0117_nsga

January 17, 2019

1 Genetic Algorithms 2 - a multiple objective genetic algorithm (NSGA-II)

If you have not looked at our description of a more simple genetic algorithm, with a single objective, then we advise you to look at that first. The example here will assume some basic understand of genetic algorithms.

In our previous example of a genetic algorithm, we looked at a genetic algorithm that optimised a single parameter. But what if we have two or more objectives to optimise?

An example in healthcare modelling is in deciding which hospitals should provide a specialist service in order to 1) minimise travel time for patients who need an emergency hospital admission, while 2) ensuring the hospital has sufficient admissions to maintain expertise and a 24/7 service, and 3) and ensuring no hospital is over-loaded with too many admissions.

One option when considering multiple objectives is to combine different objectives into a single number. This requires standardising the values in some way and giving them weights for their importance. This can simplify the optimisation problem but may be sensitive to how individual objectives are weighted (if following this approach it will probably be sensible to re-run the algorithm using alternative weighting schemes).

1.1 Pareto fronts

The approach we will describe here takes an alternative approach and seeks to explicitly investigate the trade-off between different objectives. When one objective cannot be improved without the worsening of another objective we are on what is known as the 'Pareto front'. That is easy to visualise with two objectives, but Pareto fronts exist across any number of objectives - we are on the Pareto front when we cannot improve one objective without necessarily worsening at least one other objective (as shown by the red line in the figure below).

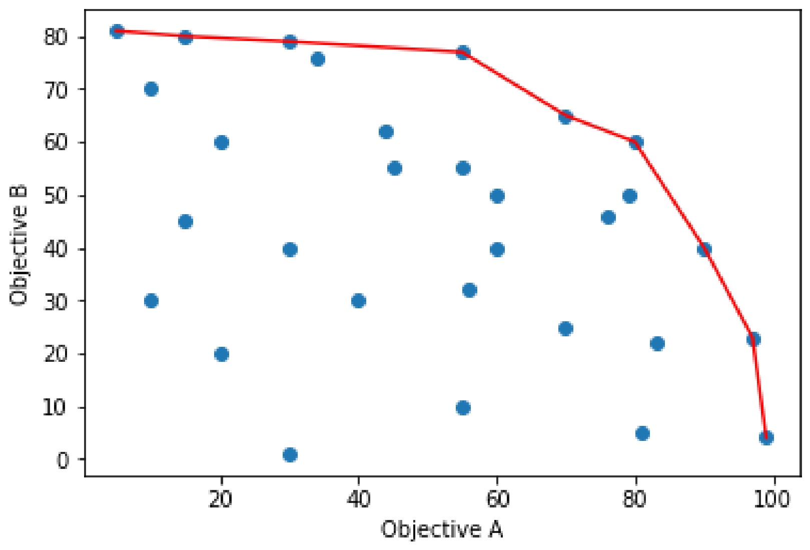
More detail on Parto fronts here:

https://pythonhealthcare.org/2018/09/27/93-exploring-the-best-possible-trade-off-between-competing-objectives-identifying-the-pareto-front/

1.2 NSGA-II

The algorithm implemented here is based on an algorithm called 'NSGA-II'. This was published by Deb et al.

Deb et al. (2002) A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation. 6, 182-197.

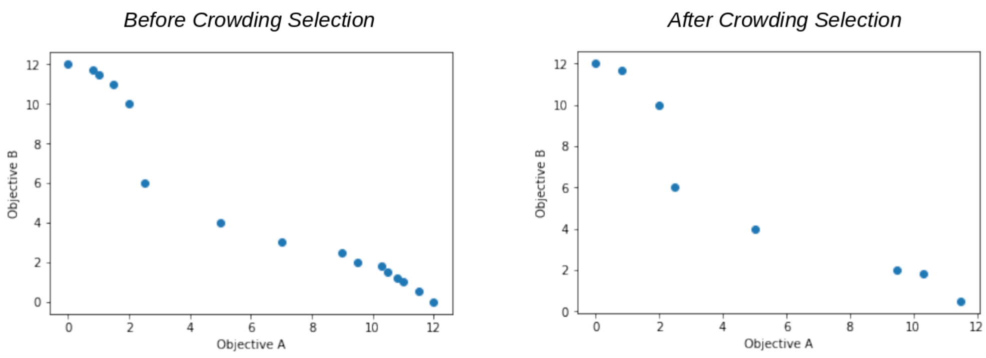


With NSGA-II we maintain a population of a given size (in the original paper that is fixed; in our implementation we define a range - the population must be larger than a minimum size, but not exceed a given maximum size.

