SBSE Group Project Group 3

Assignment 2

Mitchell Follett and Tony Dinh

**Task 1:**

**Mitch Contribution:**

My contributions included writing the calculate charge function for task 2 part 2. As well as this I constructed the algorithm in java for part 3 and ran the tests on them, as well as the report portion of task 3.

**Tony Contirbution:**

My contributions for this assignment are all of part 1 for Task 2 and part 1 for Task 4. As well as writing java algorithm for the MOEA frame and printing out results.

**Task 2:**

**Part 1:**

The aim of the calculation below is to determine the “average charge used per hour” in milli-Ampere(mA) per hour(h). In the data sheet nexus 6 we were given 8 different colours, for this specific calculation we only took into consideration the colour red, blue and green. The first step we took was to calculate the average charge consumption in (nA). This was done by averaging the column charged\_used\_accumulated of each respective colour and for each specify sample size ranging from 10, 100, 1000 and the full sample duration of the experiment. For red and blue this was 7208 samples and for green was 7207 samples. We then subtracted the average charge consumption with the average charge consumption of the colour black. This was done similar to the colours red, blue and green. We then calculated the time duration of the experiment for each sample size of each colour. We then divided the time duration by 3600000 to change the duration from ms to hours(h). The all average charge consumption of each respective colour subtracted with the average charge consumption black was then divided by the time duration in hours calculated to obtained average charge used per hour(mAh). We can simplify this in a simple to understand equation.

Let’s denote:

10, 100, 1000, 7207/7208

The equation used will be

We will then use this equation for all sample size steps of 10, 100, 1000 and 7208 for red and blue and 7207 for green. Fig 1 below is the results we obtained using this equation for each colours and sample sizes.

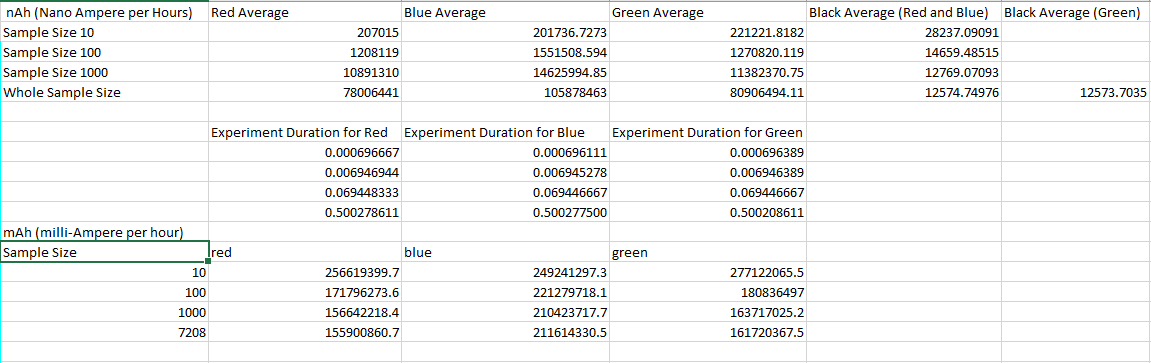
Results:

Fig 1. Table of results obtained

Fig 2 Graph of Results

Looking at the graph that we plotted we can see that initially for the sample size of 10 the color green consumes more mA compared to red and blue per hour and red uses slightly more mA compared to blue. Green initially consumes more mA per hour than red but over time the amount of mA required to continue using green reduces signficanlty more than red, to the point that green towards the end of the experiment green only slightly uses more mA per hour than red. Blue initial consumes less mA per hour compared to red and green but as the experiment continues we can start to see that the colour blue starts to consume significantly more mA per hour when compared to the colour red and green. The reason for the phenemon is due to the difference of wavelengths, blue is considered to have a shorter wavelength when compared to red and green. As such requires more energy to produce the same amount of brightness as red and green would require to use.

