Requirements Analysis

Martin Tiggerdine

Contents

Overview of Key Aspects of Problem and Solution 2

- The Next Release Problem 2
- Classic and Realistic Data 2
- Representation 3
- Three Solutions 3
- Fitness Function 3
- The NSGA-II Algorithm 4

Implementation Details 5

- Java or Python? 5
- Platypus 5
- PyPlot 5
- Reading 6
- Solving With Platypus 6
- Plotting With PyPlot 8
- Analysing 8
- Comparing 8

Presentation of Results (Classic) 9

- Single 9
- Multi 10
- Comparison 11

Presentation of Results (Realistic) 13

- Single 13
- Multi 14
- Comparison 14

Comparison of Results with Random Solutions 16

Overview of Key Aspects of Problem and Solution

The Next Release Problem

The next release problem (NRP) is an NP-hard requirements engineering problem. The idea is to choose the optimal set of requirements to be included in the next release of a system that meets any constraints (such as budget) but includes the most desirable requirements given:

- some requirements take more effort to implement than others
- some requirements are dependent on others
- a system may have many customers who have specified different requirements
- some customers are more highly valued than others
- not all requirements are equally important, and this importance may vary between customers

Classic and Realistic Data

There are a number of instances of the problem in two different formats, classic (generated under certain constraints based on the classic literature of the NRP experiments) and realistic (mined from open bug repositories of three open source software projects: Eclipse, Mozilla and Gnome). Each instance contains:

- the cost of each requirement
- the dependencies of each requirement
- the profit of each customer
- the requirements requested by each customer

Representation

I represented the problem using two lists, costs[] — the cost of each requirement — and scores[] — the score of each requirement, calculated by

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

where m is the number of customers, w_j is customer c_j's relative importance and $value(r_i, c_j)$ is the value that customer c_j places on requirement r_i.

I calculated w_j by dividing customer c_j's profit by the sum of every customer's profit, satisfying $w_j \varepsilon [0,1]$ and $\sum_{j=1}^{m} w_j = 1$.

I calculated $value(r_i, c_j)$ using the order of the requirements requested by customer c_j. Each requirement's value was 90% of last requirement's value, starting from 1. For example, if customer c_0 requested requirements r_1, r_2 and r_3 in that order, $value(r_1, c_1) = 1$, $value(r_2, c_1) = 0.9$ and $value(r_3, c_1) = 0.81$.

Three Solutions

There is more than one way to solve the problem. The first is to randomly generate releases. The second is to use a genetic algorithm to maximise score. The third is to use a multiobjective genetic algorithm to simultaneously maximise score and minimise cost.

Fitness Function

If score is being maximised, a release's fitness is its score (the sum of its requirements' scores). If score is being maximised and cost is being minimised, a release's fitness is a tuple of its score and its cost (the sum of its requirements' costs). If there is a budget, a release is feasible if its cost is less than or equal to the budget.

The NSGA-II Algorithm

I chose the NSGA-II algorithm highlighted in class to solve the problem. NSGA-II is a fast and elitist multiobjective genetic algorithm that yields not one but many solutions, hopefully nondominated ones. A solution is dominated if another solution is better than it in one respect and at least equal to it in all others. The set of nondominated solutions is known as the pareto front. Figure 1 is an example of how pareto fronts are found. The objectives are to minimise f1 and f2. C is dominated by A and B because it is worse in both f1 and f2. A and B are nondominated. In other words, they lie on the pareto front.