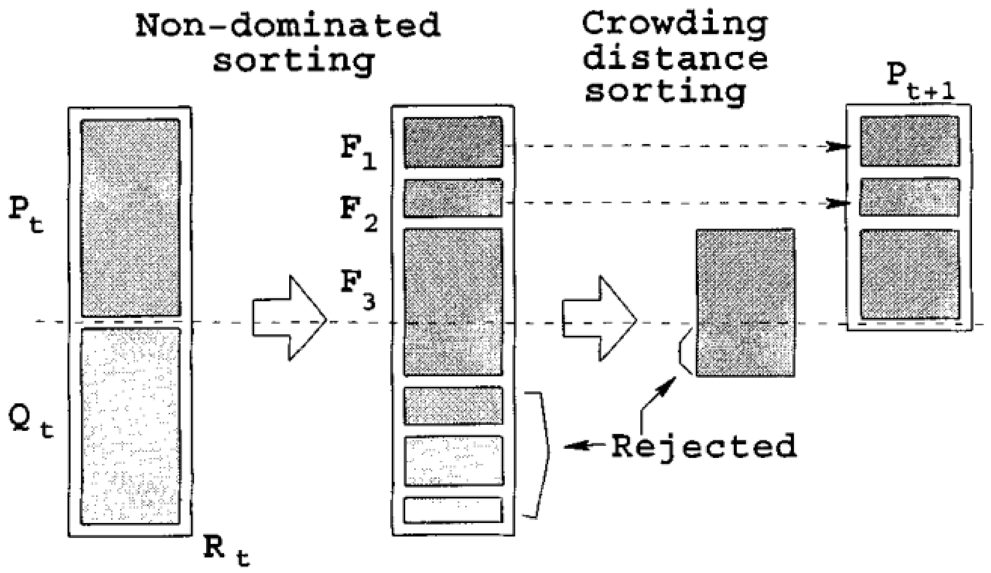
The key process steps of NSGA-II are:

1) Start with a random population of solutions (P), encoded in binary form (’chromosomes’)
2) Create a child population (Q) through crossover and mutation
3) Combine P & Q and score for all objectives
4) Identify the first Pareto front (F1); that is all solutions where there are no other solutions that are at least equally good in all objectives and better in at least one objective
5) If F1 is larger than the maximum permitted solution then reduce the size of F1 by ’crowding selection’ (see below).
6) If F1 is smaller than the required population size then repeat Pareto selection (after removal of selected already selected). This new set of solutions is F2. If the total number selected solutions is greater than the permitted maximum population size then reduce the size of just the latest selection (in this case F2) so that the number of all selected solutions is equal to the maximum permitted population size.
7) Repeat Pareto selection until the required population size is reached (and then reduce the last selected Pareto front by ’crowding selection’ as required to avoid exceeding the maximum permitted population size).
8) The selected Pareto fronts for the new population, P.
9) Repeat from (2) for the required number of generations or until some other ’stop’ criterion is reached.
10) Perform a final Pareto selection so that the final reported population are just those on the first Pareto front.

A minimum population size is required to maintain diversity of solutions. If we only selected the first Pareto front we may have a very small population which would lack the diversity required to reach good final solutions.



This process is shown diagrammatically below:



(from Deb et al., 2002)

1.3 Crowding distances/selection

When we are using a Pareto front to select solutions (as described here), all solutions are on the optimal front, that is in each solution there is no other solution that is at least as good in all scores, and better in at least one score. We therefore cannot rank solutions by performance. In order to select solutions, if we need to control the number of solutions we are generating we can use ‘crowding distances’. Crowding distances give a measure of closeness in performance to other solutions. The crowding distance is the average distance to its two neighbouring solutions. we will bias the selection of solutions to those with greater distances to neighbouring solutions.

We will use a ’tournament’ method to select between two individual solutions. The crowding distances are calculated for all solutions. Two solutions are picked at random and the one with the greater crowding distance will be selected (with the unselected solution being returned back). This process is repeated until the require number of solutions have been selected.

