COMP 6660 Fall 2020 Assignment 1D

Jordan Sosnowski jjs0029@auburn.edu

October 11, 2020

Introduction

For this assignment the author was asked to implement a Multi-Objective EA (MOEA); unlike past assignments where the EA optimizes over a single objective, MOEA looks at multiple objectives. In prior assignments the objective was the notion of *fitness*, this time there are three sub-objectives:

1. Percentage of the board that is lit up
2. Number of black cell violations
3. Number of bulbs placed in lit cells

The first sub-objective (percentage of the board that is lit up) should be maximized while the last two should be minimized. In this assignment, the author implemented a simplified NSGA-II algorithm without crowding. Additionally, fitness sharing, crowding, and a fourth sub-objective (minimizing the number of bulbs placed) where also required to be implemented by the author.

MOEA Background

As per Dr. Tauritz's slide deck, NSGA-II works as follows.

1. Initialization
    1. Create an initial population P_0
    2. Sort P_0 on the basis of non-domination
    3. Best level is level 1
    4. Fitness is set to level number; lower number, higher fitness
    5. Use Binary (k=2) tournament selection for parent selection
    6. Mutation and Recombination create children Q_0
2. Primary Loop
    1. Create a population R_t by merging the existing population (P_t) with the new children (Q_t)
    2. Now sort R_t on the basis of non-domination
    3. Create the next generation P_{t+1} by adding the best individuals from R_t
    4. Create the next set of children (Q_{t+1}) by performing Binary Tournament Selection, Recombination and Mutation on P_{t+1}

The author's actual implementation is similar but changed a few things. To allow the new code to integrate easier with the existing code base the author sets the best level to 100; a **higher** level equals a **higher** fitness.

Domination can be defined as follows. An individual **A** is said to dominate an individual **B** iff:

- **A** is no worse than **B** in all objectives
- **A** is strictly better than **B** in at least one objective

MOEA Implementation

The author has been using a *class-based* approach instead of a *functional* one throughout this assignment series. Therefore, there is an `Individual` object which stores relevant information such as:

1. Percent of tiles lit
2. Fitness
3. Bulb locations
4. Number of black cell violations
5. Number of bulb intersections
6. Solution Name (unique)

Implementing domination is quite easy with a class setup. Classes in Python allow you to define special functions referred to as *double-under (dunder)* or *magic functions*. One example of a dunder function is operator overloading. To sort a list of individuals based on non-domination one can overload the `<` operator.

```
def __lt__(self, other):
    a = (self.lit <= other.lit) and \
        (self.bulb_violations >= other.bulb_violations) and \
        (self.black_cell_violations >= other.black_cell_violations)
    b = (self.lit < other.lit) or \
        (self.bulb_violations > other.bulb_violations) or \
        (self.black_cell_violations > other.black_cell_violations)
    return a and b
```

This previous code-snippet allows the next code-snippet to produce a sorted list of individuals using non-domination.

```
sorted_pop = sorted(population, reverse=True)
```

However to apply the `rank` an additional step is required. This step can be seen in the `calculate_moea_fitness` function. One first sorts the passed population and then loops through it applying rank based on `non-domination`.

```
@ staticmethod
def calculate_moea_fitness(population):
    sorted_pop = sorted(population, reverse=True)

    rank = 100
    for count, ind in enumerate(sorted_pop):
        ind.fitness = rank

        if count + 1 == len(sorted_pop):
            break
        if ind > sorted_pop[count + 1]:
            # new level
            rank -= 1

    return sorted_pop
```

After the rank is applied as the `fitness` for the individual the code runs the same as it did in prior assignments.

Solution Logging

For this assignment, there were additional changes to the solution file. Instead of logging the best solution across all runs the best *Pareto Front* and all of its solutions should be logged. For example, say one has two fronts **P1** and **P2**. **P1** dominates **P2** if the proportion of solutions in **P1** which dominate at least one solution in **P2** is larger than the proportion of solutions in **P2** which dominate at least one solution in **P1**.

This sorting can be similarly achieved in Python as before. With a custom `ParetoFront` class one can define the `__lt__` function as follows. Here the function adds up the number of times it dominates the other front and the number of times the other front dominates it. Finally, the proportions are compared and if the other front dominates the function returns `True`.

```python
def __lt__(self, other):
    total_this_domination_count = 0
    total_other_domination_count = 0
    for other_sol in other.solutions:
        this_domination_count = 0
        other_domination_count = 0
        for solution in self.solutions:
            if solution > other_sol:
                this_domination_count += 1
            elif other_sol > solution:
                other_domination_count += 1
        total_other_domination_count += other_domination_count
        total_this_domination_count += this_domination_count

    return (total_this_domination_count / len(self.solutions)) < \
        (total_other_domination_count / len(other.solutions))
```

MOEA Results

For this assignment, two problems were bundled with the initial repo: D1 and D2. The author was asked to implement three different configurations and to test them against the two provided problems. The configurations that were chosen were:

1. Default Configuration
    1. Parent Selection Algorithm: SUS
    2. Recombination: One Point Crossover
    3. Mutation: Creep
    4. Survival Selection: Truncation
    5. Termination Algorithm: Number of Evaluations
    6. Children: 50
    7. Population: 100
    8. Mutation Rate: 0.40
    9. Survival Strategy: +
2. “NSGA” Configuration
    1. Parent Selection Algorithm: Tournament (k=2)
    2. Recombination: One Point Crossover
    3. Mutation: Creep
    4. Survival Selection: Tournament (k=2)
    5. Termination Algorithm: Number of Evaluations
    6. Children: 50

7. Population: 100
8. Mutation Rate: 0.40
9. Survival Strategy: +

3. "Uniform" Configuration
    1. Parent Selection Algorithm: SUS
    2. Recombination: One Point Crossover
    3. Mutation: Creep
    4. Survival Selection: Uniform Random Selection
    5. Termination Algorithm: Number of Evaluations
    6. Children: 50
    7. Population: 100
    8. Mutation Rate: 0.40
    9. Survival Strategy: +

Default Configuration

One can see in Figure 1 the results of running the default configuration on problem D1. It is expected that the results for D1 perform better than D2, especially the bulb violations as D2 is more complicated than D1. This configuration took ~2386 seconds or ~40 minutes to run for D1 and ~2413 seconds or ~40 minutes for D2.

