Liam Devlin

Multi-Objective Optimisation Report

Multi-objective optimisation is a method of performing genetic computations with more than one desired goal. The nature of MOEA (Multi-Objective Evolutionary Algorithms) means that no one 'fittest' solution can be found as with a single objective solution. Instead a group of candidates are presented along an optimised 'front'. With this knowledge we were tasked to provide a solution to the Next Release Problem (NRP), finding a permutation of requirements that simultaneously pleases the most customers, for the lowest cost possible.

Chapter 1 : Overview of the problem:

Our task was to find a permutation of features which would simultaneously please the most amount of customers for the lowest cost possible.The data with which we used to perform this calculation took the form of a simple txt file containing a list of requirements and their associated costs, as well as a list of "customers" with one customer being comprised of; their financial investment, the customers number of requirements and finally the list of their requirements. A section describing the relationship between requirements also existed but was disregarded for the purpose of this task.

The requirements in this scenario are a simple array, where the index is indicative of the requirements name or ID, and the value at said index being the associated cost to implement the feature.

Given that we already have the requirement vector from parsing the data, we are left with finding the score and cost vector for each permutation of requirements. These vectors will be used to provide the final fitness values for both the multi-objective and single objective solutions. Both vectors are of the same length as the requirements vector, as the value at each index of the score and cost vector respectively represent the score and cost of each requirement in an individual.

To calculate the final cost of a given individual, we simply take the dot product of the individual's cost vector and the requirement vector, giving us the total cost to implement all the requirements in the individual.

The final score is slightly more involved and will require analysis of each customers value to the company, as well as how many customers require a given feature and how much they want it. To do this every customer will be assigned a weight based on the amount of income they bring to the company. Then a value function will be created to assess a requirement's overall value to all customers. Thus a score for any given requirement can be denoted:

Figure 1. Score Calculation

$$score_i = \sum_{j=1}^{m} w_j \cdot value(r_i, c_j)$$

Where *i* represents the requirement, *m* is the total number of customers, *j* is the current customer and *w* denotes the weight of a given customer (*j*). Once the score has been calculated for all the requirements in an individual, the dot product will again be taken with the requirements vector for an overall score of the permutation.

The single objective function will reuse the final cost and score calculation for a given individual, but instead of the evaluation returning both of these values, a single weighted-sum value will be returned. The single objective formula is represented as:

Figure 2 . Single Objective Formula

$$F(x) = w \cdot f_1(x) + (1 - w) \cdot f_2(x)$$

Where x represents the individual, f1 represents score, f2 cost and w the weighting to apply, which will in turn favour one objective more than the other.

Chapter 2 : Implementation:

Packages / Libraries used:
DEAP - (Distributed Evolutionary Algorithms in Python) is a framework that provides almost all of the functionality needed to set up a GA in python with only the fitness function to be written by the user. It is easily extensible, very easy to use and was the primary tool when developing this system.

SCOOP - (Scalable Concurrent Operations in Python) was used in conjunction with DEAP to allow for multi-threading capabilities. This was a necessity as the runtime of each iteration is quite time consuming and fast iteration was extremely important during the development of the system. One side-effect of using SCOOP was the need to keep the entire system in one “main” file. This is due to the nature of Python’s package system, which would of required a large refactoring of code to make work again. As the multi-threaded functionality was added fairly late in to the development of the system, the decision was made to keep everything in one file.

PyPlot / Imageio - These libraries were used to plot the results of all algorithms and provide an animation to show the formation of the final results across each generation.

Development:
DEAP (Single / Multi Objective GA)
To begin using DEAP, we must first make a creator object which will allow us to define a fitness alias and Individual for both the single and multi-objective runs.

Figure 3. Creator initialization

```
creator.create("FitnessMulti", base.Fitness, weights=(1.0, -1.0))
creator.create("FitnessSingle", base.Fitness, weights=(1.0, ))
creator.create("Individual", list, fitness=creator.FitnessMulti)
creator.create("SingleIndividual", list, fitness=creator.FitnessSingle)
```

As can be seen in Figure 3, we define two fitness functions: one multi-objective, which will maximise its first objective and minimise the second (1.0, - 1.0); while the single objective is simply trying to maximise its only objective. We then also define what form our individuals will take, which for both functions is a list of 0’s and 1’s. For now, we simply tell DEAP the individual is in the form of a list.

After creating our two fitness functions (which don't yet do anything), we must set up a toolbox to define how we will evaluate, mutate, crossover and select individuals for each solution. At this point we must also tell deap exactly what form our individuals are going to take and generate a population. All of this functionality is contained within the Toolbox object.