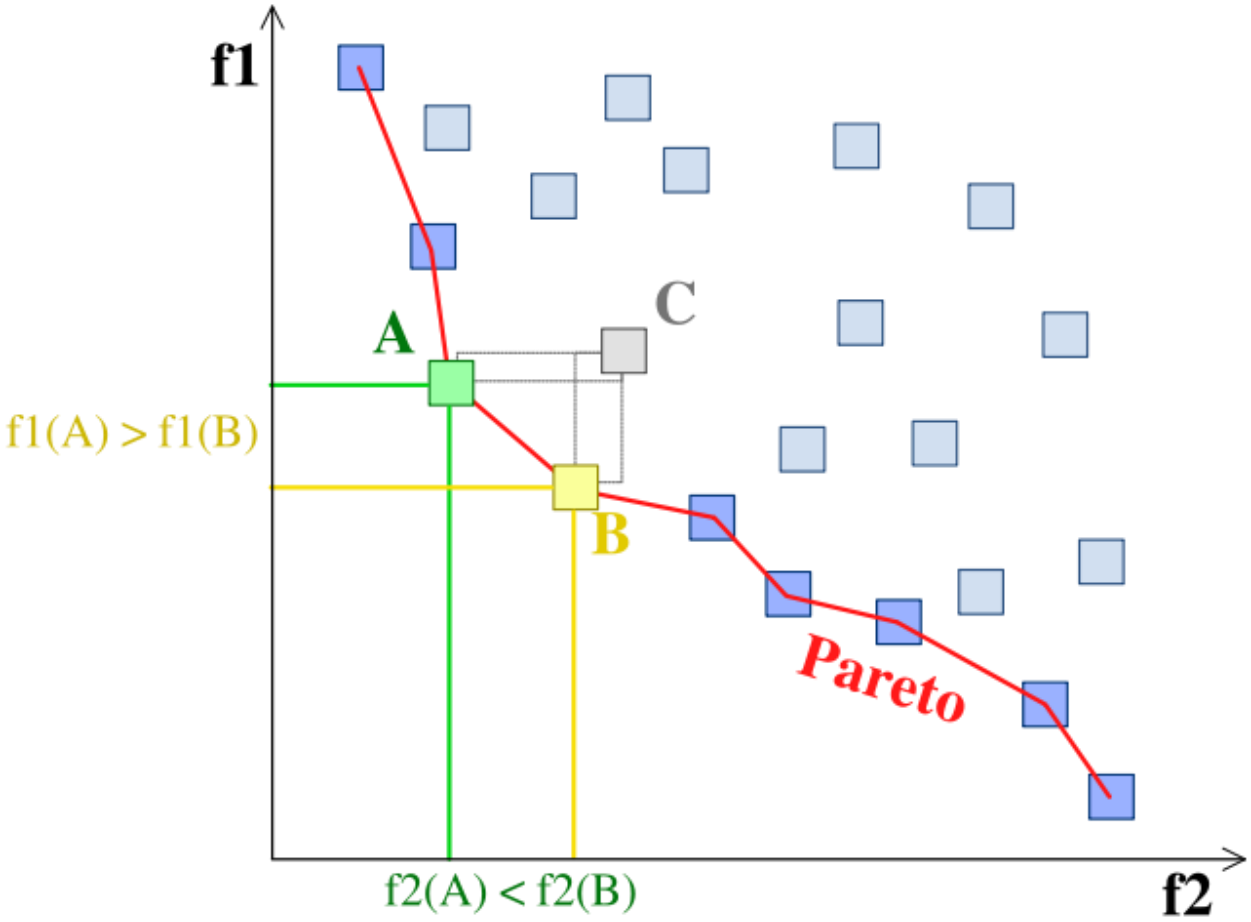

Figure 1: A pareto front.[1]

Using an off-the-shelf algorithm like NSGA-II instead of designing my own saved time because I didn't have to worry about crossover or mutation functions and basically guaranteed better results because it is known to work.

[1] https://commons.wikimedia.org/wiki/File:Front_pareto.svg

Implementation Details

Java or Python?

The first choice I had to make was whether to use Java or Python. I did what I always do and chose Java because I've used it a lot and Python not even a little. Everything was good until it was time to use NSGA-II. I had problems with both Opt4J and the MOEA framework. I thought about trying to code it myself but several friends said "use Platypus, it's the best." There was only one problem.

Platypus is a Python framework.

I threw my Java project in the bin used this as an excuse to learn Python.

Platypus

Platypus is a framework for evolutionary computing. It has three algorithms to choose from — NSGA-II, NSGA-III and ε-MOEA and some example problems. Using it has two main parts: defining a problem and solving it using an algorithm. A problem is defined by its decision variable(s), its objective(s) and optionally, its constraint(s).

PyPlot

The other framework I used, PyPlot, provides a MATLAB-like plotting framework. I don't know what else to say about it.

Reading

The first thing to do was to read the data. `read(filename)` is a switch that calls `cread(filename)` if the file is classic and `rread(filename)` if it is realistic. It also clears `costs` and `scores`, the two global lists that represent the problem. The readme and example files helped me to write `cread(...)` and `rread(...)`. They have a couple of differences because classic and realistic files are formatted differently but on a high level they are the same. They:

- read the cost of each requirement into `costs`
- skip dependencies
- read each customer's profit into `profits`
- add the value ($value(r_i, c_j)$) placed on each requirement by each customer to `values`
- calculate each customer's weight (w_j) and adds it to `weights`
- calculate each requirement's score ($score_i$) and adds it to `scores`

Solving With Platypus

I used the Defining Constrainted Problems example[2] in the Getting Started chapter of the Platypus documentation to understand how to use Platypus. `mul(filename, ratio, evaluations, show)` is the function that solves the two objective problem (maximising score and minimising cost). `ratio` is the cost ratio used to calculate the budget, `evaluations` is the number of function evaluations used to optimise the problem and `show` controls whether or not the results are plotted.

```
def mul(filename, ratio, evaluations, show):
    read(filename)
    budget = ratio * sum(costs)
    problem = Problem(len(costs), 2, 1)
    problem.constraints[0] = Constraint("<=", budget)
    problem.directions[0] = problem.MAXIMIZE
    problem.function = mfitness
    problem.types[:] = Integer(0, 1)
    algorithm = NSGAII(problem)
    algorithm.run(evaluations)
    if show:
        plot(algorithm.result, budget, "Multi")
    return [s.objectives[0] for s in algorithm.result],
           [s.constraints[0] for s in algorithm.result]
```

[2] https://platypus.readthedocs.io/en/latest/getting-started.html#defining-constrained-problems

The function calculates the budget and defines a problem with a number of decision variables equal to the number of requirements, two objectives and one constraint. It adds budget as the constraint and maximising score as the first objective (objectives are minimised by default so the second objective of minimising cost does not need to be explicitly added). The NSGA-II algorithm is chosen and run. The population size can be changed but I chose to use the default of 100. The function calls `plot(...)` to plot the results if `show` is true and returns the scores and costs of the releases returned by the algorithm.