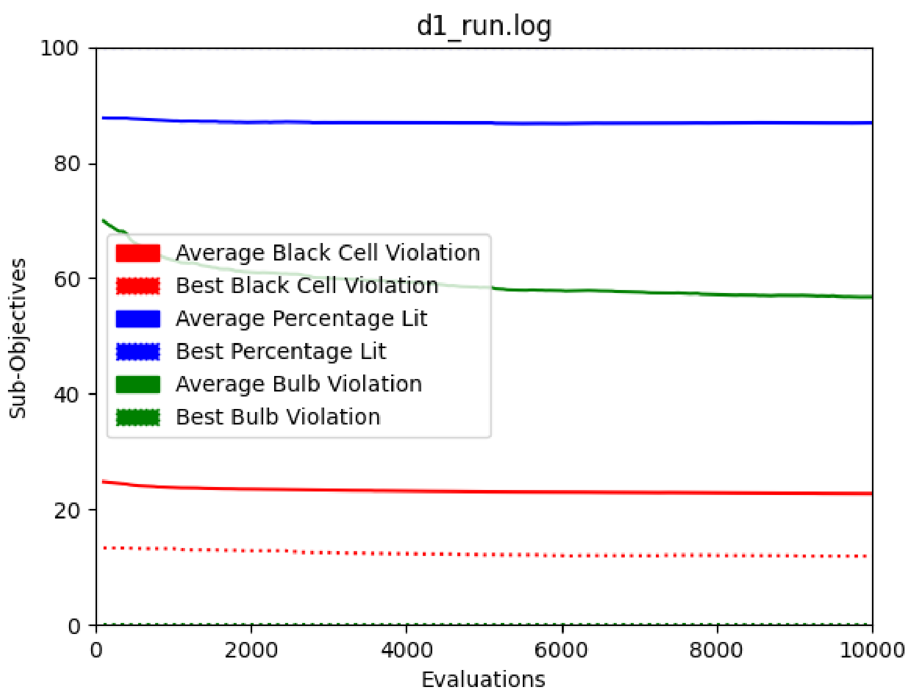

Figure 1: D1 with default configuration

One can see in Figure 2 the results of running the default configuration on problem D2.

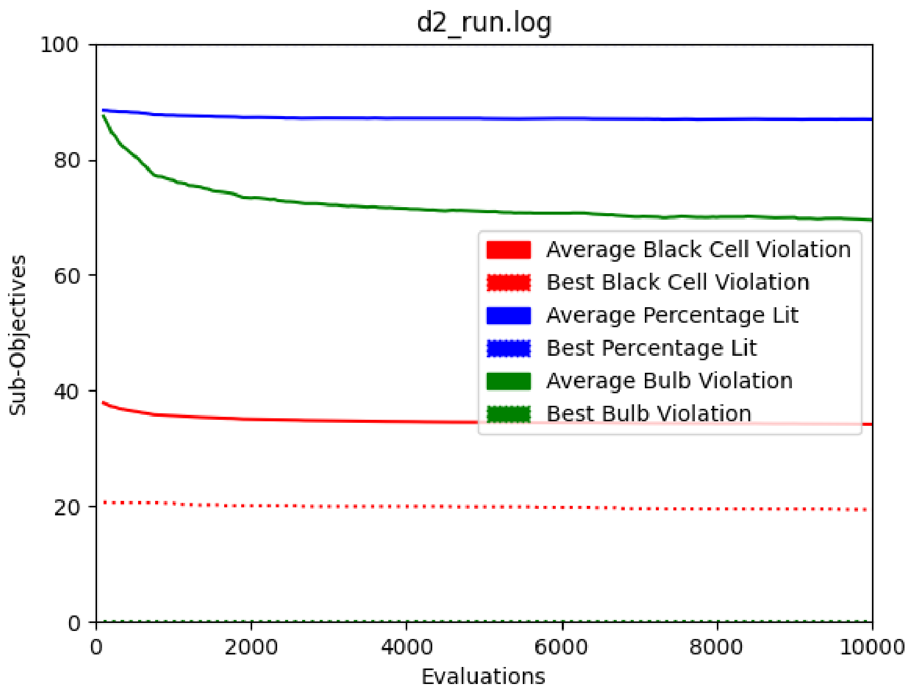

Figure 2: D2 with default configuration

"NSGA" Configuration

This configuration took ~2099 seconds or ~35 minutes to run for D1 and ~2041 seconds or ~34 minutes for D2.

One can see in Figure 3 the results of running the NSGA configuration on problem D1.

One can see in Figure 4 the results of running the NSGA configuration on problem D2.

"Uniform" Configuration

This configuration took ~1833 seconds or ~31 minutes to run for D1 and ~2022 seconds or ~34 minutes for D2.

One can see in Figure 5 the results of running the Uniform configuration on problem D1.

One can see in Figure 6 the results of running the Uniform configuration on problem D2.

Configuration Comparisons

In this section, comparisons are made between the three different configurations. t-Test statistical analysis was done to compare the two sets (note: this is the same for all comparisons made throughout this document).

D1

The comparison between the default configuration against the NSGA configuration on problem D1 can be seen in Figure 7.