**Part 2**

The “calculateChargeConsumptionPerPixel” function works as follows:

A screen shot of a computer program

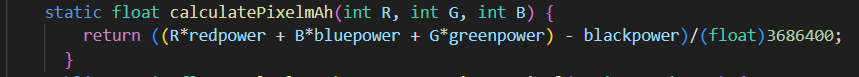
Description automatically generated

The function takes in a path string, this points to the location of the file to be calculated, firstly the input string is converted into a “File” and then imageIO.read is used to read this file into a BufferedImage object. The width and height is taken from this image, then used in a nested for loop to gather pixel data for each pixel.

A screen shot of a computer code

Description automatically generated

This is done using bitwise operations and the image.getRGB function, these values are carried through and used in our custom calculatePixelmAh, and accumulated into the totalpower variable, which is then returned from the function.



The calculatePixelmAh function takes the RGB values and uses the function from task 2 part 2 to give a power per pixel value in mAh.

**Task3:**

**Random seeding**

As it sounds, in random seeding we pick random numbers for each of the RGB values, and if the output is valid we check for power consumption, picking the best power consumption of the sample size.

**Hill climbing**

In hill climbing we generate “neighboring” solutions to the original. In this case we used +/- 50 on each of the rgb values, for a total of 6 neighbors. Then we select the best consumption from among the neighbors and repeat. As can be seen from the results below, this falls short when approaching smaller “peaks”. In this case when run more time, the result was worse, but it got to its final value on the 12th run.

**Genetic**

In the genetic algorithm we generate an original “population”, this was done using 5 randomly generated colour schemes, from these, each pair generates “offspring”, which was an average of each of the values in its scheme, from these the best performing 5 are taken and the process is repeated. Every repeat, it has a chance to “mutate”. These mutation affect one colour and can be from +150 to -150. The mutation make it so that eventually the mutations will improve the score, and from the genetics from the “pool” will be improved. As can be seen from the below data, this was by far the most reliable algorithm to achieve results.

**Random seeding data**

**10 runs**

Change in best value: [24759, 24759, 24759, 24759, 14770, 14770, 14770, 14770, 14770, 14770]

Best value after ten runs: 14770.255

A screenshot of a computer

Description automatically generated

**100 runs**

Change in best value: [24863, 24863, 24863, 24863, 20902, 20902, 17963, 17963, 17963, 17963, 17963, 17963, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 16543, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094, 13094]

Best value after 100 runs: 13094.307

A screenshot of a computer

Description automatically generated

**Hill climber data:**  
Best value after 10 runs: 10188.146  
Change in best value: [25028, 21527, 17428, 13876, 11441, 9202, 6836, 5328, 5323, 5323]A red square with a black background

Description automatically generated

Best value after 100 runs: 18493.9  
Change in best value: [25028, 21527, 17428, 13876, 11441, 9202, 6836, 5328, 5323, 5323]

A screenshot of a computer

Description automatically generated

**Genetic data:**

Best value after ten runs: 13464

Change in best value: [26969, 24760, 22660, 20979, 18708, 18494, 18494, 18494, 18494, 18494, 18494, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493, 18493]

A screenshot of a computer

Description automatically generated

Best value after 100 runs: 6106

Change in best value: [12282, 12282, 12282, 10015, 10015, 10015, 10015, 10015, 9061, 9061, 8200, 8200, 8200, 8200, 7961, 7961, 7954, 7832, 7832, 7832, 7832, 7823, 7823, 7823, 7823, 7823, 7823, 7823, 7823, 7823, 7823, 7816, 7816, 7273, 7273, 7273, 7273, 7273, 7272, 7272, 7272, 7272, 7272, 7272, 7272, 7272, 7272, 7272, 7272, 7272, 7266, 7266, 7266, 7266, 7118, 7118, 7118, 7117, 7117, 7117, 7112, 7112, 7112, 7112, 7112, 7112, 7111, 7111, 7111, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106, 6106]