The problem function's job is to take a release and return its objective and constraint values. `mfitness(...)` just calculate the cost and score of a release by summing the costs and scores of its requirements and returns `[score, cost], cost`.

The function to solve the single objective problem — `sin(...)` — is an adaptation of `mul(...)`. The only changes are that it has one objective, not two and uses a different problem function that reflects this. `sfitness(...)` is the same as `mfitness(...)` except it returns `score, cost` because there is only one objective.

The other thing to do was to generate random solutions against which to compare the others. This is the job of `ran(...)`. It takes the same parameters as `mul(...)` and `sin(...)` but works a bit differently. `sfitness(...)` is reused. Infeasible solutions are easily avoided so I chose to do so. `a` and `b` are used to make sure releases of different sizes are generated.

```
def ran(filename, ratio, generations, show):
    read(filename)
    budget = ratio * sum(costs)
    i = 0
    import random as r
    rcosts = [];
    rscores = [];
    while i < generations:
        a = r.randint(0, len(costs))
        random_solution = []
        for x in range(len(costs)):
            b = r.randint(0, len(costs))
            random_solution.append(int(b > a))
        fitness = sfitness(random_solution)
        if(fitness[1] < budget):
            i += 1
            rcosts.append(fitness[1])
            rscores.append(fitness[0])
    if show:
        rplot(rscores, rcosts, budget, "Random")
    return rscores, rcosts
```

Plotting With PyPlot

There are two plot functions. `plot(solutions, ...)` plots multi and single objective solutions and `rplot(x, y, ...)` plots random solutions. For the sake of not copying and pasting two more pages of code, I will say a few words about what they do. `plot(...)` plots infeasible, feasible and nondominated solutions in different colours. `rplot(...)` just plots solutions. Both plot the budget as a line too and calculate the x and y limits of the plot such that every solution is shown.

Analysing

I wanted to do some quantitative analysis.[3] `analyse(xys, table)` is what I came up with. `zs` is a list of each coordinate's average distance from every other coordinate.

```
zs = []
    for x in range(len(xs)):
        z = 0
        for y in range(len(xs)):
            z += hypot(xs[y] - xs[x], ys[y] - ys[x])
        zs.append(z / (len(xs) - 1))
```

The `min()`, `max()`, `range_()`, `mean()` and `pstdev()` of `xs`, `ys` and `zs` are calculated. `pstdev()` is used over `stdev()` because `xs`, `ys` and `zs` are populations, not samples. `analyse(...)` is used in two different ways. If `table` is false, it prints each stat. If `table` is true, it returns a list of each stat.

Comparing

`compare(...)` compares any combination of random, single and multi by plotting and analysing them. It only cares about nondominated solutions because dominated ones are irrelevant in this context. I won't waste any more time going through it line by line.

[3] This was before I asked a friend to send me the marking scheme and read "there is no need to perform any statistical comparisons".

Presentation of Results (Classic)

I chose problem nrp1 and cost ratio 0.3 (budget 257).

Single

I chose the number of function evaluations — the number that controls how long the algorithm runs — by a bit of trial and error. What I found was that evaluating the function too few times causes less of the solutions to be feasible. For example, Figures 2 and 3 show 100 and 500 function evaluations.

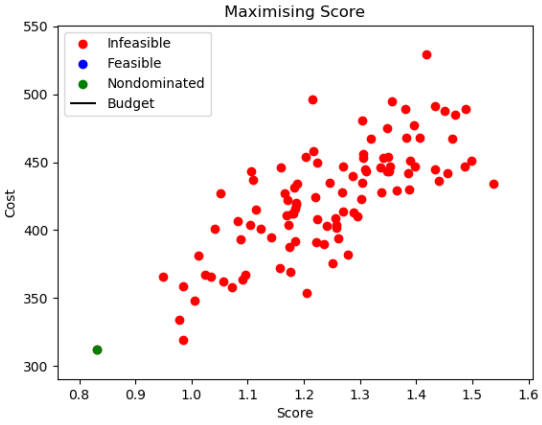

Figure 2: `sin('classic-nrp/nrp1.txt, 0.3, 100, 1).`

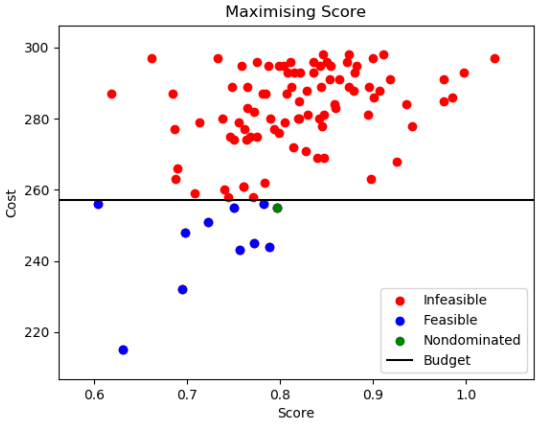

Figure 3: `sin('classic-nrp/nrp1.txt, 0.3, 500, 1).`

Each red dot is an infeasible solution and each blue dot is a feasible solution. After 100 function evaluations, not even one solution is feasible. After 500 function evaluations, about 10% of the solutions are feasible.