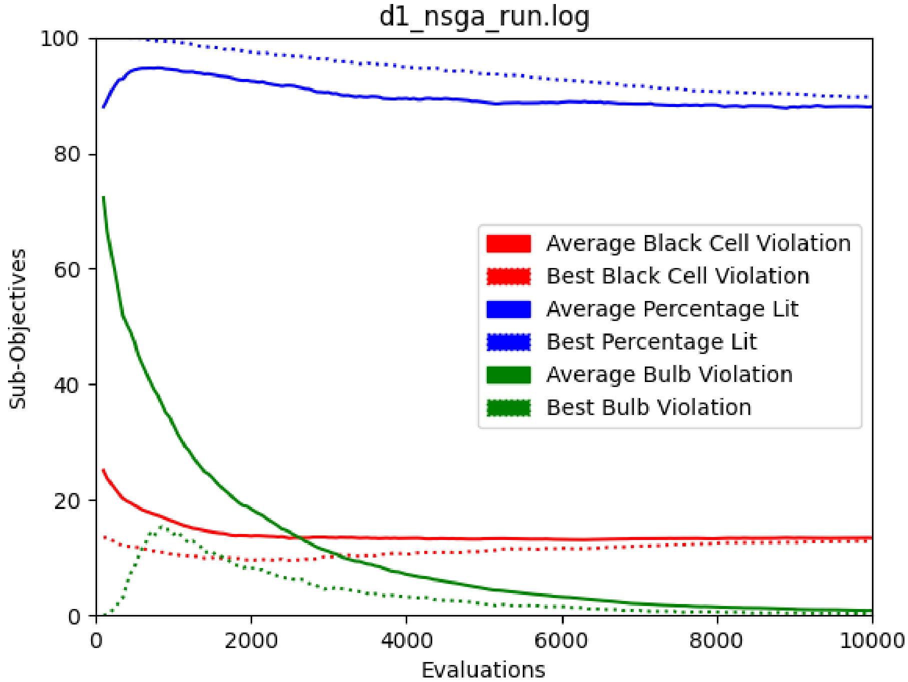

Figure 3: D1 with “NSGA” configuration

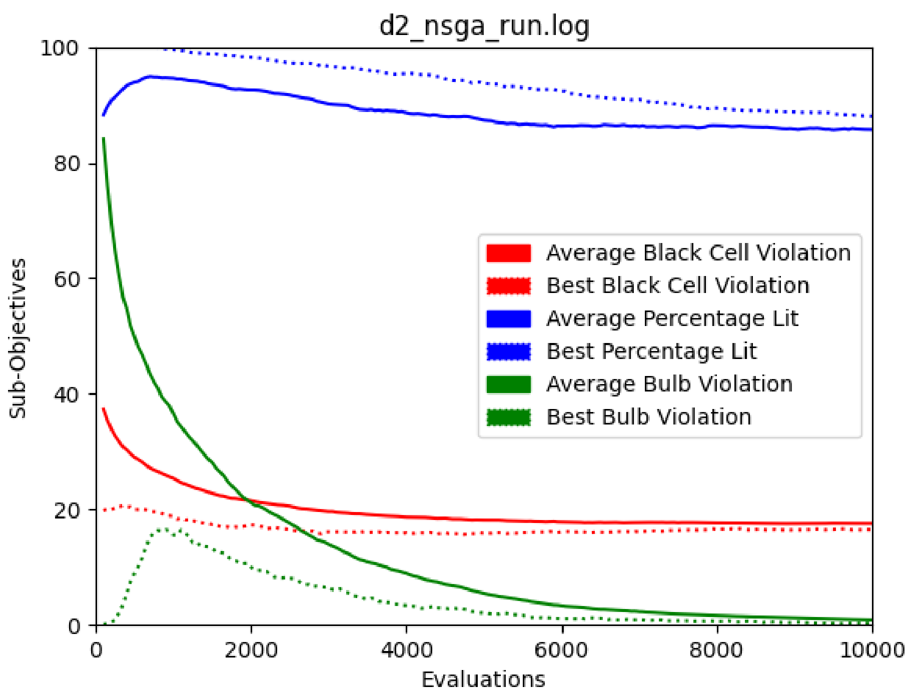

Figure 4: D2 with “NSGA” configuration

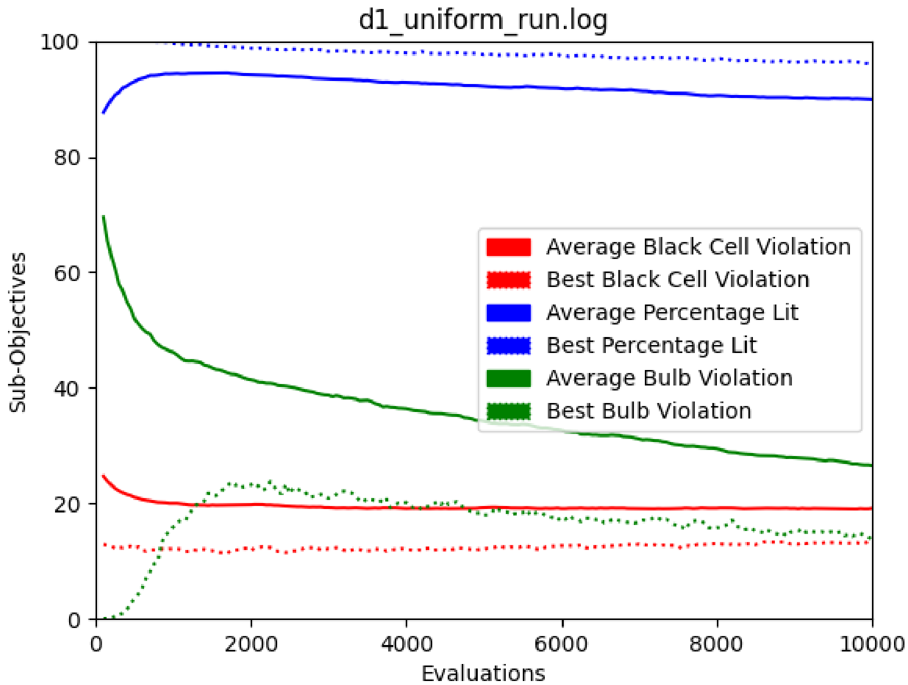

Figure 5: D1 with "Uniform" configuration

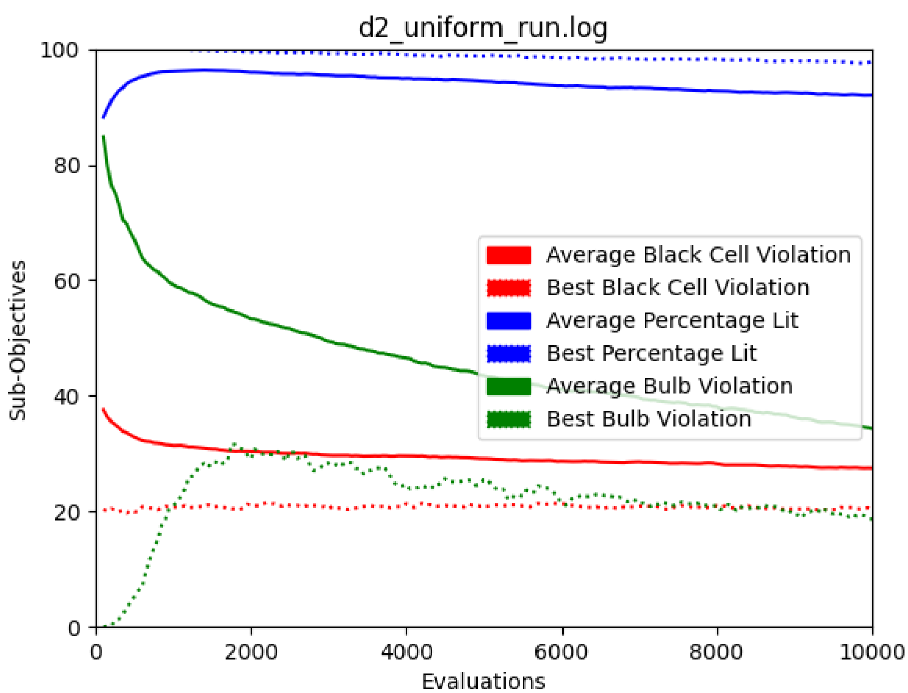

Figure 6: D2 with "Uniform" configuration

Default	NSGA
82	93
82	99
83	91
67	92
96	92
100	95
89	98
71	100
100	99
94	97
73	90
100	97
100	99
88	92
92	96
85	82
80	91
100	82
79	93
80	88
100	94
100	98
100	94
81	100
75	86
95	89
83	96
82	85
91	86
86	100
stdv.s	stdv.s
9.963381	5.335201

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	87.8	93.13333333
Variance	99.26896552	28.46436782
Observations	30	30
df	29	29
F	3.487481828	
P(F<=f) one-tail	0.000606703	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)
F > F Crit

equal variance

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	87.8	93.13333333
Variance	99.26896552	28.46436782
Observations	30	30
Pooled Variance	63.86666667	
Hypothesized Mean Difference	0	
df	58	
t Stat	-2.584682679	
P(T<=t) one-tail	0.006142485	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.012284971	
t Critical two-tail	2.001717484	

Figure 7: D1 Default Configuration vs NSGA Configuration

The comparison between the default configuration against the uniform configuration on problem D1 can be seen in Figure 8.

The comparison between the NSGA configuration against the uniform configuration on problem D1 can be seen in Figure 9.

D2

The comparison between the default configuration against the NSGA configuration on problem D2 can be seen in Figure 10.

The comparison between the default configuration against the uniform configuration on problem D2 can be seen in Figure 11.

The comparison between the NSGA configuration against the uniform configuration on problem D2 can be seen in Figure 12.

