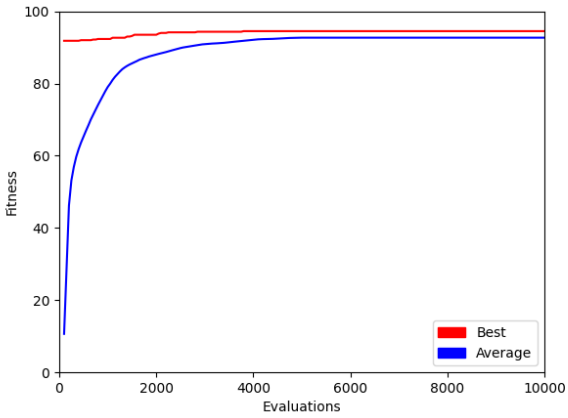
Fall 2020 Assignment 1B

Jordan Sosnowski
jjs0029@auburn.edu

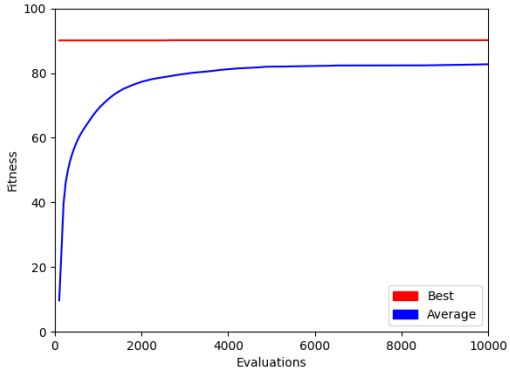
COMP 6660

Evals vs Average Local Fitness and Evals vs Best Fitness Plots

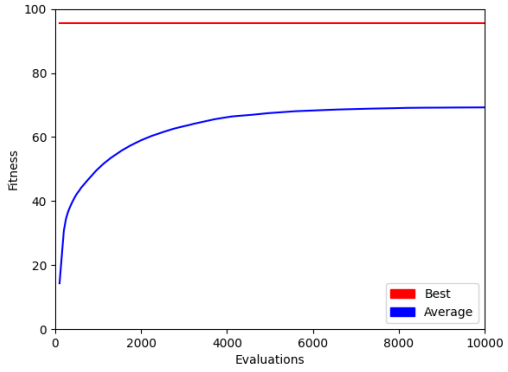
For running our EA against problem A1 we generate the below figure. Since this is an easy problem the graph is what I would expect.



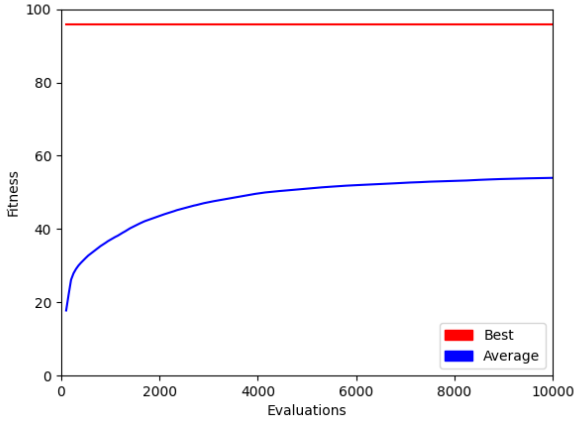
For running our EA against problem A2 we generate the below figure. Similar to A1, A2 is fairly simple albeit a litter harder which explains the lower values and less steep slope.



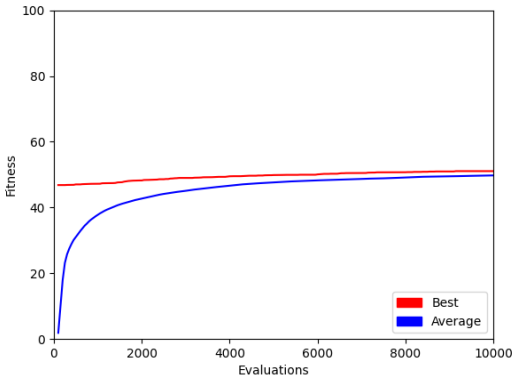
For running our EA against problem A3 we generate the below figure. For the next few graphs, the gap between the best and average fitness is a little startling but I believe it is due to the paring of the truncation survival algorithm and SUS parent algorithm.



For running our EA against problem B1 we generate the below figure.



For running our EA against problem B2 we generate the below figure.