The marking scheme talks about running the single objective solution multiple times. I did so but there was so little variance in terms of the nondominated solution that it would be a waste of time to have every run's plot here. I think this is another side effect of doing lots of function evaluations.

Figure 4 shows `sin('classic-nrp/nrp1.txt, 0.3, 1000, 1).`

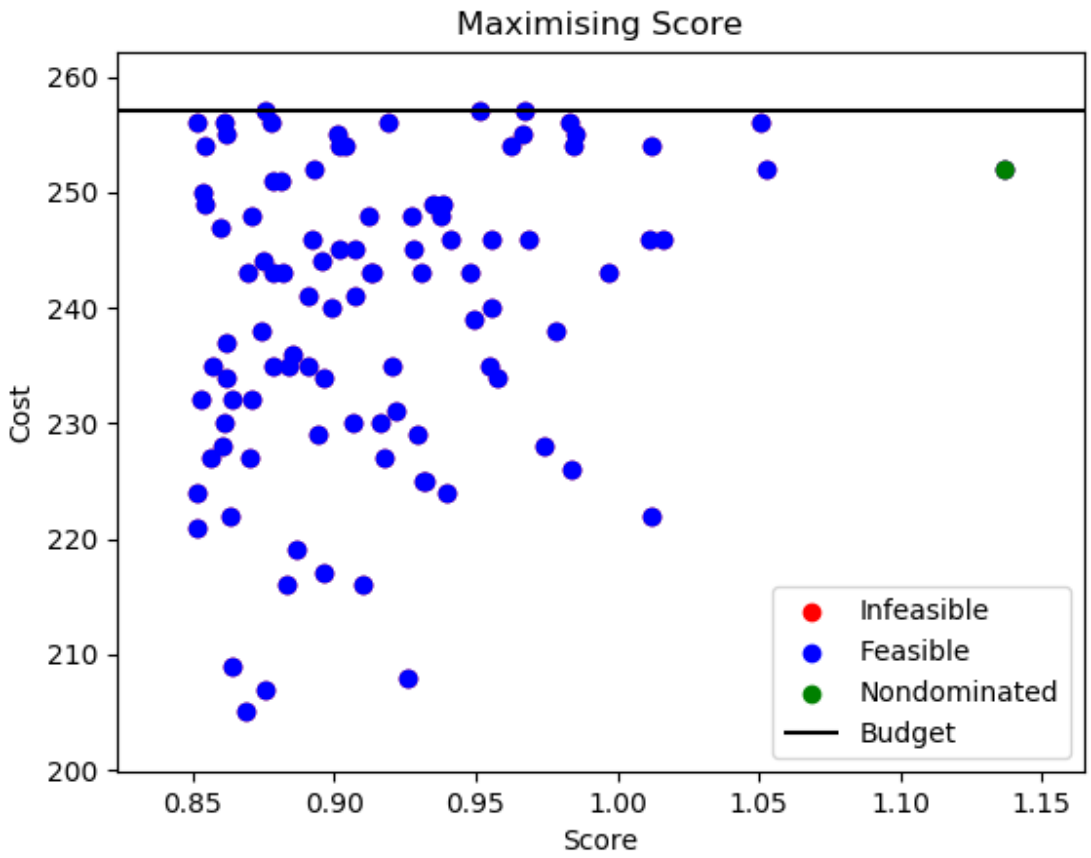

Figure 4: `sin('classic-nrp/nrp1.txt, 0.3, 1000, 1).`

The nondominated solution had score 1.13 and cost 252.

Multi

I used the same number of function evaluations to make the comparison fair. Figure 5 shows `mul('classic-nrp/nrp1.txt, 0.3, 1000, 1).`