Fitness Sharing

For objective **Yellow 1** the author was asked to add an option for *fitness sharing* and investigate and report on how this addition affects performance. *Fitness sharing*, along with crowding, is a way to increase diversity in populations that may be suffering from low diversity. Using NSGA-II's rank based fitness one can run into issues with low diversity. This occurs when a large number of individuals with the same fitness, however, their genes are quite different. This result can happen in NSGA-II as one applies rank based on levels of non-domination. Fitness sharing aims to alleviate this issue by generating a new fitness for an individual based on how "close" it is to other individuals. The formula, as provided in the slides, is as follows:

Default	Uniform
82	96
82	84
83	93
67	93
96	92
100	96
89	100
71	100
100	97
94	100
73	99
100	91
100	99
88	100
92	100
85	100
80	94
100	95
79	88
80	93
100	92
100	100
100	84
81	87
75	98
95	84
83	100
82	100
91	96
86	93
stdv.s	stdv.s
9.963381	5.248317

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	87.8	94.8
Variance	99.26896552	27.54482759
Observations	30	30
df	29	29
F	3.603905859	
P(F<=f) one-tail	0.000456902	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)

F > F Crit

equal variance

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	87.8	94.8
Variance	99.26896552	27.54482759
Observations	30	30
Pooled Variance	63.40689655	
Hypothesized Mean Difference	0	
df	58	
t Stat	-3.404673112	
P(T<=t) one-tail	0.000603861	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.001207723	
t Critical two-tail	2.001717484	

Figure 8: D1 Default Configuration vs Uniform Configuration

NSGA	Uniform
93	96
99	84
91	93
92	93
92	92
95	96
98	100
100	100
99	97
97	100
90	99
97	91
99	99
92	100
96	100
82	100
91	94
82	95
93	88
88	93
94	92
98	100
94	84
100	87
86	98
89	84
96	100
85	100
86	96
100	93
stdv.s	stdv.s
5.335201	5.248317

F-Test Two-Sample for Variances

	Variable 1	Variable 2
Mean	93.13333333	94.8
Variance	28.46436782	27.54482759
Observations	30	30
df	29	29
F	1.033383408	
P(F<=f) one-tail	0.465073849	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)

F < F Crit

unequal variance

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	Variable 2
Mean	93.13333333	94.8
Variance	28.46436782	27.54482759
Observations	30	30
Hypothesized Mean Difference	0	
df	58	
t Stat	-1.21977495	
P(T<=t) one-tail	0.113743536	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.227487072	
t Critical two-tail	2.001717484	

Figure 9: D1 NSGA Configuration vs Uniform Configuration

Default	NSGA
79	88
74	100
72	92
84	80
100	89
99	100
77	93
80	90
81	89
100	99
76	94
79	100
88	82
100	93
86	99
85	94
75	90
74	91
77	93
80	92
100	86
100	93
72	86
74	91
73	91
77	100
80	87
100	87
72	89
100	91
stdv.s	stdv.s
10.64926742	5.196041818

F-Test Two-Sample for Variances

	Variable 1	Variable 2
Mean	83.8	91.63333333
Variance	113.4068966	26.99885057
Observations	30	30
df	29	29
F	4.200434246	
P(F<=f) one-tail	0.000114453	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)

F > F Crit

equal variance

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	Variable 2
Mean	83.8	91.63333333
Variance	113.4068966	26.99885057
Observations	30	30
Pooled Variance	70.20287356	
Hypothesized Mean Difference	0	
df	58	
t Stat	-3.620885506	
P(T<=t) one-tail	0.000309083	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.000618165	
t Critical two-tail	2.001717484	

Figure 10: D2 Default Configuration vs NSGA Configuration

Default	Uniform
79	84
74	95
72	100
84	81
100	99
99	76
77	87
80	87
81	93
100	100
76	100
79	98
88	77
100	88
86	98
85	100
75	88
74	85
77	100
80	83
100	97
100	100
72	93
74	84
73	84
77	87
80	94
100	81
72	76
100	99
stdv.s	stdv.s
10.64926742	8.16102315

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	83.8	90.46666667
Variance	113.4068966	66.60229885
Observations	30	30
df	29	29
F	1.70274748	
P(F<=f) one-tail	0.078882622	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)

F < F Crit

unequal variance

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	83.8	90.46666667
Variance	113.4068966	66.60229885
Observations	30	30
Hypothesized Mean Difference	0	
df	54	
t Stat	-2.721585754	
P(T<=t) one-tail	0.004363421	
t Critical one-tail	1.673564906	
P(T<=t) two-tail	0.008726841	
t Critical two-tail	2.004879288	

Figure 11: D2 Default Configuration vs Uniform Configuration

NSGA	Uniform
88	84
100	95
92	100
80	81
89	99
100	76
93	87
90	87
89	93
99	100
94	100
100	98
82	77
93	88
99	98
94	100
90	88
91	85
93	100
92	83
86	97
93	100
86	93
91	84
91	84
100	87
87	94
87	81
89	76
91	99
stdv.s	stdv.s
5.196041818	8.16102315

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	91.63333333	90.46666667
Variance	26.99885057	66.60229885
Observations	30	30
df	29	29
F	0.405374154	
P(F<=f) one-tail	0.008856444	
F Critical one-tail	0.537399965	

mean(Var1) > mean(Var2)

F < F Crit

equal variance

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	91.63333333	90.46666667
Variance	26.99885057	66.60229885
Observations	30	30
Pooled Variance	46.80057471	
Hypothesized Mean Difference	0	
df	58	
t Stat	0.660490881	
P(T<=t) one-tail	0.255776845	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.511553689	
t Critical two-tail	2.001717484	

Figure 12: D2 NSGA Configuration vs Uniform Configuration

$f'(i) = \frac{f(i)}{\sum_{j=1}^{\mu} sh(d(i,j))} \quad sh(d) = \begin{cases} 1 - d/\sigma & \text{if } d < \sigma \\ 0 & \text{otherwise} \end{cases}$

σ is a tunable parameter that correlates to how close an individual should be to another. Fitness sharing should be calculated right before individual selection occurs for the next generation.

The author uses the Euclidean distance between the three sub-objectives as his distance metrics. Which can be calculated as follows: $d(p,q) = \sqrt{\sum_{i=1}^{j} (p_i^2 - q_i^2)^2}$

In this assignment, j is three as there are three sub-objectives.

The author uses the popular Python package SciPy to calculate the Euclidean distance. Additionally, the fitness sharing formula was tuned as sometimes the one above one can produce results that move beyond the existing rank the individual started at which is not desirable. Therefore the function has been modified to the following:

$f'(i) = \frac{0.5}{1+\sum_{j=1}^{\mu} sh(d(i,j))} \quad sh(d) = \begin{cases} 1 - d/\sigma & \text{if } d < \sigma \\ 0 & \text{otherwise} \end{cases}$

For the modified formula `1` is added to ensure the fitness doesn't become a level higher than it was when it started and `0.5` can be any values [0,1]. The following Python snippet shows how the author performs fitness sharing.

```
def fitness_sharing(self, current, population):
    sh_vals = []
    for individual in population:

        # skip current individual from population
        if current.name == individual.name:
            continue

        # generate coordinates used to calculate euclidean distance
        a = (current.lit, current.black_cell_violations,
                current.bulb_violations)
        b = (individual.lit, individual.black_cell_violations,
                individual.bulb_violations)

        distance = scipy.spatial.distance.euclidean(a, b)
        sh_vals.append(self.sh(distance))

    sum_sh = sum(sh_vals)
    fitness = 0.5 / ((sum_sh if sum_sh else 1) + 1)
    return current.fitness + fitness
```

Results

For both D1 and D2, the same configuration was used which was an adapted version of the "Default" configuration. The only difference was that fitness sharing was used.