Statistical Analysis for Random Search and Evolutionary Algorithms for A Problems

For this section, we were asked to compare the results of the Random Search algorithm against our EA. Below is a pseudo-code implementation of how the random search algorithm would perform. This algorithm simply executes two for loops and for each execution of the inner loops it generates a random possible solution of the board. After the solution is generated it is logged.

```
board = [[]..] # array implementation of game board
for run in {0..10}:
    for evaluation in {0..10,000}
        individual = uniform_random_selection(board)

        logging(individual)
```

Here we have a pseudo-code implementation of how the EA would perform. The main difference between the two is the passing of `genes` between parents and children. Additionally, something I did not realize until later, was that per individual created the evaluation ticker goes up. Therefore, if you are using several evaluations for termination criteria you do not run n generations where n is the max number of evaluations, but n/μ+λ where μ is the number of children produced each run and λ is the number of parents and the number of initial individuals created.

```
board = [[]..] # array implementation of game board
for run in {0..10}:
    population = init_pop(board)
    while not terminated
        parents = parent_selection(population)
        children = child_selection(parents, board)
        population = survival_selection(population, children)


        logging(individual)

        terminated = check_termination_criteria()
```

Here we can see the statistical data of running the random algorithm and the EA against problems A1, A2, A3. Both perform quite well on A1 which is to be expected as it is a simple problem.

EA	Random
100	100
90	100
90	100
90	100
80	100
100	100
95	100
100	100
100	100
80	100
100	100
100	100
100	100
90	100
100	100
90	100
90	100
95	95
80	100
100	100
100	100
90	100
95	100
100	100
100	100
100	100
95	100
95	100
100	100
90	95

stddevS	stddevS
6.47941	1.268541

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	94.5	99.666667
Variance	41.98275862	1.6091954
Observations	30	30
df	29	29
F	26.08928571	
P(F<=f) one-tail	4.11553E-14	
F Critical one-tail	1.860811435	

mean(var1) < mean(var2) and F > F Crit assume equal

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	94.5	99.666667
Variance	41.98275862	1.6091954
Observations	30	30
Pooled Variance	21.79597701	
Hypothesized Mean Difference	0	
df	58	
t Stat	-4.286155296	
P(T<=t) one-tail	3.47292E-05	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	6.94583E-05	
t Critical two-tail	2.001717484	

On A2 we see both perform worse than A1, I expected the EA to perform better but due to the random nature of both algorithms I may have just been a certain experiment that was running.

EA	Random
90	93
90	93
85	97
88	95
95	90
97	97
95	90
90	88
93	97
80	95
88	95
97	95
80	93
88	95
97	95
93	90
80	93
97	97
88	90
90	95
90	97
90	95
90	93
90	95
90	93
93	95
90	95
90	88
93	97
88	90

stddevS	stddevS
4.646714	2.756184

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	90.16666667	93.7
Variance	21.59195402	7.596551724
Observations	30	30
df	29	29
F	2.842336208	
P(F<=f) one-tail	0.003172991	
F Critical one-tail	1.860811435	

mean(var1) < mean(var2) and F > F Crit assume equal

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	90.16666667	93.7
Variance	21.59195402	7.596551724
Observations	30	30
Pooled Variance	14.59425287	
Hypothesized Mean Difference	0	
df	58	
t Stat	-3.582113204	
P(T<=t) one-tail	0.000349083	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.000698166	
t Critical two-tail	2.001717484	

However, one A3 we see that the EA vastly outperformed the random algorithm which is expected.

EA	Random
88	75
96	73
96	80
99	85
99	79
93	80
95	79
97	80
95	83
97	76
95	69
95	79
96	80
99	75
96	77
96	77
97	83
99	76
93	76
93	77
97	88
93	80
96	76
97	79
95	77
96	77
95	72
96	83
96	81
95	81

stddevS	stddevS
2.233496	3.901223

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	95.66666667	78.43333333
Variance	4.988505747	15.21954023
Observations	30	30
df	29	29
F	0.327769806	
P(F<=f) one-tail	0.001829998	
F Critical one-tail	0.537399965	

mean(var1) > mean(var2) and F < F Crit assume equal

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	95.66666667	78.43333333
Variance	4.988505747	15.21954023
Observations	30	30
Pooled Variance	10.10402299	
Hypothesized Mean Difference	0	
df	58	
t Stat	20.99750798	
P(T<=t) one-tail	4.3687E-29	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	8.73739E-29	
t Critical two-tail	2.001717484	

Stochastic Uniform Sampling vs Fitness Proportional Selection

For the parent selection, we were required to go with three different algorithms: stochastic uniform sampling, fitness proportion selection, and k-tournament selection.