Figure 4 . Multi-Objective Toolbox

```
# set up DEAP
# https://deap.readthedocs.io/en/master/tutorials/basic/part1.html
# https://deap.readthedocs.io/en/master/overview.html
# define aliases for fitness function and individual representation
# create an NSGA2 toolbox
nsgaToolbox = base.Toolbox()
# enable multi-threading
nsgaToolbox.register("map", futures.map)
nsgaToolbox.register("attr_bool", random.randint, 0 ,1 )
nsgaToolbox.register("individual", tools.initRepeat, creator.Individual,
                     nsgaToolbox.attr_bool, n = numberOfRequirements)
nsgaToolbox.register("population", tools.initRepeat, list, nsgaToolbox.individual)
pop = nsgaToolbox.population(n=popSize)
nsgaToolbox.register("mate", tools.cxTwoPoint)
nsgaToolbox.register("mutate", tools.mutFlipBit, indpb=0.1)
nsgaToolbox.register("select", tools.selNSGA2)
nsgaToolbox.register("evaluate", evaluate, customers=customers, customerWeights=customerWeights,
                     requirements=requirements, numberOfRequirements=numberOfRequirements)
```

Firstly, we will look a the multi-objective toolbox. We create a toolbox and register SCOOP's map function - this is what allows the algorithm to work across multiple threads. It is also possible to set up multi-threading in a similar way with python's built in multiprocessor module, however DEAP specifically mentions good compatibility with SCOOP and as such it was used to minimise potential issues.

Next we define our individual. We tell DEAP each element of the individual will be of type attr_bool, which is a random int between 0 and 1. We then define an individual as a list of attr_bools. Finally, we define the population as a list of individuals. This is only the skeleton of the population, as we will need to specify the population size and number of generations when using one of the built in evaluation algorithms.

Next, we define each of the tools we wish to use for the various stages of the GA. A standard two point crossover was used for mating, a flip bit method for mutation and importantly the NSGA-II algorithm for selection. Finally, we register the evaluation function we have written to the the toolbox.

Figure 5. Multi Objective evaluation

```
def evaluate(individual, customers, customerWeights, requirements, numberOfRequirements):
    scores, costs = [], []
    for j in range(numberOfRequirements):
        if(individual[j] == 1):
            costs.insert(j, requirements[j])
            scores.insert(j, getScore(requirements[j], customers, customerWeights))
        else:
            costs.insert(j, 0)
            scores.insert(j, 0.0)


    # dot product with requirements for indiviudal final cost and score.
    finalScore = numpy.dot(scores, requirements)
    finalCost = numpy.dot(costs, requirements)
    return (finalScore, finalCost)
```

The evaluation function returns two values (the first to be maximised, and the second to be minimised, as per our fitness alias in the creator object). To achieve the overall score and cost of each individual, we create two lists, one for the costs and one for the scores. The cost vector is extremely simple - if the requirement occurs in the individual, find its price and append this to the list, and if it does not occur, simply add 0.

To get the given score of a requirement, we must first know each customer's worth to the company, represented as the customerWeights. This is extremely simple to calculate: we take the profit from a given customer and divide it by the total profit of all customers, meaning the sum of all customer weights will add up to 1. Next, we must iterate through each customer and assess how much they want a given requirement.

Figure 6. Value and Score Calculation

```
def getValue(requirement, customer):
    customerRequirements = customer[1]
    numberOfCustomerRequirements = len(customerRequirements)
    if requirement in customerRequirements:
        # returns value in range 0-1 based on how important requirement is to customer
        return (numberOfCustomerRequirements - customerRequirements.index(requirement)) / numberOfCustomerRequirements
    else:
        return 0


def getScore(requirement, customers, customerWeights):
    score = 0
    for i in range(len(customers)):
        score += customerWeights[i] * getValue(requirement, customers[i])
    return score
```

We use the getValue function to calculate the requirement's worth to a customer. This function simply checks if the customer wants the requirement and if so, returns a value in the range 0 to 1 based on what position the requirement occurred in the customer's list of requirements (e.g. how much do they need this requirement). If the customer does not need the given requirement, getValue returns 0. The overall score of a requirement is thus found by summing every customer's requirement value multiplied by their weight, resulting in a vector of scores after iterating through each requirement in the individual.

Once we have both the cost and score vector, we take the dot product with the requirement vector for both, giving us the final score and cost values which will be used as the main fitness values during the multi-objective run.