This configuration took ~4851 seconds or ~81 minutes to run for D1 and ~3706 seconds or ~62 minutes for D2.

One can see in Figure 13 the results of running the fitness sharing configuration on problem D1.

One can see in Figure 14 the results of running the fitness sharing configuration on problem D2.

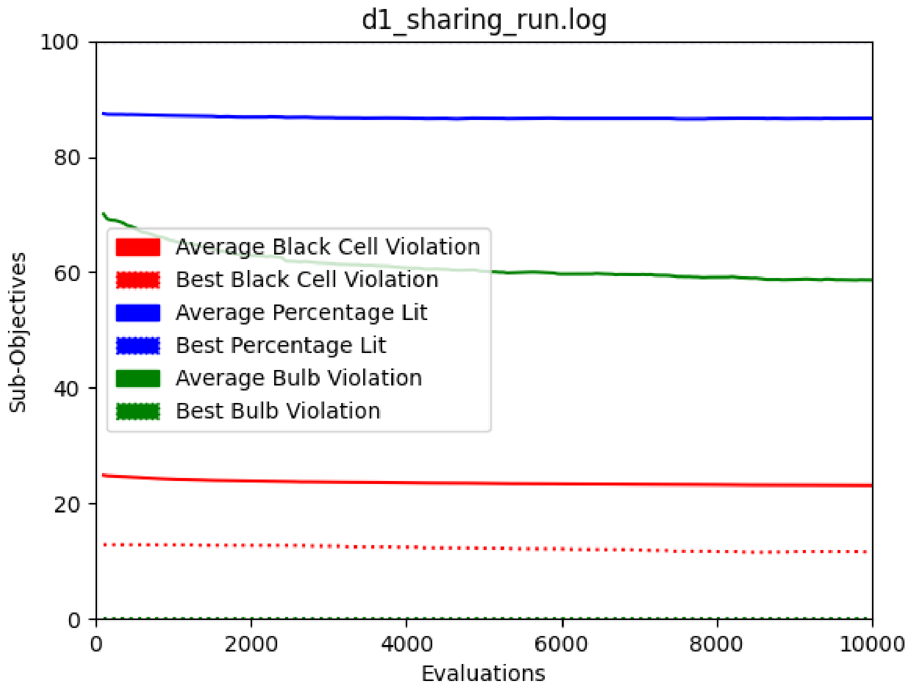

Figure 13: D1 with Fitness Sharing

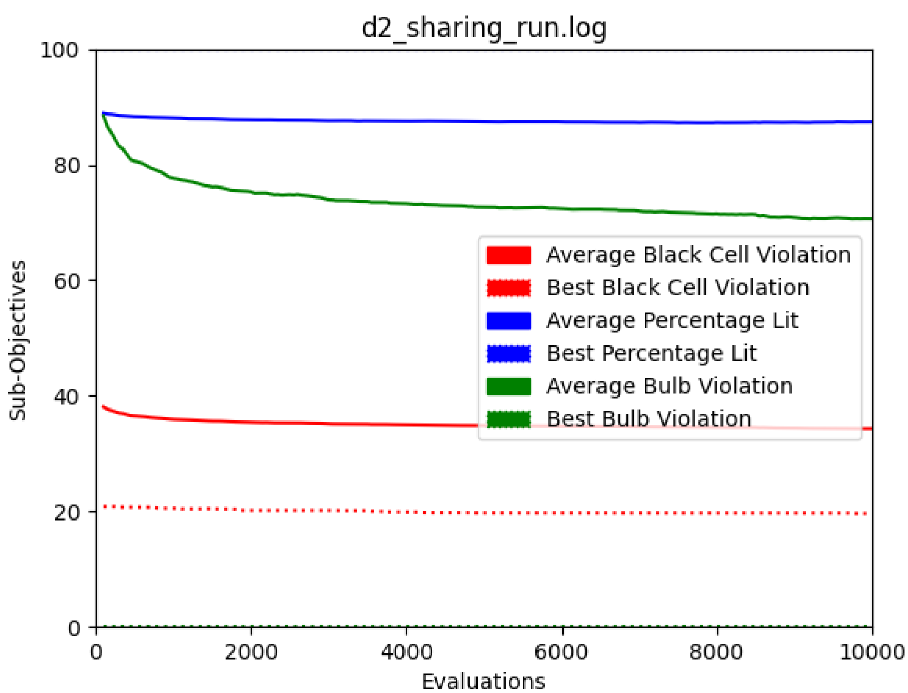

Figure 14: D2 with Fitness Sharing

The comparison between the default configuration against the fitness sharing configuration on problem D1 can be seen in Figure 15.

Default	Fitness Sharing	Fitness Sharing Floor
82	94.08388045	94
82	85.05812582	85
83	90.04695496	90
67	91.05290467	91
96	74.04907655	74
100	73.05822563	73
89	85.0450966	85
71	83.04090965	83
100	74.03151365	74
94	96.04503516	96
73	79.04441273	79
100	93.04975561	93
100	100.1806847	100
88	100.0968205	100
92	100.0328547	100
85	100.0324874	100
80	78.17142159	78
100	72.03079119	72
79	81.04722017	81
80	100.0419777	100
100	79.04091186	79
100	80.03751933	80
100	100.3016449	100
81	76.03929781	76
75	100.1798729	100
95	84.05943423	84
83	75.04937246	75
82	79.07750852	79
91	94.05458376	94
86	75.06150115	75
stdv.s	stdv.s	
9.963381	10.14064649	

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	87.8	86.33333333
Variance	99.26896552	102.3678161
Observations	30	30
df	29	29
F	0.969728273	
P(F<=f) one-tail	0.467300935	
F Critical one-tail	0.537399965	

mean(Var1) > mean(Var2)
F > F Crit

unequal variance

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	87.8	86.40472654
Variance	99.26896552	102.8327113
Observations	30	30
Hypothesized Mean Difference	0	
df	58	
t Stat	0.537569971	
P(T<=t) one-tail	0.296465319	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.592930638	
t Critical two-tail	2.001717484	

Figure 15: D1 Default Configuration vs Fitness Sharing Configuration

The comparison between the default configuration against the fitness sharing configuration on problem D2 can be seen in Figure 16.

Crowding

For objective **Yellow 2** the author was asked to add an option for *crowding* and investigating and reporting on how this addition affects performance. As stated earlier crowding is another way of increasing diversity in a population. For the author crowding was more complicated to understand and to implement than fitness sharing. For the crowding implementation, NSGA-II's crowding was used as a reference. Which is defined as the following

```
l = |I|
for each i, set I[i].distance = 0
for each objective m
    I = sort(I, m)
    I[1].distance = I[l].distance = \inf
    for i = 2 (l - 1)
        I[i].distance += (I[i+1].m - I[i-1].m) / (max(m) - min(m))
```

Essentially, for each objective one will sort the population-based on it, in ascending order. After sorting one will pick out the boundaries and set their distance to infinity. Following this, every other individual's distance will be calculated based on the proximity of the surrounding individuals.

