Advanced Software Engineering
Assignment 3

Sukhpal Singh
Reg. No. 201788436

Overview

Objective: To understand how to search for the optimal solution by using the Next Release Problem (NRP) as a template problem. This uses multiple objectives and fitness functions to find an ideal set of solutions. For the NRP case, the optimal case is choosing the number of requirements to be made to keep the customers happy whilst keeping implementation cost for the company at a minimum.

Next Release Problem

The next release problem is based on an organisation having many customers with different requests for functions the company offers. The organisation gives their customers a selection of functions they offer to implement. Each required function (requirement) is placed into an ordered list depending on how important it is to the customer. Not all customers are treated equally, and are weighted to determine how important they are to the company. Each requirement carries its own weight, which depends on the number of requests made for its implementation and the weight of the customers requesting it be implemented. This prioritises implementation of functions with higher weights. This data can then be plotted to show which requirements are more in demand, as well as the cost of implementation. Choosing the best order to implement those functions, in the interest of both the clients and the organisation, can be found using a Pareto front (PF). A PF is used to evaluate the solutions from an output that meet the constraints of the organisation. The constraints can be either keeping the customer happy, or keeping the cost of implementation low. The PF places significance on solutions that achieve an objective better than the other solutions. The better the solution is over other solutions, the likelier that solution will be chosen to be added to the front.

Single/Multi-Objective genetic algorithms

The objective of a Single-Objective GA (SOGA) is to determine how to get the best solution based on one given objective. The Multi-Objective GA (MOGA) is focused more towards finding the best solution based on multiple objectives. Given that there is an unconstrained budget, a SOGA would be used to maximise the customer happiness. Whereas, if there is a cost restriction as well as customer satisfaction needing to be prioritised, then a MOGA would be used.

The text file

The first value in the text file indicates the number of levels of requirements. This meant that multiplying 2 by that value indicated how many values to skip over, because those were not required pieces of information for this assignment. The next important value was the

number of dependencies, therefore using this value added to the level of requirements value, gave a clear marker of where the required information was located. The next value was the number of customers, therefore using this in conjunction with its placement in the text file, the customer information could be acquired. The rows were then read into a dataframe using pandas.

Choice of implementation

Whilst there are many frameworks to use to implement a solution for the NRP problem such as ecspy, deap, evoalgos, platypus, the route taken was to use pandas, alongside numpy. Due to frameworks not being used, not all functions are designed as effectively as they would've, had a framework been used. What pandas offered was management of large volumes of data using dataframes, with functions that enabled adding, subtracting, merging dataframes, along with many more.

Customer weight

The weight of all the customers was normalised by summing overall customer weights and dividing each customer weight by the summation. This gives the proportion that customer offers to the company overall.

Requirement weight – Score fitness

The requirements were ordered from left to right, where the left location was more important than the farthest right requirement in that row within the dataframe. To weight each requirement in the order they were found in the original text file, the row with the maximum number of columns occupied was found, and used to make each and every row have the same length, filling any missing values within rows with zeros.

The columns holding the requirements were then turned into a list, list 0. Another list was made filled with values, using 1 divided by the first column number to the max number of columns, list 1. Using list 0, another list was made identifying the locations that had values larger than 0, list 2. Those locations were filled with 1's. List 1 was multiplied with list 2 to make list 3. This found the weight each requirement had depending on its placement within each row. List 3 was then truncated by the maximum column number to make nested lists, list 4. List 4 was then turned into a dataframe. The 3rd list was then concatenated with the list of requirements in the order they were found in the dataframe. To find the overall weight, each requirement was grouped using the groupby and sum functions, to give the overall score for each requirement.

Cost fitness

The next part was to find the cost fitness. This was simply done by dividing 1 by the cost (requirement value).

Understanding the fitness's

Lastly to understand the fitness's directly, the higher the cost fitness the lower the cost of implementation. Whilst the higher the score fitness, the more times the requirement was

chosen, or the weight of the customers wanting that requirement was higher. When comparing the cost fitness and score fitness, both score fitness and cost fitness were normalised by dividing each value by the total sum of all scores/costs.

Mutation

Mutation occurred only on the 3^{rd}-5^{th} fronts because, those are not the most important fronts when considering what requirements are best suited for a given objective. Instead the 1^{st} and 2^{nd} are, so they had no form of mutation applied on them. The mutation operated by looping over a specified number of iterations, each loop chose a different random integer. If that integer was within a specific region, then one of the three fronts were mutated, by randomly selecting one sample within a front that had been triggered by the random integer meeting that frontal requirements, and another from the mutation population, then switching them. Irrespective of whether that sample could have been better than the already population within that front.