Out of the three SUS was the hardest to wrap my head around. Luckily, the book provided pseudocode for an implementation that I borrowed.

```
def stochastic_uniform_sampling(self, individuals):
    """ Selects parents randomly

    individuals - list of Individual objects
    """
    fitness_sum = sum(individuals)
    if fitness_sum == 0:
        return random.choices(individuals, k=self.children*2)

    probs = [individual.fitness /
                fitness_sum for individual in individuals]

    for i in range(1, len(probs)):
        probs[i] = probs[i] + probs[i-1]

    mating_pool = []

    r = random.random()/(self.children*2)

    current_member = 0
    i = 0

    while current_member < (self.children*2
                            ):
        while r < probs[i]:
            mating_pool.append(individuals[i])
            r += 1/(self.children*2)
            current_member += 1
        i += 1

    return mating_pool
```

Following this, we have the FPS algorithm which I believe I implemented correctly. The selection for this is based on the absolution fitness of all the individuals. Based on this each individual is selected randomly with replacement.

```
def fitness_proportional_selection(self, individuals):
    """ Selects parents based on fitness probability.

    individuals - list of Individual objects
    """

    if self.windowed:
        # f'(x) = f(x) - β + 1; where β = min(fitness)
        min_fit = min(individuals).fitness
        for individual in individuals:
            individual.windowed(min_fit)

    fitness_sum = sum(individuals)
    if fitness_sum == 0:
        return random.choices(individuals, k=self.children*2)

    probs = [individual.fitness /
                fitness_sum for individual in individuals]

    self.cleanup(individuals)

    return random.choices(individuals, probs, k=self.children*2)
```

Lastly, we have tournament selection, due to time constraints I, unfortunately, did not run many experiments with this beyond just initial testing. For this instead of absolute fitness, relative fitness is used.

```
def tournament_selection_parent(self, individuals):
    """ Selects parents using a tournament based system. Grabs k individuals
    and selects the best individual out of that set.

    For parent selection we will be using replacement

    individuals: list of Individual objects
    """

    chosen_parents = []
    for _ in range(self.children*2):

        selection = random.choices(individuals, k=self.tournament_parent)

        selection_sorted = sorted(selection)
        chosen_parents.append(selection_sorted.pop())
    return chosen_parents
```

Here we see some statistical analysis for SUS and FPS. Both perform quite well for the problem they ran against which was B1. We can see that FPS has a higher variance than SUS which makes sense based on the biased it can have.

SUS	FPS
96	97
95	97
96	96
96	93
96	97
95	99
95	96
96	96
98	94
96	94
96	97
99	93
98	96
96	99
97	93
95	97
96	96
96	96
93	96
93	96
96	96
96	94
98	99
96	93
98	97
96	96
94	94
94	96
93	97
96	99

stddevS	stddevS
1.487496	1.809617

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	95.83333333	95.96666667
Variance	2.212643678	3.274712644
Observations	30	30
df	29	29
F	0.675675676	
P(F<=f) one-tail	0.148415526	
F Critical one-tail	0.537399965	

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	95.83333333	95.96666667
Variance	2.212643678	3.274712644
Observations	30	30
Pooled Variance	2.743678161	
Hypothesized Mean Difference	0	
df	58	
t Stat	-0.311758127	
P(T<=t) one-tail	0.378171145	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.75634229	
t Critical two-tail	2.001717484	

Investigation of Variation Operators

For my framework, I chose initially to go with a one-point crossover as the seemed easiest to implement. I did not realize till later that due to my gene implementation there would not be an equal crossover between parents. In the future, I believe I can solve this in the same way that I used in the uniform crossover function. which will be discussed shortly.