For the author's implementation, this was modified, mainly after calculating all the distances they were normalized between 0 and 1. Values that were marked infinity were normalized between [0.5, 1) other values were normalized between [0, 0.5]. Following the normalization of the distance values the

Default	Fitness Sharing	Fitness Sharing Floor
79	76.06721987	76
74	100.0470606	100
72	72.04985092	72
84	97.07252896	97
100	75.03657195	75
99	100.0577599	100
77	100.0785853	100
80	79.03346143	79
81	81.06394994	81
100	100.0630036	100
76	73.05536636	73
79	78.04243111	78
88	76.08677764	76
100	76.03911789	76
86	100.0655396	100
85	76.05876774	76
75	90.14458085	90
74	74.06525762	74
77	72.05069011	72
80	83.05559502	83
100	100.1005109	100
100	75.04844645	75
72	97.03545282	97
74	80.05389345	80
73	82.06699268	82
77	79.08196197	79
80	80.04141286	80
100	99.05924007	99
72	76.05984915	76
100	79.04547495	79
stdv.s	stdv.s	
10.64927	10.64472127	

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	83.8	84.16666667
Variance	113.4068966	113.1781609
Observations	30	30
df	29	29
F	1.002021023	
P(F<=f) one-tail	0.497849861	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)

F < F Crit

unequal variance

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	83.8	84.22757839
Variance	113.4068966	113.310091
Observations	30	30
Hypothesized Mean Difference	0	
df	58	
t Stat	-0.155537223	
P(T<=t) one-tail	0.438468935	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.876937871	
t Critical two-tail	2.001717484	

Figure 16: D2 Default Configuration vs Fitness Sharing Configuration

individual's original fitness was increased by this normalized distance metric. The implementation of crowding can be seen in the following code snippet. The `update_fitness` function simply adds new values to the existing fitness rank.

```
def crowding(self, population):
    fronts = self.split_on_pareto_front(population)
    out_distances = {}
    for individuals in fronts.values():
        distance = {individual.name: 0 for individual in individuals}

        distance = self.crowding_lit(individuals, distance)
        distance = self.crowding_black_cell(individuals, distance)
        distance = self.crowding_bulb_violations(individuals, distance)

        out_distances = self.normalize_data(distance, out_distances)

    return self.update_fitness(population, out_distances)
```

All the sub-objective had a different function called, however the logic of the overlying function was the same. The author was unsure how in Python to achieve a dynamic way to call the same code but to pass a different class instance variable to be used.

```
def crowding_subobjective(individuals, distance):
    max_val = max([
        individual.subobjective for individual in individuals])
    min_val = min([
        individual.subobjective for individual in individuals])
```

```python
    sorted_pop = sorted(individuals, reverse=False,
                                    key=lambda ind: ind.subobjective)

    distance[sorted_pop[0].name] = 1000
    distance[sorted_pop[-1].name] = 1000

    for index in range(2, len(sorted_pop) - 1):
        current = sorted_pop[index]
        next_individual = sorted_pop[index + 1]
        prior_individual = sorted_pop[index - 1]
        try:
            distance[current.name] += \
                (next_individual.subobjective - prior_individual.subobjective) / \
                (max_val - min_val)
        except ZeroDivisionError:
            distance[current.name] += 0
    return distance
```

The `normalize_data` function is implemented as follows. First, it pulls out the infinite distance individuals as they should be normalized in a different range than the non-infinite ones.

```python
def normalize_data(distance, out_distances):
    """ normalizes data between 0 and 1 / val """

    def normalize(data, val):
        # actual function that normalizes data
        # called twice as we want to normalize the non inf values between 0 and some value
        # infinite between some value and .99
        if data:
            return {name: value / (val * max(data.values())) for name, value in data.items()}
        return {}

    # split distances between inf and non inf
    no_inf_distances = {name: value for name,
                        value in distance.items() if value < 1000}
    inf_distances = {name: value for name,
                        value in distance.items() if value >= 1000}

    normalized_no_inf = normalize(no_inf_distances, 2)
    normalized_inf = normalize(inf_distances, 1.0101010101010102)
    out_distances = {**out_distances, **normalized_no_inf, **normalized_inf}
    return out_distances
```

Results

For both D1 and D2 the same configuration was used as for fitness sharing; except crowding was used instead.

This configuration took ~2394 seconds or ~40 minutes to run for D1 and ~2426 seconds or ~40 minutes for D2.

One can see in Figure 17 the results of running the crowding configuration on problem D1.

One can see in Figure 18 the results of running the crowding configuration on problem D1.