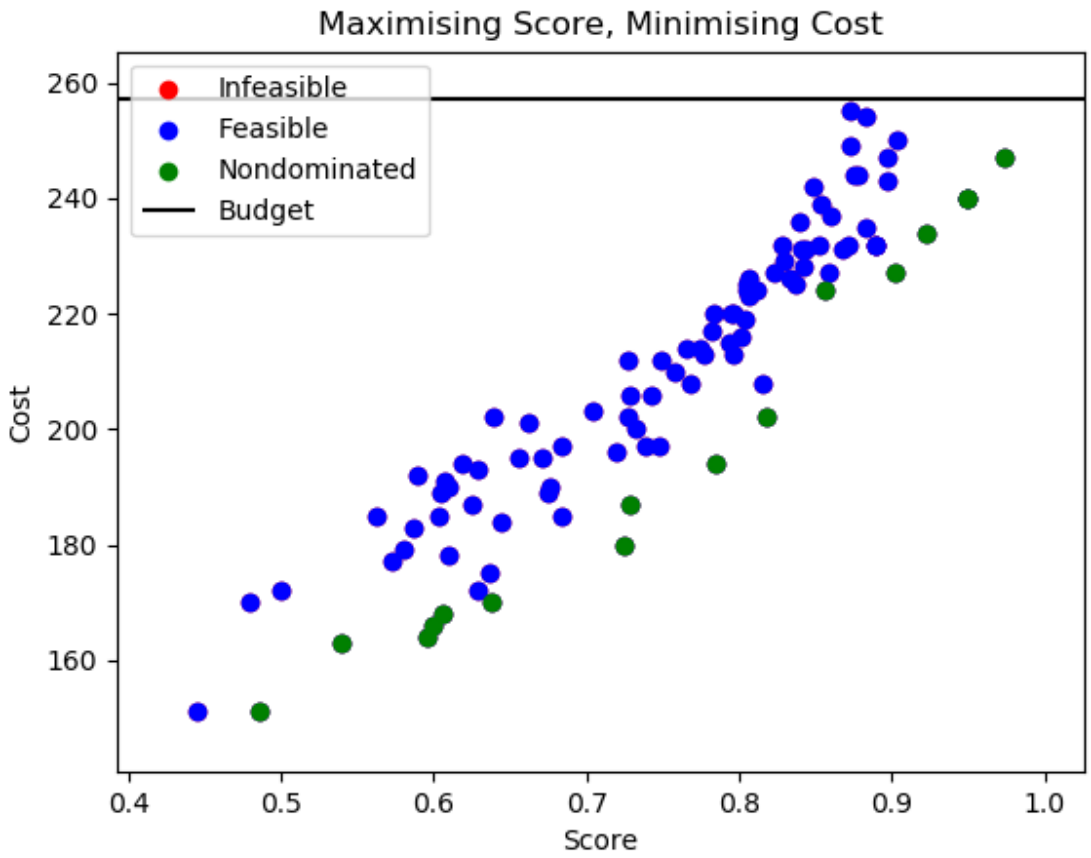

Figure 5: `mul('classic-nrp/nrp1.txt, 0.3, 1000, 1).`

The nondominated solutions had scores from 0.49 to 0.97 and costs from 151 to 247.

Comparison

I used `compare('classic-nrp/nrp1.txt',0.3,0,0,1,1000,1,1000)` to make it easier to compare nondominated solutions, Figure 6.

