Result 2 - Z-ratio

The *Z*-Score is -3.2127. The *p*-value is .00132. The result is significant at $p < .05$.

Statistics 8