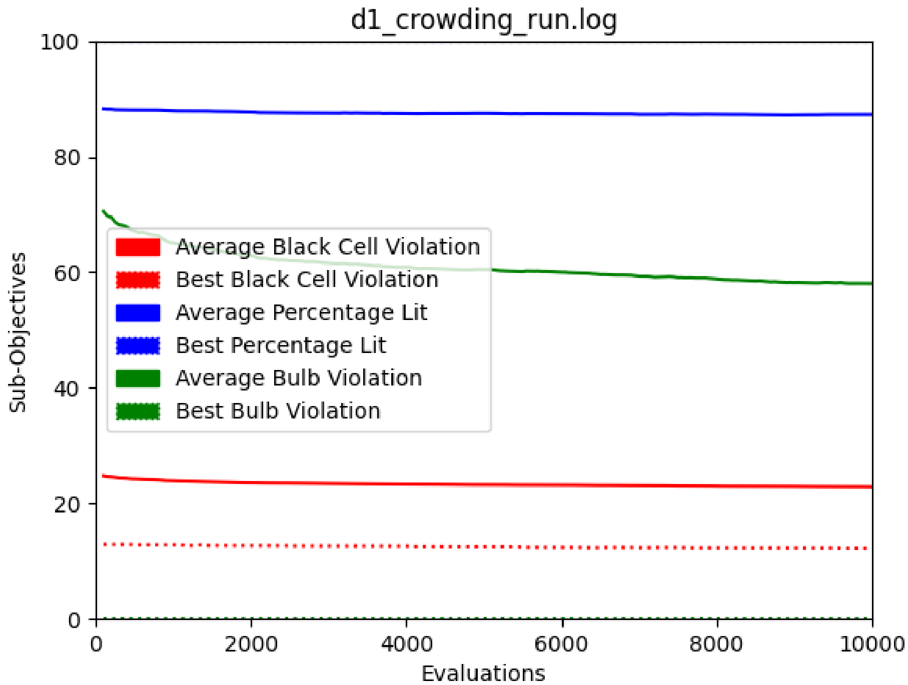

Figure 17: D1 with Crowding

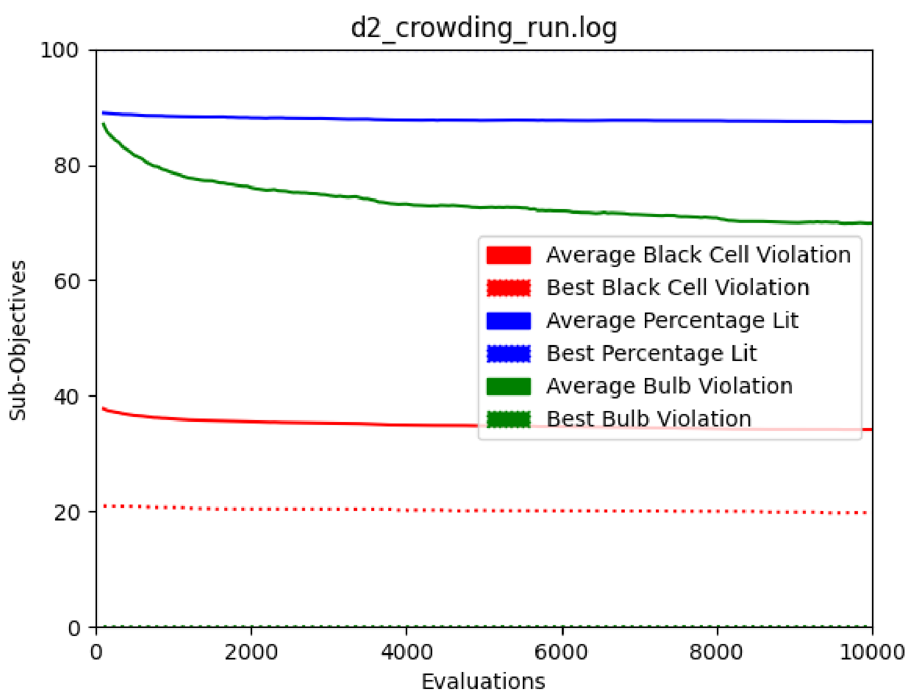

Figure 18: D2 with Crowding

The comparison between the default configuration against the crowding configuration on problem D1 can be seen in Figure 19.

Default	Crowding	Crowding - Floor
82	100.99	100
82	79.99	79
83	100.99	100
67	81.9896408	81
96	100.99	100
100	99.99	99
89	92.98937015	92
71	74.98952427	74
100	89.99	89
94	100.99	100
73	79.98968895	79
100	100.99	100
100	100.99	100
88	100.99	100
92	90.99	90
85	95.5	95
80	83.99	83
100	81.10425553	81
79	100.99	100
80	99.99	99
100	100.99	100
100	100.99	100
100	73.99	73
81	82.99	82
75	74.39517287	74
95	95.5	95
83	99.99	99
82	100.128229	100
91	100.99	100
86	75.99	75
stdv.s	stdv.s	
9.963381229	10.18603516	

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	87.8	91.3
Variance	99.26896552	103.1137931
Observations	30	30
df	29	29
F	0.962712771	
P(F<=f) one-tail	0.459601589	
F Critical one-tail	0.537399965	

mean(Var1) < mean(Var2)
F > F Crit

equal variance

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	87.8	92.17919605
Variance	99.26896552	103.7553123
Observations	30	30
Pooled Variance	101.5121389	
Hypothesized Mean Difference	0	
df	58	
t Stat	-1.683375603	
P(T<=t) one-tail	0.048839062	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.097678124	
t Critical two-tail	2.001717484	

Figure 19: D1 Default Configuration vs Crowding Configuration

The comparison between the default configuration against the crowding configuration on problem D2 can be seen in Figure 20.

Minimizing Bulbs

For the **Red 1** objective the author was asked to add a fourth objective; minimizing the number of bulbs placed. Once added the author would need to investigate and report on how the performance and behavior were impacted by having four objectives rather than three. Due to the modularity of the author's code, this was relatively easy to implement. First, a new instance variable had to be added to the `Individual` class called `bulbs` which is the number of bulbs placed for those individuals.

1. Percent of tiles lit
2. Fitness
3. Bulb locations
4. Number of black cell violations
5. Number of bulb intersections
6. A unique name
7. Number of bulbs

After that, the `<` dunder function needed to be updated to include the bulb comparison.

```
def __lt__(self, other):
    a = (self.lit <= other.lit) and \
        (self.bulb_violations >= other.bulb_violations) and \
        (self.black_cell_violations >= other.black_cell_violations) and \
```

Default	Crowding	Crowding Floor
79	98.33297	98
74	72.99	72
72	85.5	85
84	77.99	77
100	100.99	100
99	100.99	100
77	88.99	88
80	88.99	88
81	75.98973	75
100	68.25554	68
76	73.07738	73
79	86.99	86
88	73.98979	73
100	67.99	67
86	83.99	83
85	83.99	83
75	93.99	93
74	95.99	95
77	90.99	90
80	100.99	100
100	77.20493	77
100	80.99	80
72	82.99	82
74	98.99	98
73	100.99	100
77	82.99	82
80	78.09819	78
100	87.99	87
72	70.99	70
100	84.2315	84
stdv.s	stdv.s	
10.64927	10.36641	

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	83.8	84.4
Variance	113.4068966	105.6275862
Observations	30	30
df	29	29
F	1.073648472	
P(F<=f) one-tail	0.424779893	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)
F < F Crit

unequal variance

t-Test: Two-Sample Assuming Unequal Variances

	Variable 1	*Variable 2*
Mean	83.8	85.21566774
Variance	113.4068966	107.4623737
Observations	30	30
Hypothesized Mean Difference	0	
df	58	
t Stat	-0.521740216	
P(T<=t) one-tail	0.301919015	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	0.60383803	
t Critical two-tail	2.001717484	

Figure 20: D2 Default Configuration vs Crowding Configuration

```python
        (self.bulbs <= other.bulbs)
    b = (self.lit < other.lit) or \
        (self.bulb_violations > other.bulb_violations) or \
        (self.black_cell_violations > other.black_cell_violations) or \
        (self.bulbs < other.bulbs)
    return a and b
```

Additionally, when the individuals are instantiated bulbs would need to be passed to the constructor.

This can be achieved easily by using the *kwargs and args* pattern. In Python the * operator is used to unpack variables, ** is used to unpack dictionaries. When using * in function calls it unpacks position parameters and ** is used to unpack named parameters. Therefore the two following code snippets achieve the same goal.

```python
new_individual = Individual(lit=fitness,\
                            solution=solution,\
                            black_cell_violations=black_cells,\
                            bulb_violations= intersections)
```

```python
kwargs = {'lit': fitness, \
          'solution': solution,\
          'black_cell_violations': black_cells,\
          'bulb_violations': intersections}
new_individual = Individual(**kwargs)
```

The following snippet is how the author allowed the `bulb` variable to be passed to the constructor depending on if the EA was configured for four objectives or three.

```
fitness, solution, black_cells, intersections = self.initialization_selection(
    board)
kwargs = {'lit': fitness, 'solution': solution,
            'black_cell_violations': black_cells, 'bulb_violations': intersections}
if self.bulb_objective:
    kwargs['bulbs'] = len(solution)

initialize_population.append(Individual(**kwargs))
```

Results

Running with a fourth objective used the default configuration except there was an extra line specifying to use the fourth objective.

This configuration took ~2433 seconds or ~41 minutes to run for D1 and ~2524 seconds or ~42 minutes for D2.

One can see in Figure 21 the results of running the bulb minimization configuration on problem D1.