A screenshot of a computer

Description automatically generated

**Task 4:**

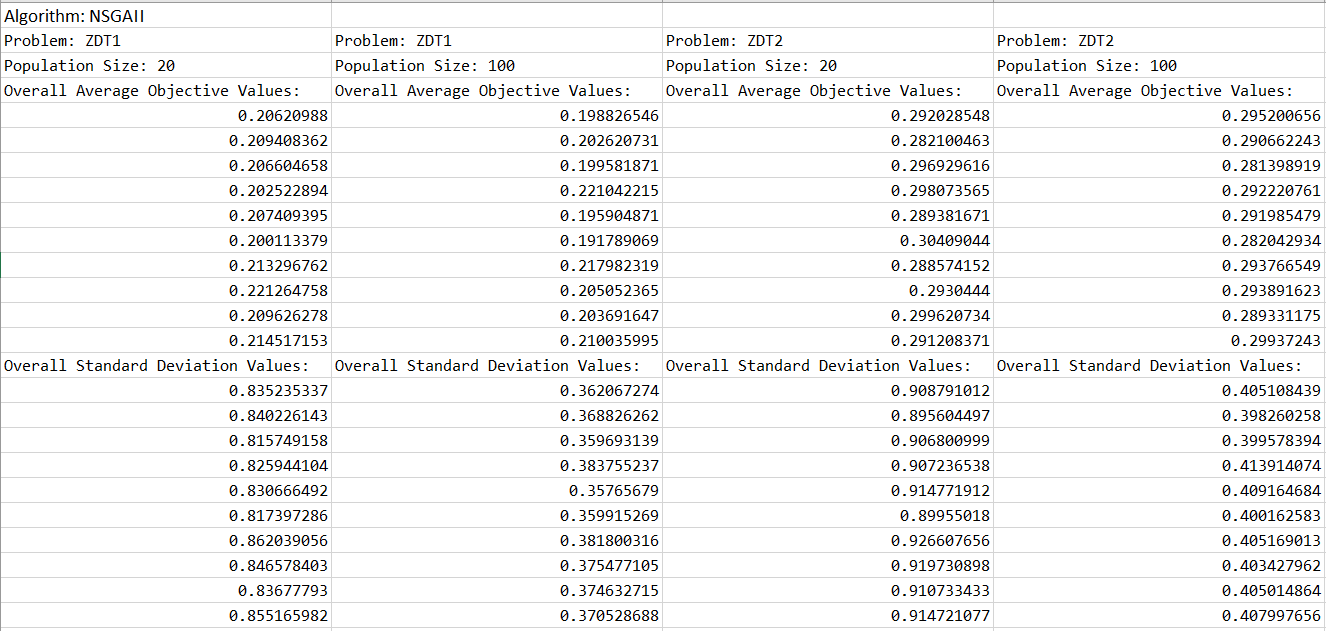


Fig 3. Results for NSGAII Algorithm for Problems ZDT1 and ZDT2

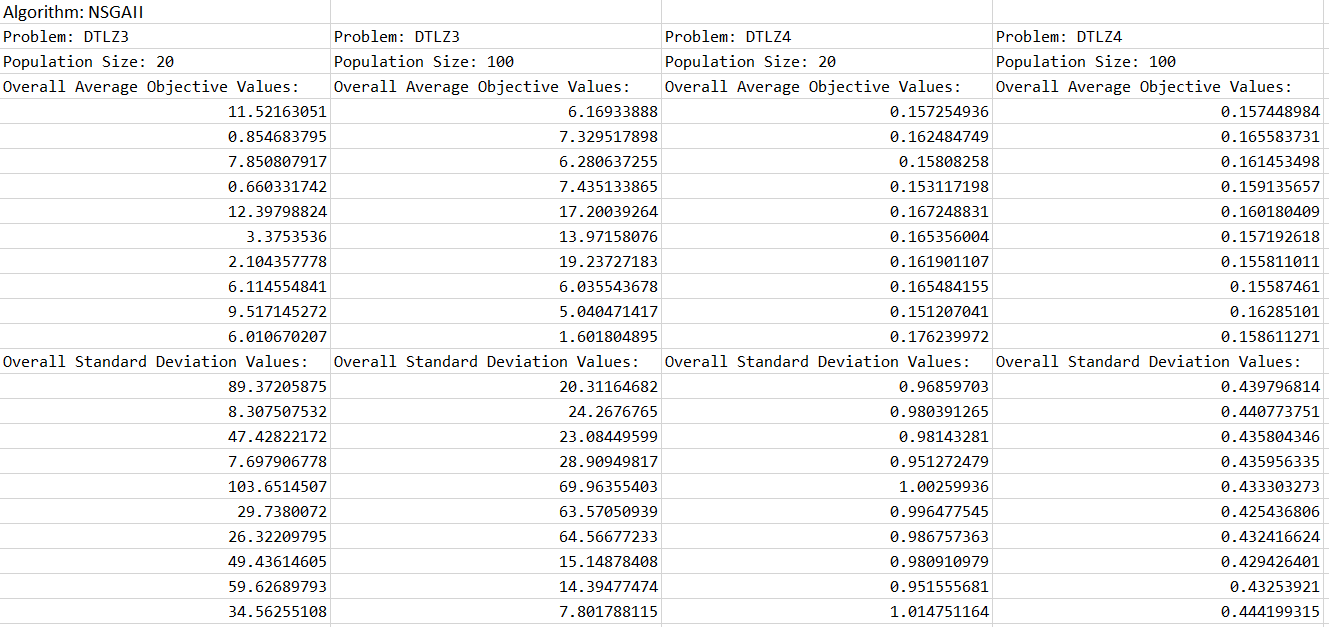


Fig 4. Results for NSGAII Algorithm for Problems DTLZ3 and DTLZ4

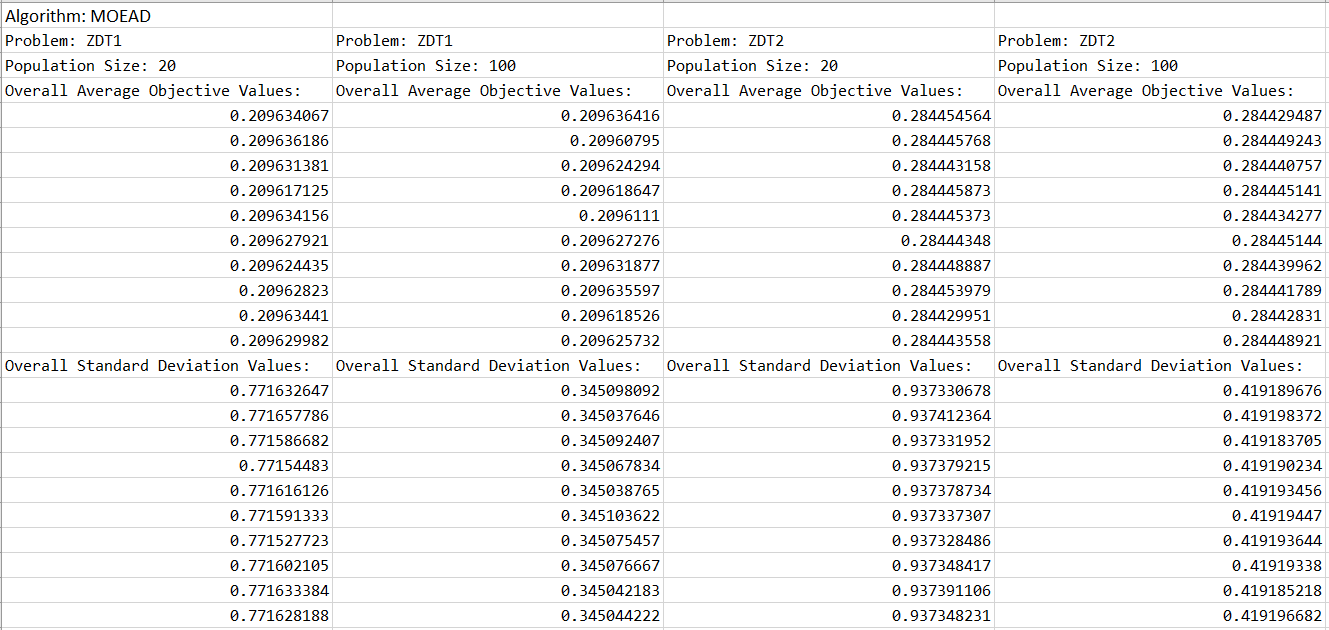