For the one-point crossover implementation, I pick a random value between 0 and 1. This is then multiplied against the length of both parents' gene pools to scale to each pool. After this, I determine from which parent I should start from and which parent I should end with. Following the merging of the genes, I return the child's solution to be evaluated.

```
def one_point_crossover(parent_one, parent_two):
    val = random.random()
    child_solution = []

    parent_one_scale = round(parent_one.iterations * val)
    parent_two_scale = round(parent_two.iterations * val)
    if random.randint(0, 1):
        child_solution += parent_one.solution[parent_one_scale:]
        child_solution += parent_two.solution[:parent_two_scale]
    else:
        child_solution += parent_one.solution[:parent_one_scale]
        child_solution += parent_two.solution[parent_two_scale:]

    return child_solution
```

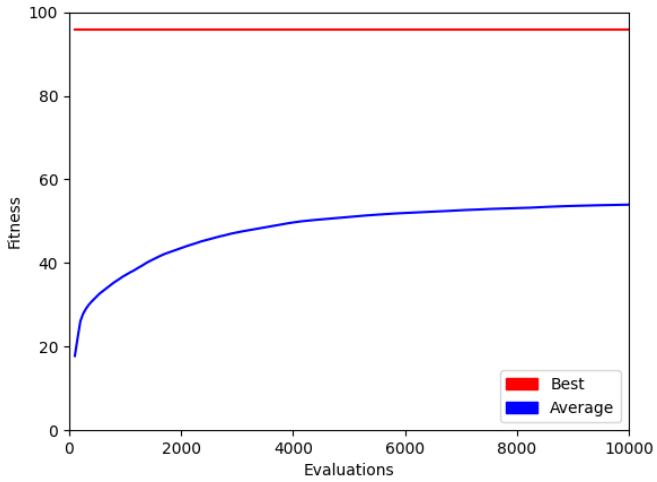
The additional recombination function I created was a uniform crossover implementation. With this instead of splicing the gene pools and selecting a range from each parent, each gene is stochastically determined to pass on to the parent. I first extend the parent's gene pools artificially, so they match in length. Following this, the genes are iterated and with a coin flip, we determine which parent the gene should be inherited from.

```
def uniform_crossover(self, parent_one, parent_two):
    max_iterations = parent_one.iterations if parent_one.iterations > parent_two.iterations \
        else parent_two.iterations

    p1 = self.list_extend(parent_one.solution,
                          abs(parent_one.iterations - max_iterations))
    p2 = self.list_extend(parent_two.solution,
                          abs(parent_two.iterations - max_iterations))

    child_list = []
    while not child_list:
        for x in range(max_iterations):
            if random.randint(0, 1):
                child_list.append(p1[x])
            else:
                child_list.append(p2[x])

        # clear out temp 0 values
        child_list = [x for x in child_list if x != 0]

    return child_list
```

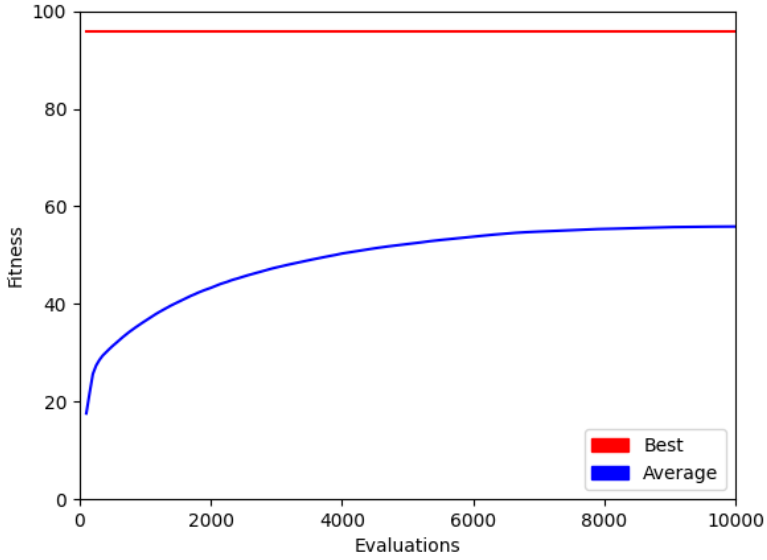
Unfortunately, as seen in the plots below the two implementations do not seem to have a huge effect on overall fitness. I believe this is due mainly to the survivor implementation and not the child selection algorithm. For both runs, I used truncation for the survivor algorithm, and due to running out of time, I was unable to test additional ones with the uniform sampling algorithm.

Added with the plots are some statistical analysis. Throughout this project, only problem B2 when comparing uniform against one point produced datasets that were not equal in variance.