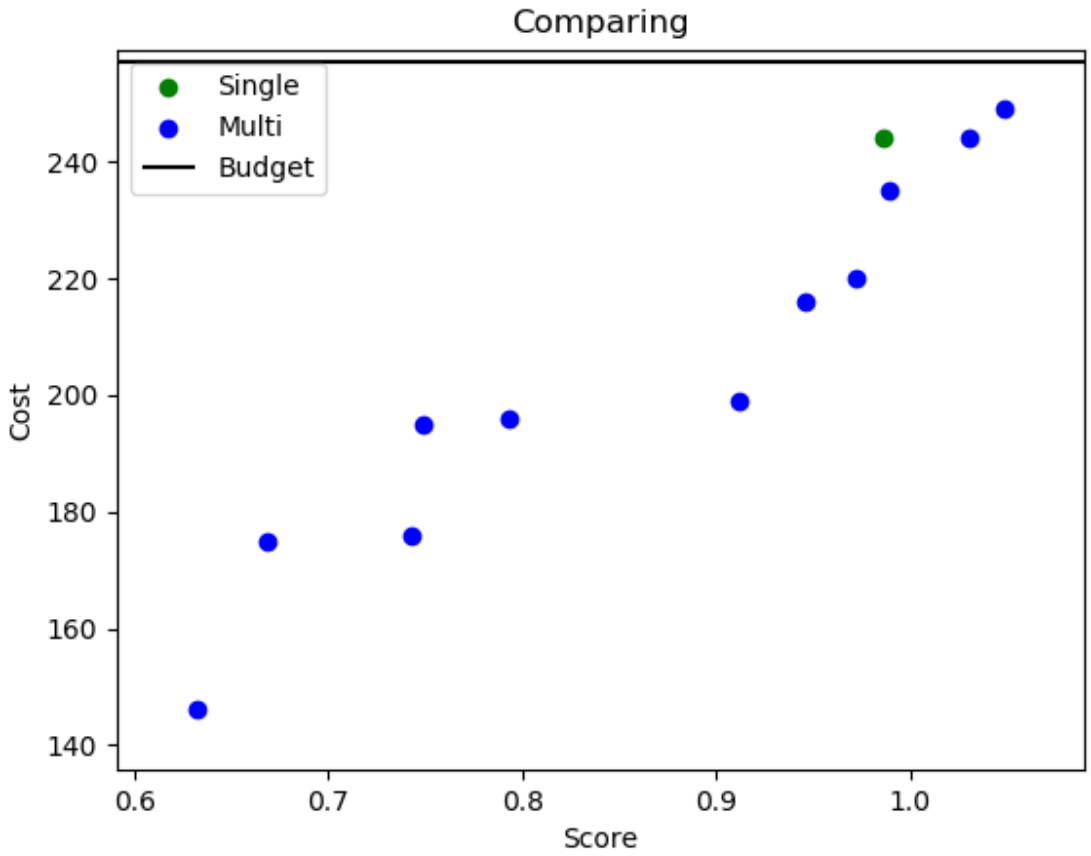

Figure 6: `('classic-nrp/nrp1.txt',0.3,0,0,1,1000,1,1000).`

What I learned was that single's nondominated solution is extremely similar to multi's highest score, highest cost solution. Sometimes it is a little better, sometimes it is a little worse. The advantage of multi over single is that it lets the dev choose what size they want the next release to be instead of forcing them to spend their whole budget.

Presentation of Results (Realistic)

I chose problem nrp-e1 and cost ratio 0.3 (budget 3495).

Single

Realistic problems are bigger then classic ones. I chose the number of function evaluations in the same way as before and found 10000 to be enough. I ran `sin('realistic-nrp/nrp-e1.txt', 0.3, 10000, 1)`, Figure 7.

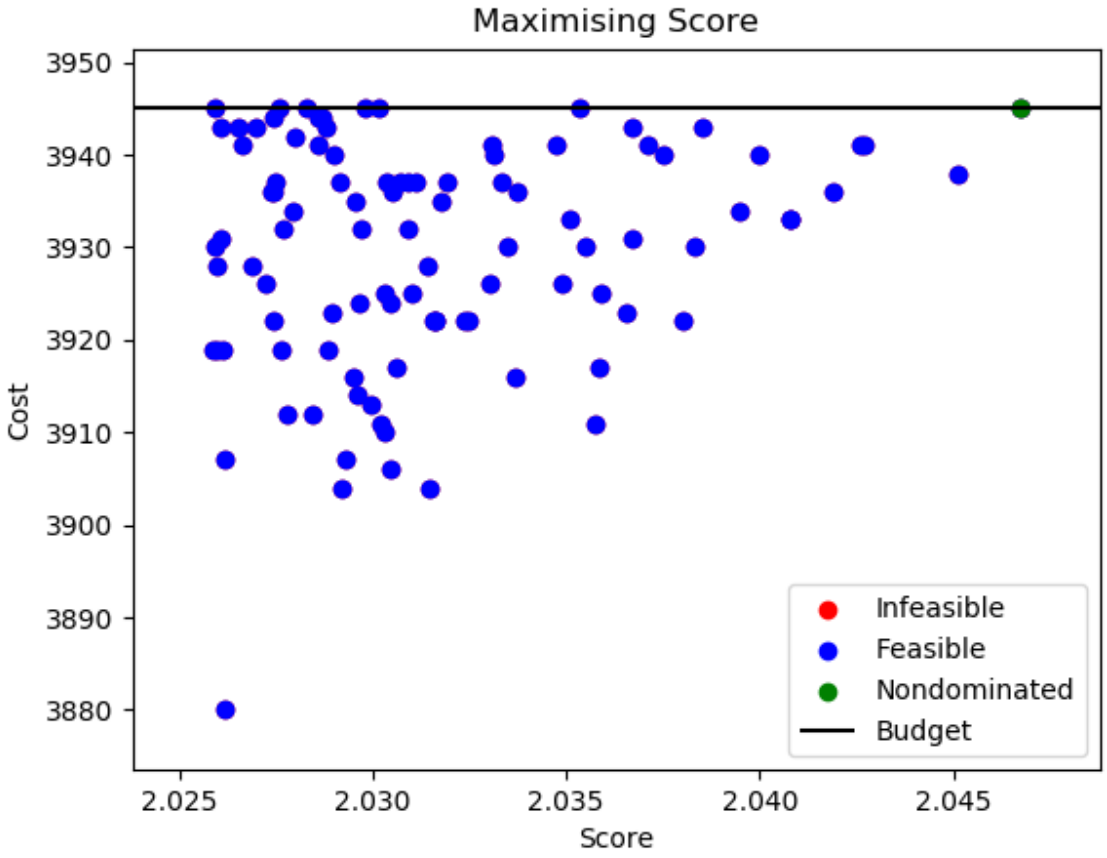

Figure 7: `sin('realistic-nrp/nrp-e1.txt', 0.3, 10000, 1)`.

The nondominated solution had score 2.05 and cost 3945.

Multi

I ran `mul('realistic-nrp/nrp-e1.txt', 0.3, 10000, 1)`, Figure 8.