The single objective algorithm is setup in an almost identical way, using the same mutation, selection and crossover methods, with only a new evaluation method and fitness weightings, favouring one higher value. (See Figure 3)

Figure 7. Single Objective Evaluation

```
def singleEvaluate(individual, customers, customerWeights, requirements, numberOfRequirements, weight):
    # Single objective forumla
    # f(x) = w . f1(x) + (1 - w).f2(x)
    finalScore, finalCost = evaluate(individual, customers, customerWeights, requirements, numberOfRequirements)
    final = numpy.dot(weight, finalScore) + numpy.dot((1 - weight), finalCost)
    # must return two values for deap (expects all fitness functions to be multi-objective formatted)
    return final,
```

The single objective evaluation reuses the multi-objective evaluation method, with finalScore and finalCost representing the first and second fitness value in Figure 2. We use a weight value between 0.1 and 0.9 as a weighted sum for the evaluation, meaning lower values will favour the first objective and higher values will favour the second objective.

With the toolboxes configured, we are now able to run the GA.

Figure 8. Single Objective GA Run

```
pop, logbook = algorithms.eaMuPlusLambda(pop, singleObjectiveToolbox, popSize, popSize, crossover, mutation, nGens, stats, hof, True)
```

Figure 9. Multi-Objective GA Run

```
pop, logbook = algorithms.eaMuPlusLambda(pop, nsgaToolbox, popSize, popSize, crossover, mutation, nGens, stats, hof, True)
```

Running the GA returns a final optimised population and a statistics object named logbook, which can be used to extract information about the run for plotting.

Random

For random comparison, a simple algorithm was written to fill a population of individuals with random 0s or 1s.

Figure 10. Random Generation

```
def randomGenerate(popSize, numberOfRequirements):
    pop = []
    for i in range(popSize):
        individual = []
        for j in range(numberOfRequirements):
            individual.append(random.randint(0 , 1))
        pop.append(individual)

    return pop
```

After the population has been generated, we calculate its score and fitness using the same evaluate function as the GA. The scores for each generation are returned for plotting.

Figure 11. Random Algorithm

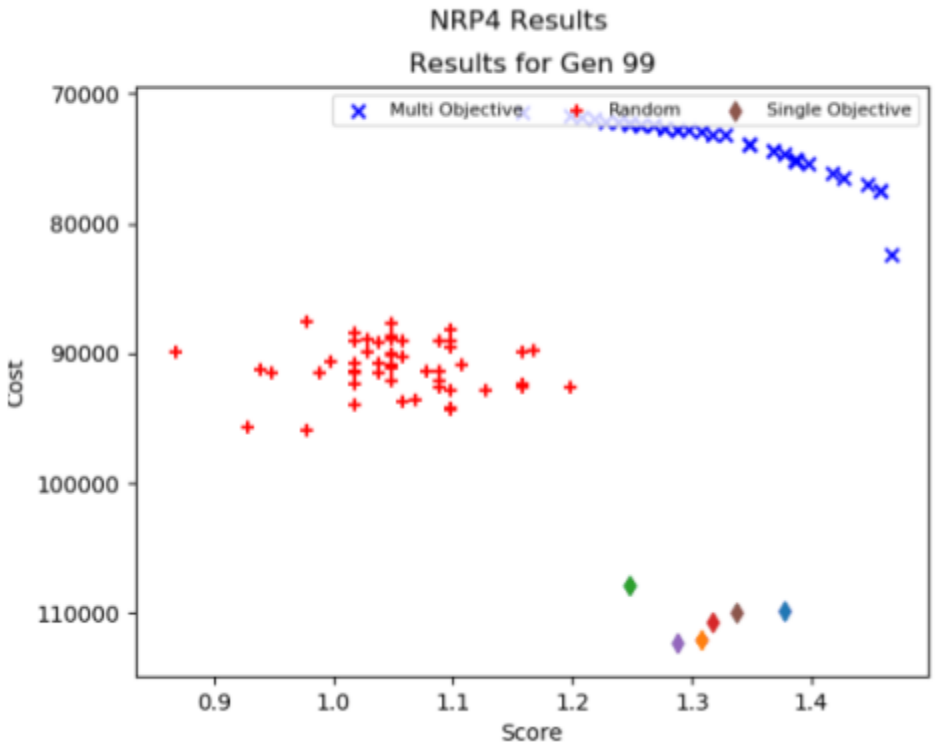
```
def randomAlgorithm(raw_data, numberOfGenerations, populationSize):
    popSize = populationSize
    nGens = numberOfGenerations
    requirements, customers = parseFile(raw_data)
    numberOfRequirements = len(requirements)
    totalProfit, customerWeights = getWeights(customers)

    finalRandomResults = []
    for i in range(nGens):
        print("Random Algorithm: Gen %d" % i)
        pop = randomGenerate(popSize, numberOfRequirements)
        popResults = []
        for j in range(popSize):
            currentScore, currentCost = evaluate(pop[j], customers, customerWeights, requirements, numberOfRequirements)
            popResults.append((currentScore, currentCost))

        finalRandomResults.append(popResults)

    return finalRandomResults
```

Chapter 3 : Results

Across both the classic and realistic data-sets, all parameters of the GA were the same: a population size of 50, 100 max generations, 75% crossover rate and 0.1% mutation rate. When discussing the single objective results, the algorithm was run 6 times with weightings of 0.1, 0.25, 0.4, 0.55, 0.7 and 0.9 in the hope that this would accurately represent what the single solution was capable of.