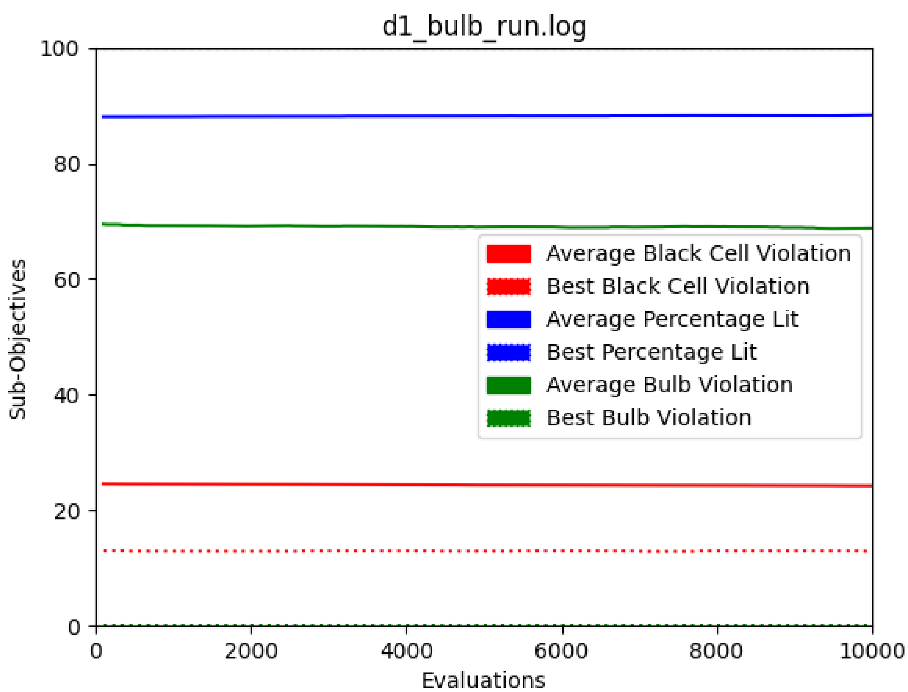

Figure 21: D1 with Fitness Sharing

One can see in Figure 22 the results of running the bulb minimization configuration on problem D2.

The comparison between the default configuration against the bulb minimization configuration on problem D1 can be seen in Figure 23.

The comparison between the default configuration against the bulb minimization configuration on problem D2 can be seen in Figure 24.

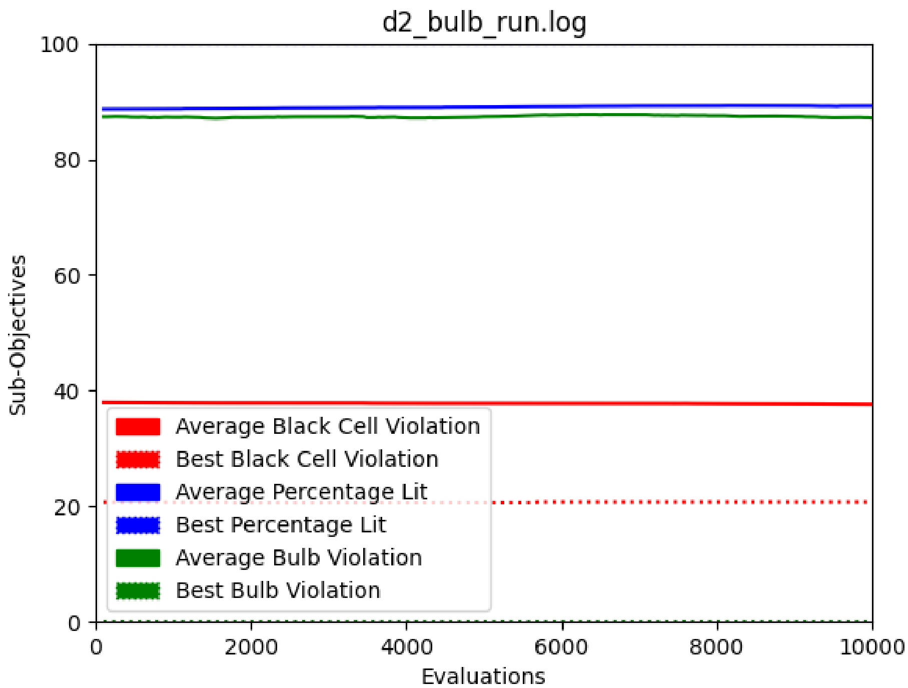

Figure 22: D2 with Fitness Sharing

Conclusion

For this assignment, the author was asked to implement a Multi-Objective EA with the objectives being: percentage of lit cells, the number of black cell constraint violations, and the number of bulbs placed in lit cells. Additionally, it was requested to implement fitness sharing and crowding for the MOEA. For bonus points, one could also add a fourth objective; which was minimizing the number of bulbs placed. Thankfully, due to the modularity of the existing code base, none of these requirements were too difficult to implement. The most difficult portion of this assignment was probably understanding the math behind the fitness sharing algorithms. Due to the long run time of the experiments the author ran out of time compiling the report unfortunately cutting some of the statistical analysis short.

Default	Minimize Bulbs
82	100
82	100
83	100
67	95
96	94
100	100
89	94
71	100
100	100
94	100
73	96
100	98
100	100
88	97
92	100
85	90
80	93
100	100
79	100
80	100
100	100
100	100
100	100
81	100
75	100
95	100
83	100
82	100
91	100
86	100
stdv.s	stdv.s
9.963381	2.712466

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	87.8	98.56666667
Variance	99.26896552	7.357471264
Observations	30	30
df	29	29
F	13.49226683	
P(F<=f) one-tail	2.33822E-10	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)

F > F Crit

equal variance

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	87.8	98.56666667
Variance	99.26896552	7.357471264
Observations	30	30
Pooled Variance	53.31321839	
Hypothesized Mean Difference	0	
df	58	
t Stat	-5.71096432	
P(T<=t) one-tail	2.03118E-07	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	4.06236E-07	
t Critical two-tail	2.001717484	

Figure 23: D1 Default Configuration vs Minimize Bulb Configuration

Default	Minimize Bulbs
79	100
74	100
72	100
84	100
100	100
99	100
77	100
80	100
81	96
100	100
76	100
79	100
88	100
100	100
86	100
85	99
75	100
74	100
77	97
80	100
100	100
100	89
72	97
74	100
73	100
77	92
80	100
100	92
72	100
100	100
stdv.s	stdv.s
10.64926742	2.851899951

F-Test Two-Sample for Variances

	Variable 1	*Variable 2*
Mean	83.8	98.73333333
Variance	113.4068966	8.133333333
Observations	30	30
df	29	29
F	13.94347089	
P(F<=f) one-tail	1.5416E-10	
F Critical one-tail	1.860811435	

mean(Var1) < mean(Var2)
F > F Crit

equal variance

t-Test: Two-Sample Assuming Equal Variances

	Variable 1	*Variable 2*
Mean	83.8	98.73333333
Variance	113.4068966	8.133333333
Observations	30	30
Pooled Variance	60.77011494	
Hypothesized Mean Difference	0	
df	58	
t Stat	-7.419204807	
P(T<=t) one-tail	2.89355E-10	
t Critical one-tail	1.671552762	
P(T<=t) two-tail	5.78709E-10	
t Critical two-tail	2.001717484	

Figure 24: D2 Default Configuration vs Minimize Bulb Configuration