B1 One-Point



B1 Uniform



Uniform	One Point
93	91
95	95
96	99
97	96
98	96
96	96
96	94
96	93
95	95
94	99
93	96
96	95
96	96
98	96
97	93
96	97
97	95
96	96
98	96
95	97
96	99
96	96
96	94
96	95
96	96
95	98
97	97
93	98
98	99
95	92

stddevS	stddevS
1.382984	2.018592

F-Test Two-Sample for Variances

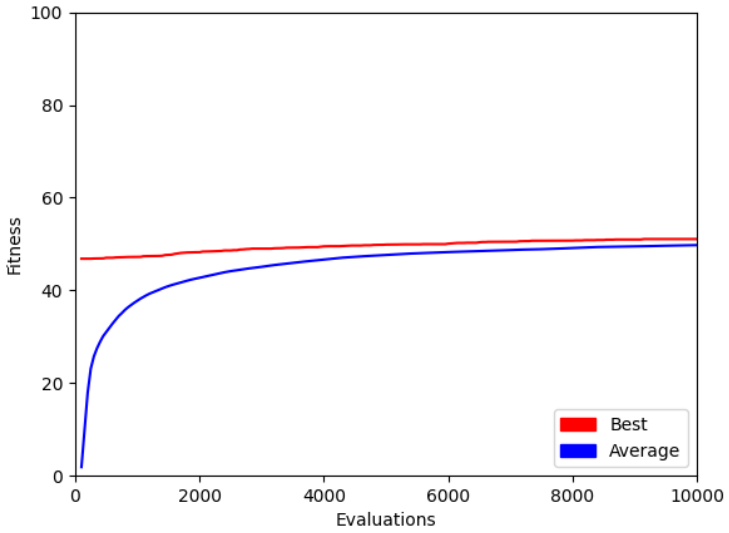
	Variable 1	*Variable 2*
Mean	95.86666667	95.83333333
Variance	1.912643678	4.074712644
Observations	30	30
df	29	29
F	0.469393512	
P(F<=f) one-tail	0.022975162	
F Critical one-tail	0.537399965	

mean(var1) > mean(var2) and F < F Crit assume equal

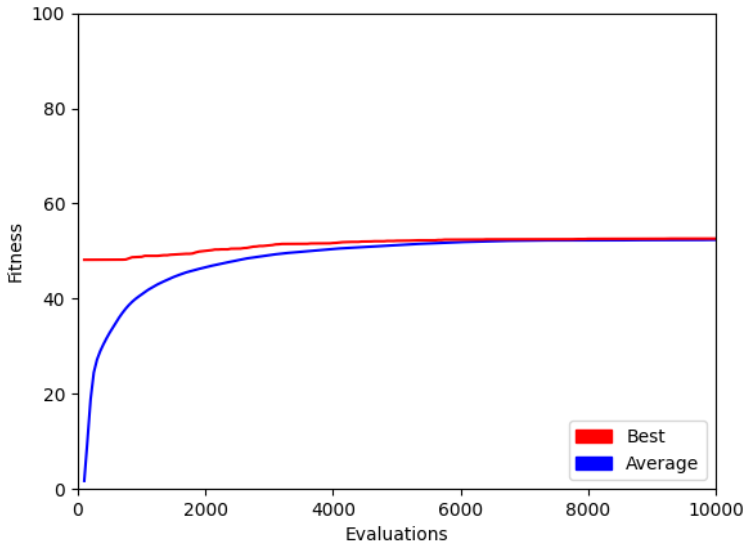
t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	95.86666667	95.83333333
Variance	1.912643678	4.074712644
Observations	30	30
Pooled Variance	2.993678161	
Hypothesized Mean Difference	0	
df	58	
t Stat	0.074614257	
P(T<=t) one-tail	0.470389192	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.940778383	
t Critical two-tail	2.001717484	

B2 One-Point



B2 Uniform



Uniform	One Point
63	31
54	43
31	59
61	57
51	47
54	56
51	37
62	60
47	47
52	60
62	54
55	21
57	42
54	62
72	50
46	37
56	65
56	47
54	57
44	46
37	53
42	42
42	65
42	61
71	47
53	60
37	47
61	50
47	60
65	69

stddevS	stddevS
9.873278	10.92272

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	52.63333333	51.06666667
Variance	97.4816092	119.3057471
Observations	30	30
df	29	29
F	0.817073876	
P(F<=f) one-tail	0.294995463	
F Critical one-tail	0.537399965	

mean(var1) > mean(var2) and F > F Crit assume not equal

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	52.63333333	51.06666667
Variance	97.4816092	119.3057471
Observations	30	30
Hypothesized Mean Difference	0	
df	57	
t Stat	0.582800957	
P(T<=t) one-tail	0.281162966	
t Critical one-tail	1.672028888	
P(T<=t) two-tail	0.562325932	
t Critical two-tail	2.002465459	