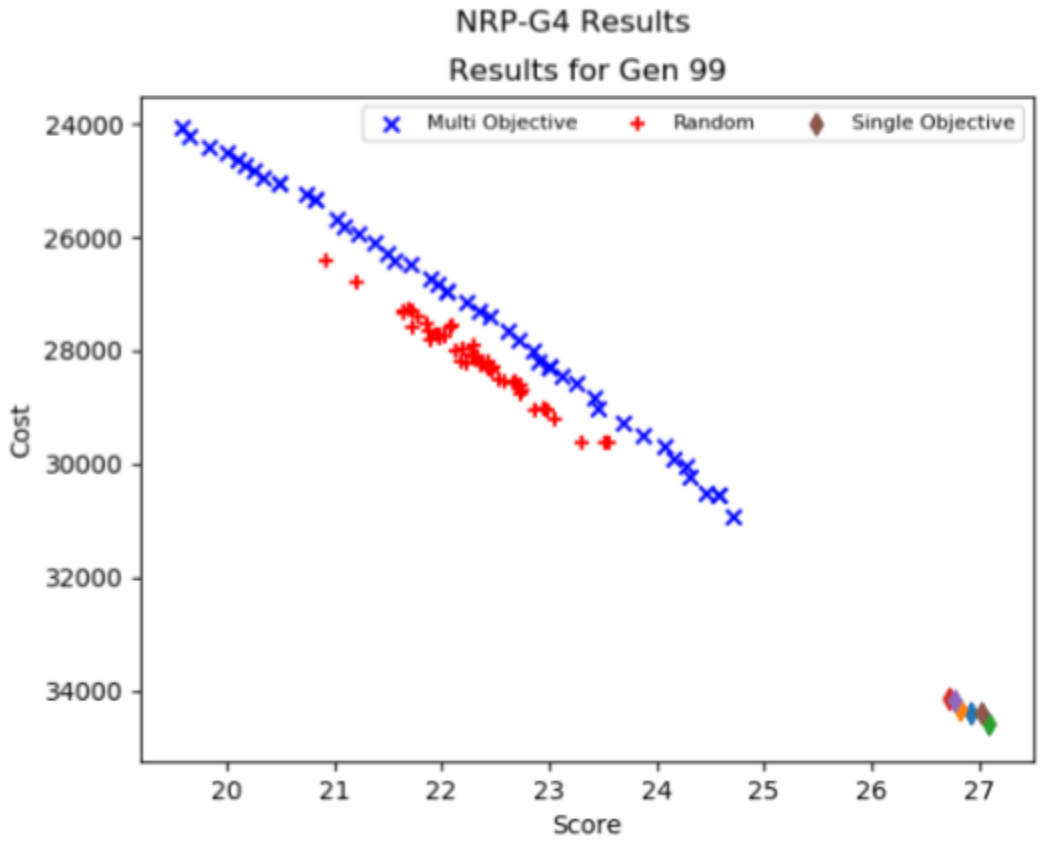
Classic NRP-4 Results

Figure 12. NRP4 Final Results



For the NRP4 Dataset, the Multi-Objective solution was the best solution by far, providing the highest scores for the lowest cost. The single objective was able to find high scoring individuals but across 6 different weights was much less concerned with the cost. It is strange to note that single objective did not form a pareto curve while sampling different weights, despite the fact an NSGA II non dominated selection was used. This could be due to the fact that each plot from single objective comes from a different population, and slight errors are being introduced due to the relatively low population count.

Figure 13, NRP G4 Final Results



The realistic NRP-G4 Set gave interesting results, with the multi-objective providing the best balance between cost and score while the single objective favoured raw score with no regard for cost. Random was much closer to the multi-objective than expected, this could be due to the fact that in the NRP-G4 data set many of the customers have a large amount of requirements, meaning more of the requirements are being covered even with a purely random distribution. Furthermore the single-objective is extremely concentrated in the high-score and high cost area of the graph, which may be indicative of the single objective algorithm needing further refinement.From the results, the single objective is very good at finding high score results, if the weight of the single objective fitness was flipped to -1, we would likely see a strong occurence of low scoring and low cost individuals. This highlights the strengths of the multi-objective GA, which is clearly suited to the NRP problem much more than a single objective solution.