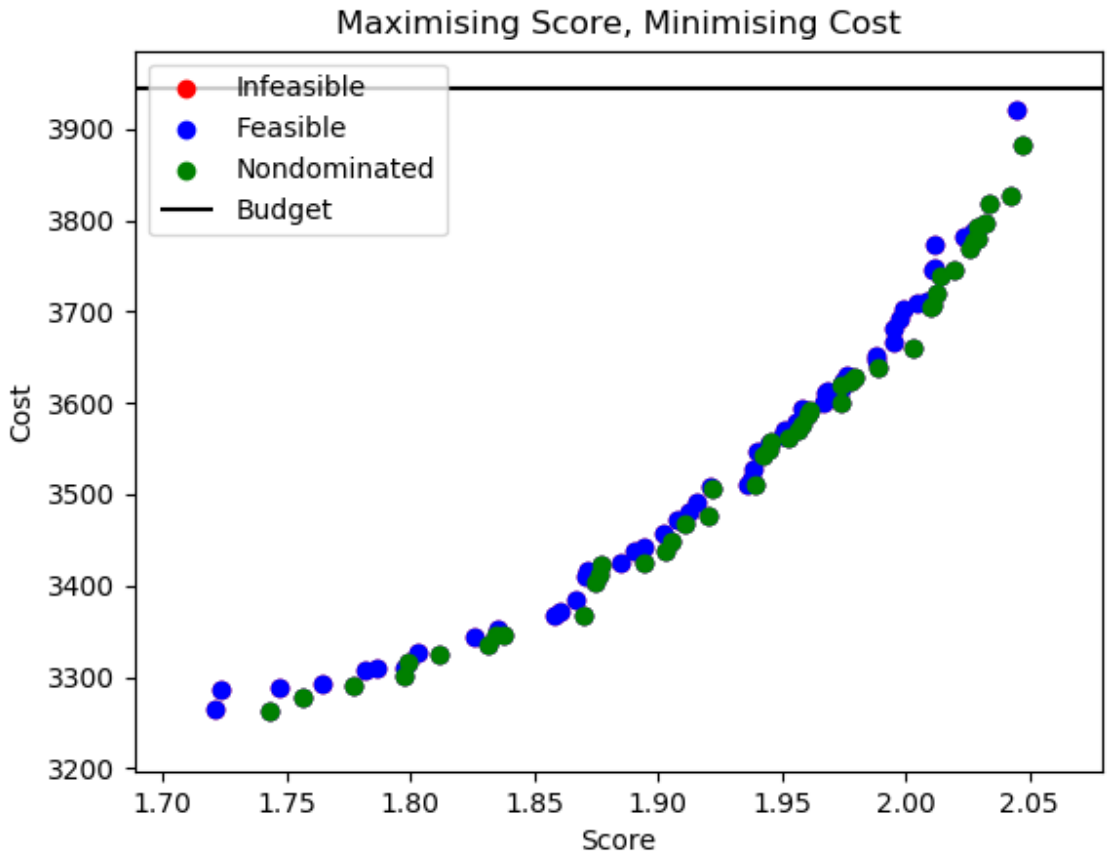

Figure 8: `mul('realistic-nrp/nrp-e1.txt', 0.3, 10000, 1).`

The nondominated solutions had scores from 1.74 to 2.05 and costs from 3262 to 3882.

Comparison

I ran `compare('realistic-nrp/nrp-e1.txt',0.3,0,0,1,10000,1,10000)`, Figure 9.