An example is shown below - by using crowding distances we have reduced the number of points in densely packed regions by more than in the sparsely populated areas.

More on croding distances here:

https://pythonhealthcare.org/2018/10/06/95-when-too-many-multi-objective-solutions-exist-selecting-solutions-based-on-crowding-distances/

2 The Code

We will break the code down into sections and functions.

To enale easier visualisation we will look to identify a Pareto population based on two competing objectives. But all the code will work on as many objectives as you wish (though it is not recommended to try to optimise more than five objectives).

2.1 Import required libraries

```
In [1]: import random as rn
        import numpy as np
        import matplotlib.pyplot as plt
        # For use in Jupyter notebooks only:
        % matplotlib inline
```

2.2 Create reference solutions

In this example we will pick the easiest example possible. We will create two reference chromosomes as 'ideal solutions'. This function may be used to create more references to mimic optimising against more than two objectives.

We will pick a chromosome length of 25 (this mimics solutions that would be coded as 25 binary 'genes').

In real life this binary representation would lead to better or worse solutions. We might, for example, use the binary notation to denote open/closed hospitals in a location problem. Or any number of binary digits might be used to encode the value of a variable.

```
In [2]: def create_reference_solutions(chromosome_length, solutions):
            """
            Function to create reference chromosomes that will mimic an ideal solution
            """
            references = np.zeros((solutions, chromosome_length))
            number_of_ones = int(chromosome_length / 2)

            for solution in range(solutions):
                # Build an array with an equal mix of zero and ones
                reference = np.zeros(chromosome_length)
                reference[0: number_of_ones] = 1

                # Shuffle the array to mix the zeros and ones
                np.random.shuffle(reference)
                references[solution,:] = reference

            return references
```

Show an example set of reference solutions:

```
In [3]: print (create_reference_solutions(25, 2))

[[0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 1. 1. 1.
  0.]
 [0. 1. 0. 1. 0. 0. 0. 1. 0. 0. 1. 0. 1. 1. 0. 1. 1. 1. 0. 0. 1. 0. 1. 1.
  0.]]
```

2.3 Evaluating solutions

Normally this section of the code would be more complex - it decodes the chromosome and applies it to a problem, and assesses the performance. For example we might use the chromosome to store whether hospitals are open/closed and then a piece of code will test travel distances for all patients to the open hospitals, and will also calculate the number of admissions to each hospital.

In our 'toy' example here we will simply calculate how many binary digits in each solution are the same as our two reference solutions (so we will have two scores for each solution).

We have two functions. The first will calculate the score ('fitness') against a single reference. The second will loop through all references (equivalent to all objectives) and call for the score/fitness to be calculated.

```
In [4]: def calculate_fitness(reference, population):
            """
            Calculate how many binary digits in each solution are the same as our
            reference solution.
            """
            # Create an array of True/False compared to reference
            identical_to_reference = population == reference
            # Sum number of genes that are identical to the reference
            fitness_scores = identical_to_reference.sum(axis=1)

            return fitness_scores

In [5]: def score_population(population, references):
            """
            Loop through all reference solutions and request score/fitness of
            population against that reference solution.
            """
            scores = np.zeros((population.shape[0], references.shape[0]))
            for i, reference in enumerate(references):
                scores[:,i] = calculate_fitness(reference, population)

            return scores
```