Fig 5. Results for MOEAD Algorithm for Problems ZDT1 and ZDT2

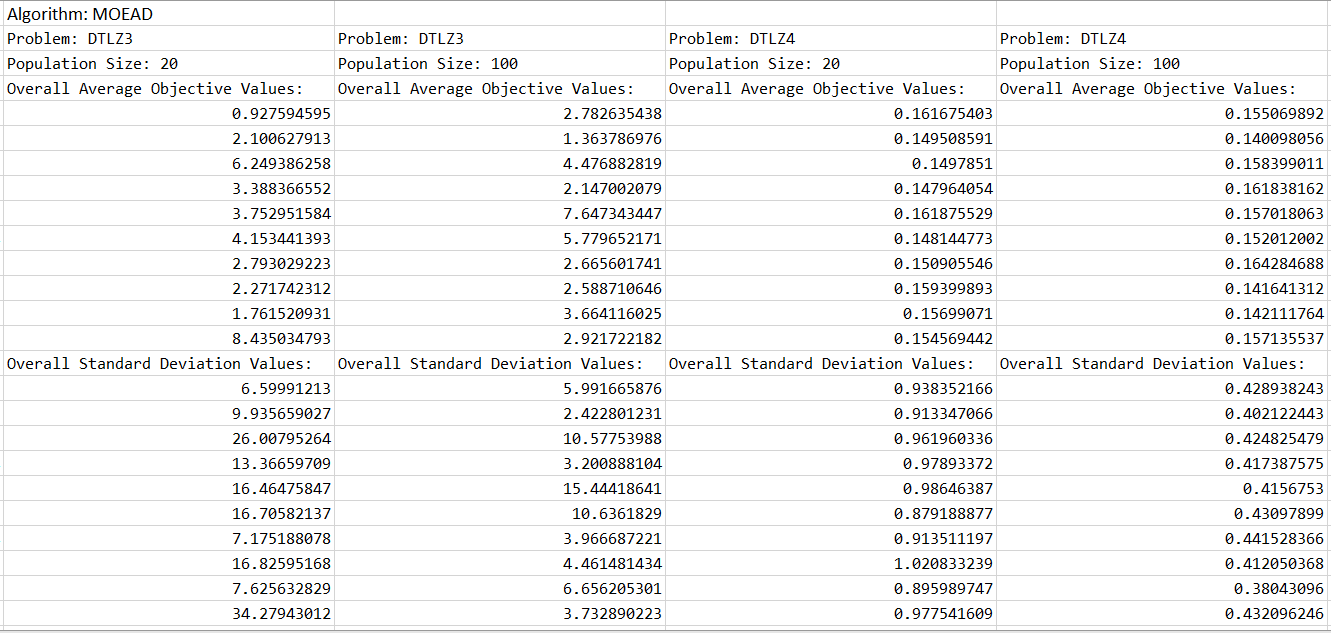


Fig 6. Results for MOEAD Algorithm for Problems DTLZ3 and DTLZ4

Fig 7. Average Objective value of ZDT1 vs ZDT2

Fig 8. Standard Deviation of ZDT1 vs ZDT2

Fig 8. Average Objective value of DTLZ3 vs DTLZ4

Fig 8. Standard Deviation of DTLZ3 vs DTLZ4

A lot of the previous code based was reused, such as the use of the Algorithm NSGAII and MOEAD. Furthermore, we reused the problem ZDT1, ZDT2, DTLZ3 and DTLZ4. New code that we had to implement was the calculation to determine the average objective value of each problem. We also had to implement our own function to determine the standard deviation of each problem.

**Part 2:**