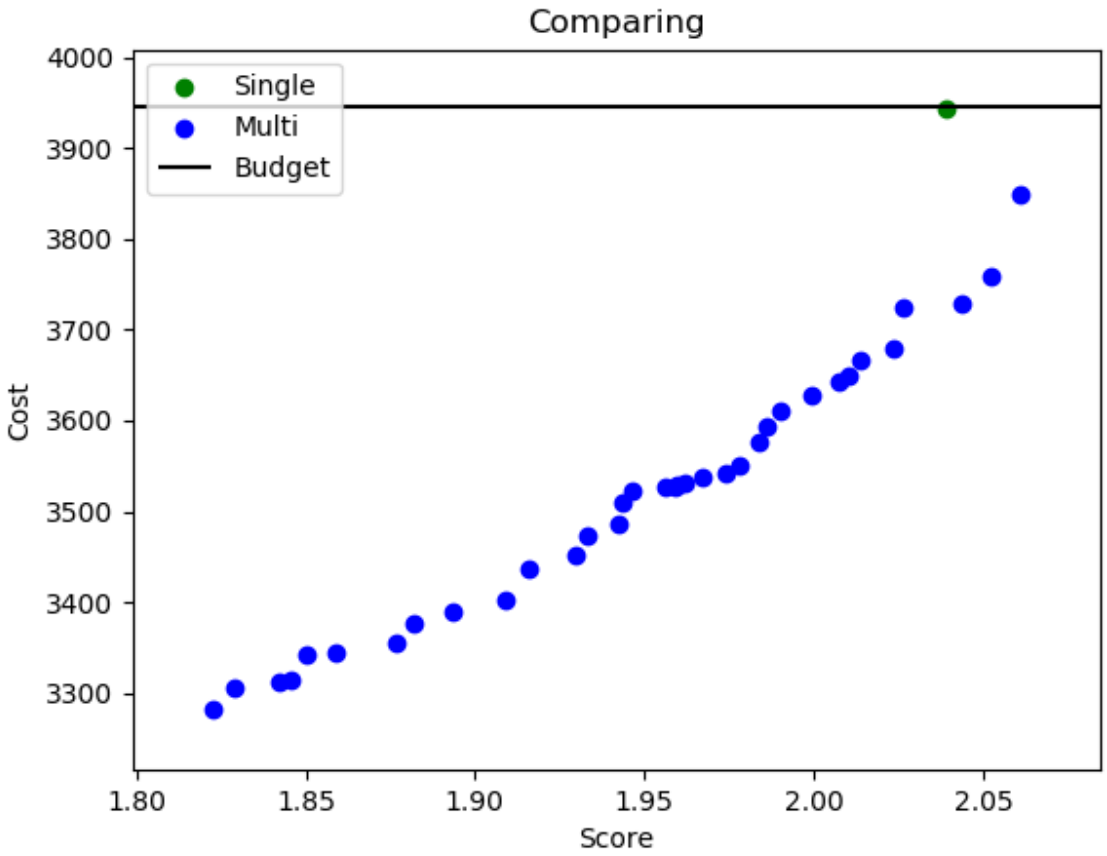

Figure 9: `compare('realistic-nrp/nrp-e1.txt',0.3,0,0,1,10000,1,10000).`

Again, not only was single's solution dominated by more than one of multi's, multi yielded a range of solutions.

Comparison of Results with Random Solutions

It is always a good idea to compare with random. Figure 10 is the plot returned by `compare('classic-nrp/nrp1.txt', 0.3, 1, 100, 1, 1000, 1, 1000)`. 100 random solutions are generated and those which are nondominated are compared with single and multi nondominated solutions.

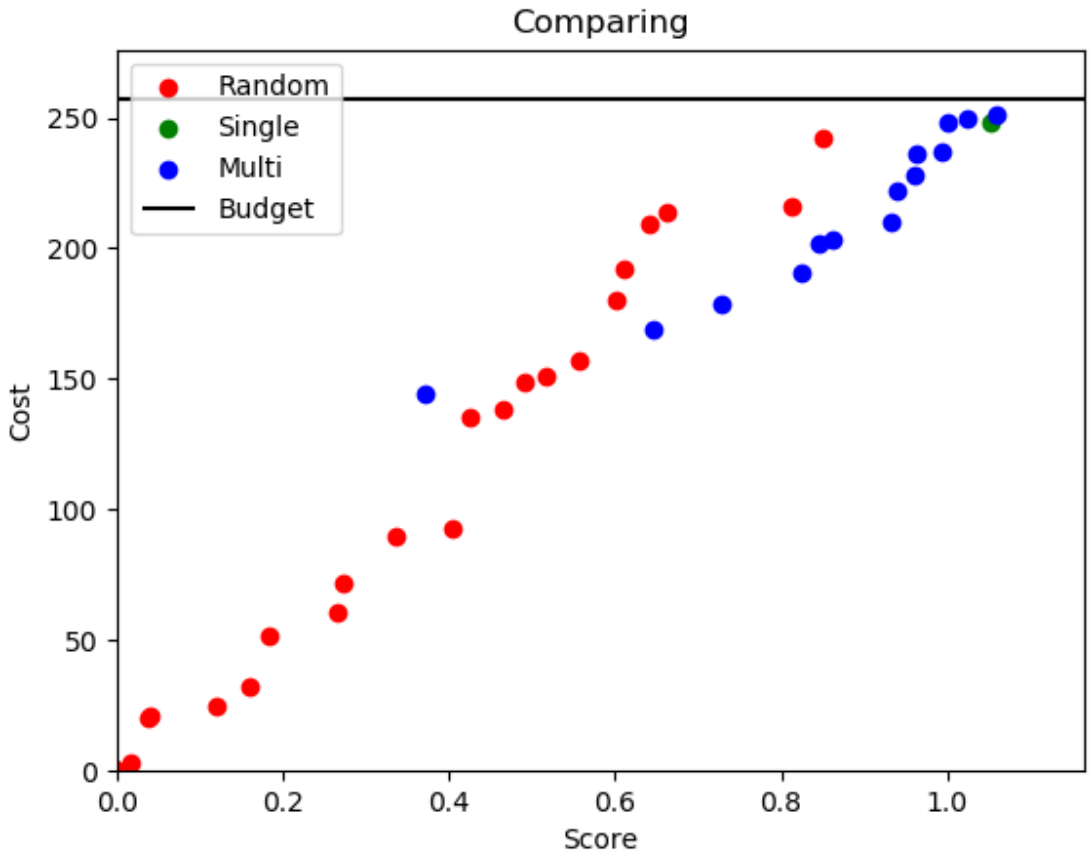

Figure 10: `compare('classic-nrp/nrp1.txt', 0.3, 1, 100, 1, 1000, 1, 1000).`

The visual comparison speaks for itself. Single's nondominated solution is extremely similar to one of multi's. Both are better than random's.