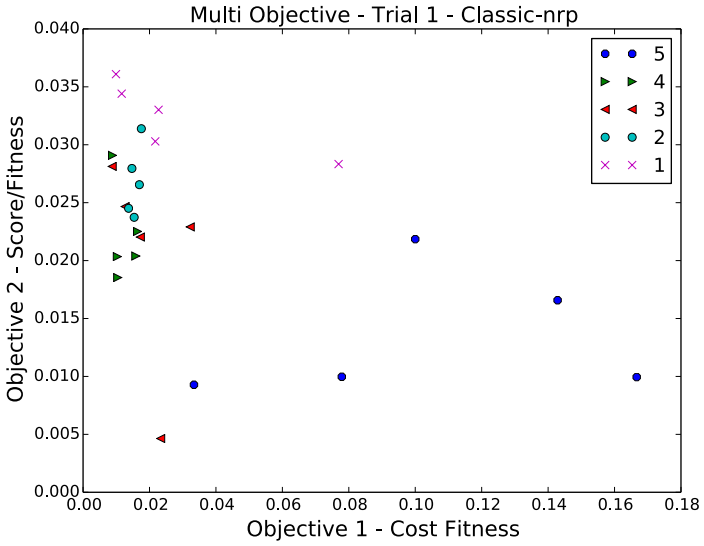
Crossover – multi-objective

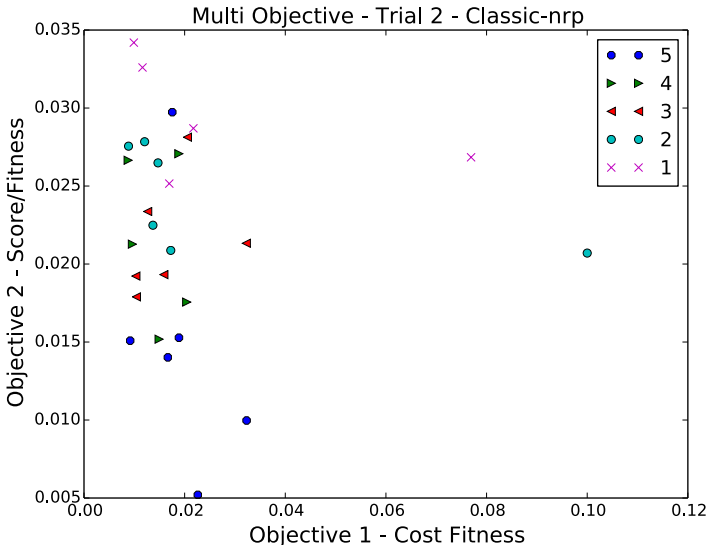
Cross over happened to all five of the fronts. Cross over implementation was based on using the crowding distance. If the spacing between the first and second point was larger than the spacing between the second and third point, then the third point was crossed over with one of the other fronts. The spacing was calculated using Pythagoras's theorem, $x^2+y^2=z^2$. The cross over happened between the next front in the number of fronts, eg. front 1 crossed over with front 2, front 2 crossed over with front 3 etc. This choice was made since the method used for developing the fronts was based on getting the highest fitness requirements and ordering them. Therefore, they would have been grouped together in a linear fashion. However, to space out the requirements to cover both objectives, requirements were chosen from the next front, since those requirements should still be high scorers but may possibly have better fitness for a different objective.