2.4 Calculate crowding and select a population based on crowding scores

We have two functions here. The first to calculate the crowding of a population, based on similarity of scores. The second is a Tournament selection method that uses those crowding scores to pick a given number of solutions from a population.

```
In [6]: def calculate_crowding(scores):
            """
            Crowding is based on a vector for each individual
            All scores are normalised between low and high. For any one score, all
            solutions are sorted in order low to high. Crowding for chromsome x
            for that score is the difference between the next highest and next
            lowest score. Total crowding value sums all crowding for all scores
            """

            population_size = len(scores[:, 0])
            number_of_scores = len(scores[0, :])

            # create crowding matrix of population (row) and score (column)
            crowding_matrix = np.zeros((population_size, number_of_scores))

            # normalise scores (ptp is max-min)
            normed_scores = (scores - scores.min(0)) / scores.ptp(0)

            # calculate crowding distance for each score in turn
            for col in range(number_of_scores):
                crowding = np.zeros(population_size)

                # end points have maximum crowding
                crowding[0] = 1
                crowding[population_size - 1] = 1

                # Sort each score (to calculate crowding between adjacent scores)
                sorted_scores = np.sort(normed_scores[:, col])

                sorted_scores_index = np.argsort(
                    normed_scores[:, col])

                # Calculate crowding distance for each individual
                crowding[1:population_size - 1] = \
                    (sorted_scores[2:population_size] -
                     sorted_scores[0:population_size - 2])

                # resort to orginal order (two steps)
                re_sort_order = np.argsort(sorted_scores_index)
                sorted_crowding = crowding[re_sort_order]

                # Record crowding distances
                crowding_matrix[:, col] = sorted_crowding

            # Sum crowding distances of each score
            crowding_distances = np.sum(crowding_matrix, axis=1)

            return crowding_distances
```

```
In [7]: def reduce_by_crowding(scores, number_to_select):
            """
            This function selects a number of solutions based on tournament of
            crowding distances. Two members of the population are picked at
            random. The one with the higher croding dostance is always picked
            """
            population_ids = np.arange(scores.shape[0])

            crowding_distances = calculate_crowding(scores)

            picked_population_ids = np.zeros((number_to_select))

            picked_scores = np.zeros((number_to_select, len(scores[0, :])))

            for i in range(number_to_select):

                population_size = population_ids.shape[0]

                fighter1ID = rn.randint(0, population_size - 1)

                fighter2ID = rn.randint(0, population_size - 1)

                # If fighter # 1 is better
                if crowding_distances[fighter1ID] >= crowding_distances[
                    fighter2ID]:

                    # add solution to picked solutions array
                    picked_population_ids[i] = population_ids[
                        fighter1ID]

                    # Add score to picked scores array
                    picked_scores[i, :] = scores[fighter1ID, :]

                    # remove selected solution from available solutions
                    population_ids = np.delete(population_ids, (fighter1ID),
                                               axis=0)

                    scores = np.delete(scores, (fighter1ID), axis=0)

                    crowding_distances = np.delete(crowding_distances, (fighter1ID),
                                                   axis=0)
                else:
                    picked_population_ids[i] = population_ids[fighter2ID]

                    picked_scores[i, :] = scores[fighter2ID, :]

                    population_ids = np.delete(population_ids, (fighter2ID), axis=0)
                    scores = np.delete(scores, (fighter2ID), axis=0)

                    crowding_distances = np.delete(
                        crowding_distances, (fighter2ID), axis=0)

            # Convert to integer
            picked_population_ids = np.asarray(picked_population_ids, dtype=int)

            return (picked_population_ids)
```

2.5 Pareto selection

Pareto selection involves two functions. The first will select a single Pareto front. The second will, as necessary, repeat Pareto front selection to build a population within defined size limits, and, as necessary, will reduce a Pareto front by applying crowding selection.

```
In [8]: def identify_pareto(scores, population_ids):
            """
            Identifies a single Pareto front, and returns the population IDs of
            the selected solutions.
            """
            population_size = scores.shape[0]
            # Create a starting list of items on the Pareto front
            # All items start off as being labelled as on the Parteo front
            pareto_front = np.ones(population_size, dtype=bool)
            # Loop through each item. This will then be compared with all other items
            for i in range(population_size):
                # Loop through all other items
                for j in range(population_size):
                    # Check if our 'i' pint is dominated by out 'j' point
                    if all(scores[j] >= scores[i]) and any(scores[j] > scores[i]):
                        # j dominates i. Label 'i' point as not on Pareto front
                        pareto_front[i] = 0
                        # Stop further comparisons with 'i' (no more comparisons needed)
                        break
            # Return ids of scenarios on pareto front
            return population_ids[pareto_front]
```

```
In [9]: def build_pareto_population(
                population, scores, minimum_population_size, maximum_population_size):
            """
            As necessary repeats Pareto front selection to build a population within
            defined size limits. Will reduce a Pareto front by applying crowding
            selection as necessary.
            """
            unselected_population_ids = np.arange(population.shape[0])
            all_population_ids = np.arange(population.shape[0])
            pareto_front = []
```

```
        while len(pareto_front) < minimum_population_size:
            temp_pareto_front = identify_pareto(
                    scores[unselected_population_ids, :], unselected_population_ids)

            # Check size of total parteo front.
            # If larger than maximum size reduce new pareto front by crowding
            combined_pareto_size = len(pareto_front) + len(temp_pareto_front)
            if combined_pareto_size > maximum_population_size:
                number_to_select = combined_pareto_size - maximum_population_size
                selected_individuals = (reduce_by_crowding(
                        scores[temp_pareto_front], number_to_select))
                temp_pareto_front = temp_pareto_front[selected_individuals]

            # Add latest pareto front to full Pareto front
            pareto_front = np.hstack((pareto_front, temp_pareto_front))


            # Update unslected population ID by using sets to find IDs in all
            # ids that are not in the selected front
            unselected_set = set(all_population_ids) - set(pareto_front)
            unselected_population_ids = np.array(list(unselected_set))

        population = population[pareto_front.astype(int)]
        return population
```