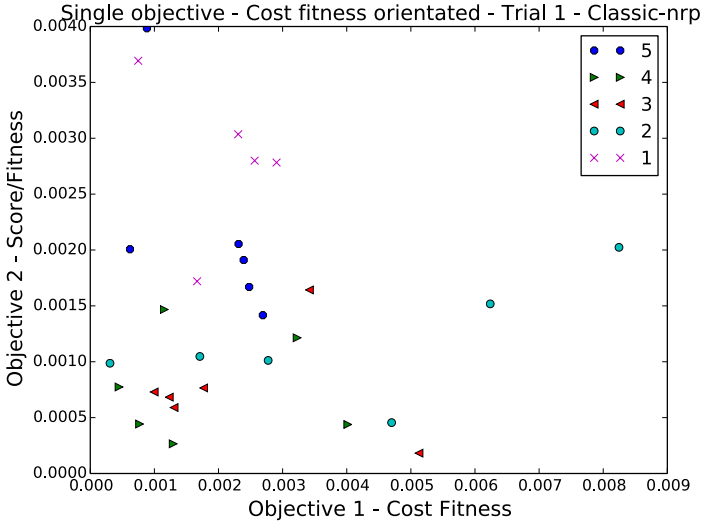
Crossover – single-objective

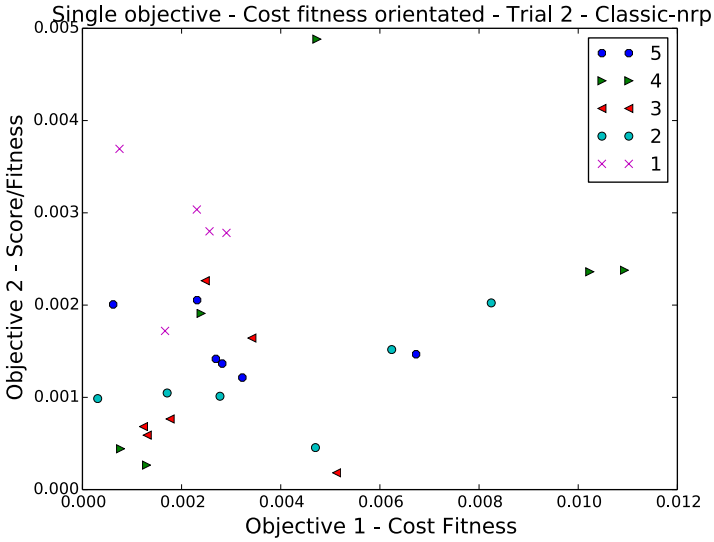
Crossover for the single-objective was similar to the multi-objective, but instead of placing x as the cost and y as the score within Pythagoras's theorem, only one fitness was chosen, e.g x=cost and y=cost, or x=score and y=score.