2.6 Population functions

There are four functions concerning the population

1) Create a random population
2) Breed by crossover - two children produced from two parents
3) Randomly mutate population (small probability that any given gene in population will switch between 1/0 - applied to the child population)
4) Breed population -Create child population by repeatedly calling breeding function (two parents producing two children), applying genetic mutation to the child population, combining parent and child population, and removing duplicate chromosomes.

```
In [10]: def create_population(individuals, chromosome_length):
             """
             Create random population with given number of individuals and chroosome
             length.
             """
             # Set up an initial array of all zeros
             population = np.zeros((individuals, chromosome_length))
             # Loop through each row (individual)
             for i in range(individuals):
                 # Choose a random number of ones to create
                 ones = rn.randint(0, chromosome_length)
```

```
            # Change the required number of zeros to ones
            population[i, 0:ones] = 1
            # Sfuffle row
            np.random.shuffle(population[i])

        return population
```

```
In [11]: def breed_by_crossover(parent_1, parent_2):
             """
             Combine two parent chromsomes by crossover to produce two children.
             """
             # Get length of chromosome
             chromosome_length = len(parent_1)

             # Pick crossover point, avoding ends of chromsome
             crossover_point = rn.randint(1,chromosome_length-1)

             # Create children. np.hstack joins two arrays
             child_1 = np.hstack((parent_1[0:crossover_point],
                                 parent_2[crossover_point:]))

             child_2 = np.hstack((parent_2[0:crossover_point],
                                 parent_1[crossover_point:]))

             # Return children
             return child_1, child_2
```

```
In [12]: def randomly_mutate_population(population, mutation_probability):
             """
             Randomly mutate population with a given individual gene mutation
             probability. Individual gene may switch between 0/1.
             """
             # Apply random mutation
             random_mutation_array = np.random.random(size=(population.shape))

             random_mutation_boolean = \
                 random_mutation_array <= mutation_probability

             population[random_mutation_boolean] = \
             np.logical_not(population[random_mutation_boolean])

             # Return mutation population
             return population
```

```
In [13]: def breed_population(population):
             """
             Create child population by repetedly calling breeding function (two parents
             producing two children), applying genetic mutation to the child population,
             combining parent and child population, and removing duplicatee chromosomes.
             """
             # Create an empty list for new population
             new_population = []
             population_size = population.shape[0]
             # Create new popualtion generating two children at a time
             for i in range(int(population_size/2)):
                 parent_1 = population[rn.randint(0, population_size-1)]
                 parent_2 = population[rn.randint(0, population_size-1)]
                 child_1, child_2 = breed_by_crossover(parent_1, parent_2)
                 new_population.append(child_1)
                 new_population.append(child_2)

             # Add the child population to the parent population
             # In this method we allow parents and children to compete to be kept
             population = np.vstack((population, np.array(new_population)))
             population = np.unique(population, axis=0)

             return population
```