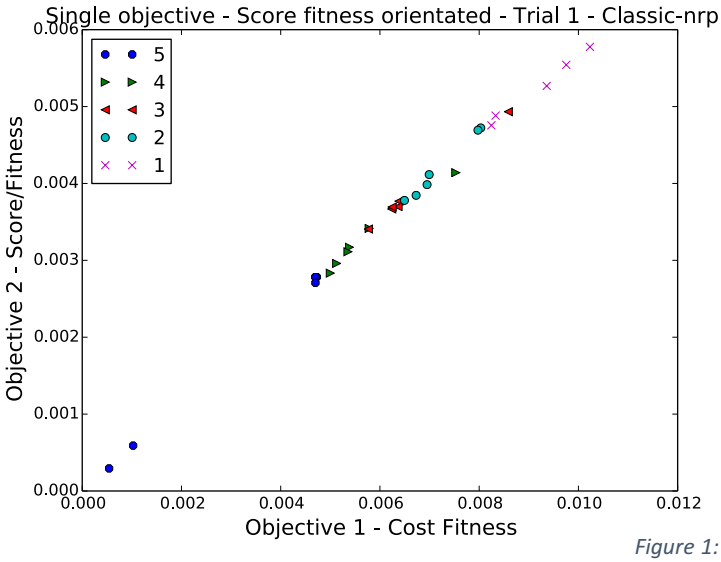
Results











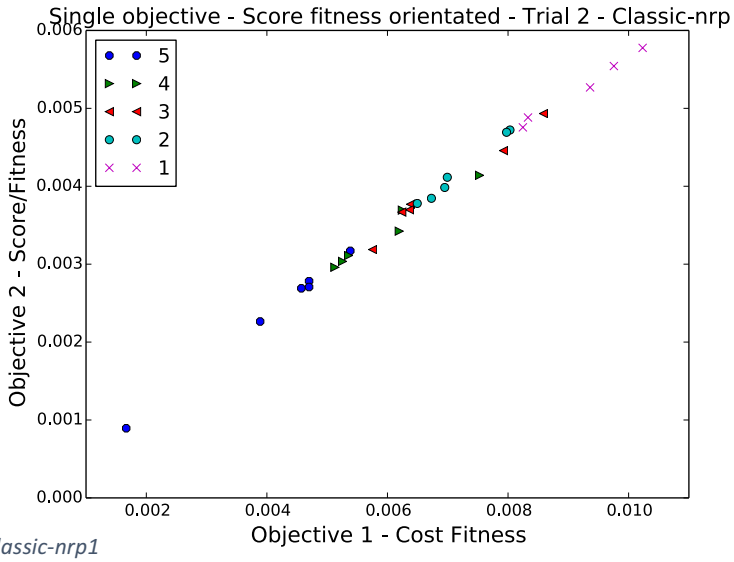
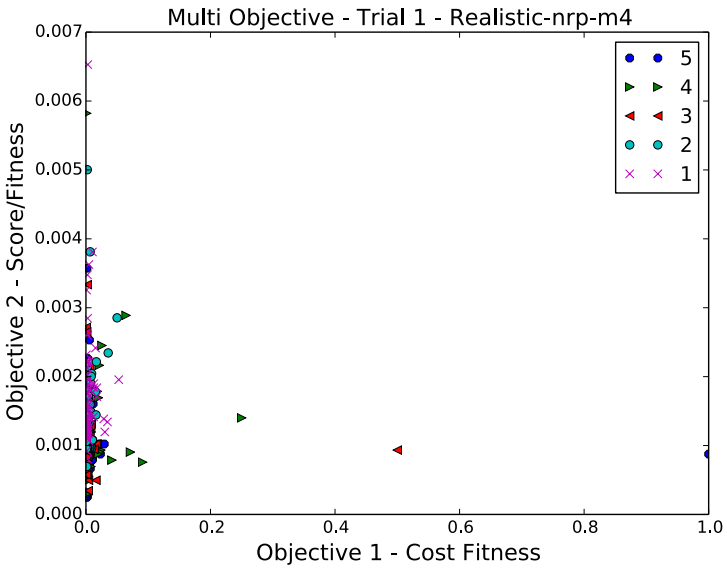
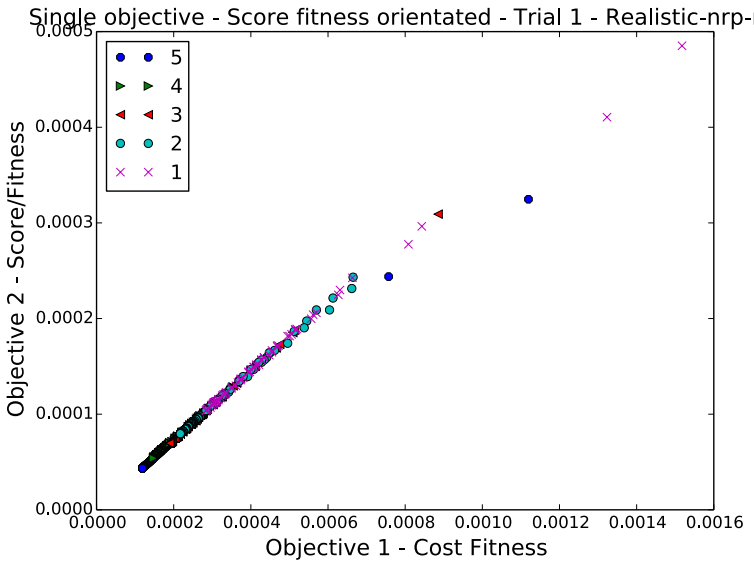


Figure 1: Classic-nrp1

Each front is represented by a number given in the legend. In figure 1, requirements dispersion for the MOGA case provides a result based solely on the fitness's, irrespective if there being any extra weighting for a given objective. Whilst the crowding function implemented was intended to disperse the requirements over a greater range than what is observed, it did not achieve this. Thus, the MOGA has clumped requirements.

For the case of the SOGA, the requirements are spread more across the objective it's concerned with. The requirements can be seen more dispersed towards the objective. This is most transparent where the intended front 1, covers a greater spread over a specific objective. The SOGA case is also more reliable at giving solutions to one scenario rather than covering multiple, as is intended from a SOGA.





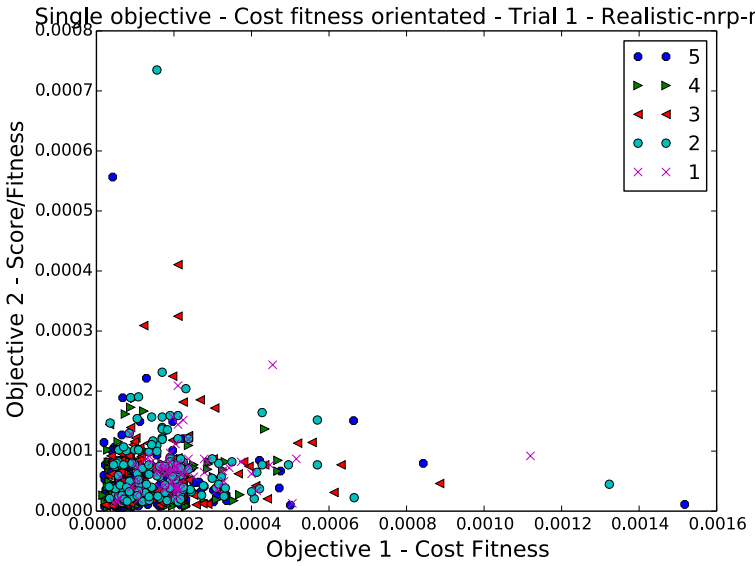


Figure 2: Realistic nrp-m4

Similarly, the results in figure 2 harmonise with what was found in figures 1. Since the aim of the MOGA is based around gathering a generally good solution, the dispersion of the requirements is not required to be spread over any one objective. Therefore, a clumped solution towards one objective is still an acceptable solution.

In conclusion, the main issue is a better method for finding appropriate requirements for each front is needed. Whilst the fitness functions could be better chosen, the most dominating problem is selection of appropriate frontal requirements.

Random search

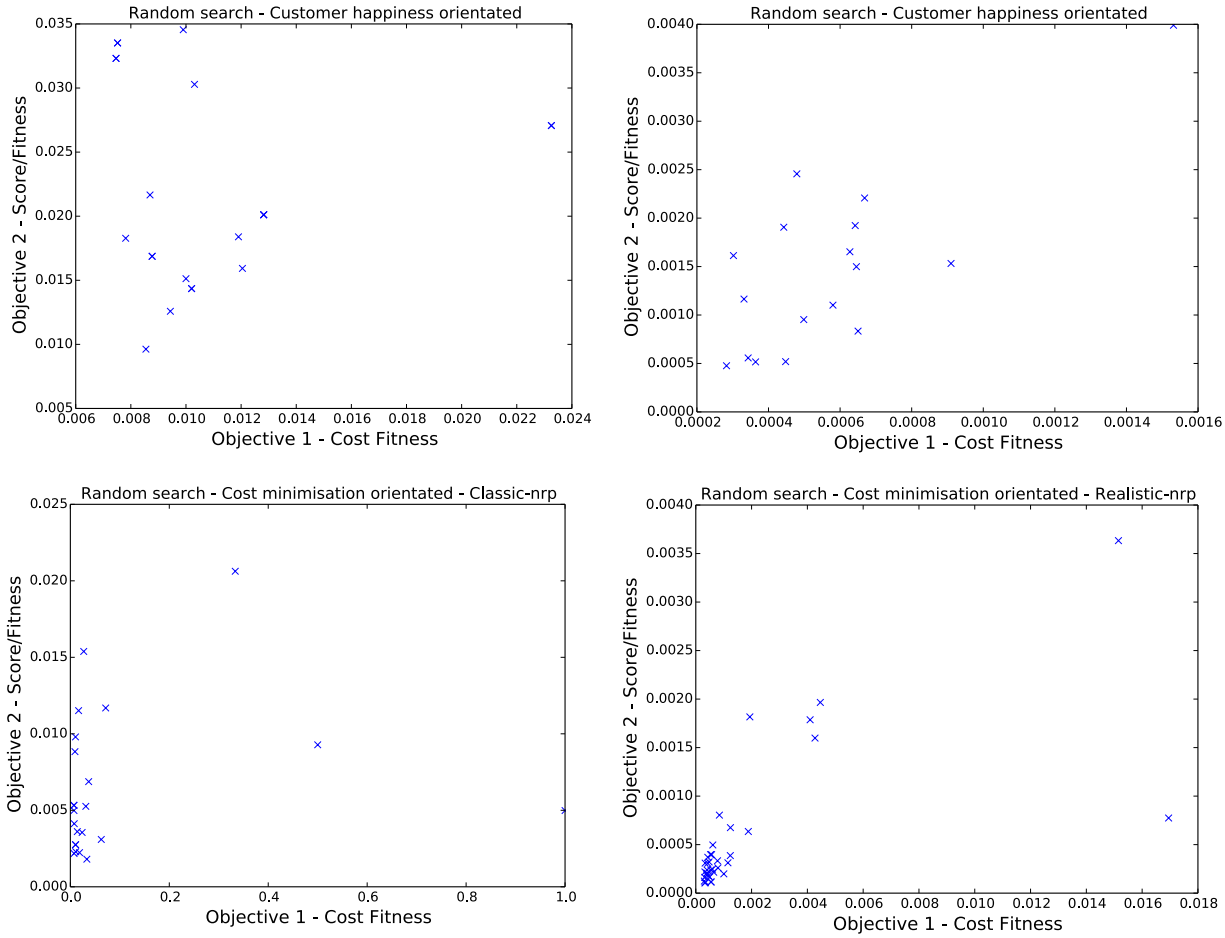


Figure 3: Random Search

Despite the random case having no guidance towards what requirements should be chosen, other than a simple sort towards one given objective, the solutions are still reasonable in contrast with what the multi and single objective solutions provide. This is primarily due to the frontal solutions being the problem rather than any other problem.