3 Main algorithm code

Now let's put it all together!

```
In [14]: # Set general parameters
         chromosome_length = 50
         starting_population_size = 5000
         maximum_generation = 250
         minimum_population_size = 500
         maximum_population_size = 1000

         # Create two reference solutions
         # (this is used just to illustrate GAs)
         references = create_reference_solutions(chromosome_length, 2)

         # Create starting population
         population = create_population(
                 starting_population_size, chromosome_length)

         # Loop through the generations of genetic algorithm

         for generation in range(maximum_generation):
             if generation %10 ==0:
                 print ('Generation (out of %i): %i '%(maximum_generation, generation))

             # Breed
             population = breed_population(population)
```

```
        # Score population
        scores = score_population(population, references)

        # Build pareto front
        population = build_pareto_population(
                population, scores, minimum_population_size, maximum_population_size)


    # Get final pareto front
    scores = score_population(population, references)
    population_ids = np.arange(population.shape[0]).astype(int)
    pareto_front = identify_pareto(scores, population_ids)
    population = population[pareto_front, :]
    scores = scores[pareto_front]
```

```
Generation (out of 250): 0
Generation (out of 250): 10
Generation (out of 250): 20
Generation (out of 250): 30
Generation (out of 250): 40
Generation (out of 250): 50
Generation (out of 250): 60
Generation (out of 250): 70
Generation (out of 250): 80
Generation (out of 250): 90
Generation (out of 250): 100
Generation (out of 250): 110
Generation (out of 250): 120
Generation (out of 250): 130
Generation (out of 250): 140
Generation (out of 250): 150
Generation (out of 250): 160
Generation (out of 250): 170
Generation (out of 250): 180
Generation (out of 250): 190
Generation (out of 250): 200
Generation (out of 250): 210
Generation (out of 250): 220
Generation (out of 250): 230
Generation (out of 250): 240
```

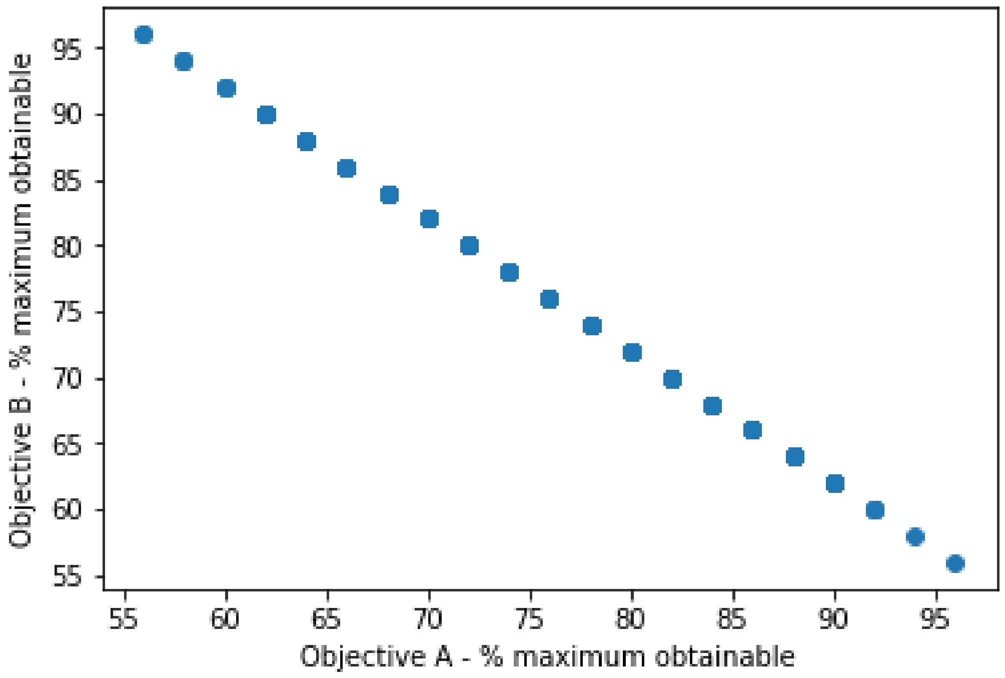
4 Plot final Pareto front

This is the only part of the code that works only for two objectives.

You will see in this example we have a well-described trade-off between achieving the two objectives. With the population sizes used and the number of generations we achieved 96% maximum score on each objective (a larger population and/or more generations would get us to 100% on this quite simple example, though Pareto algorithms should not be expected to find the perfect solution for all objectives, but rather they should identify good solutions).

```
In [15]: # Plot Pareto front (for two scores only)
         x = scores[:, 0]/chromosome_length*100
         y = scores[:, 1]/chromosome_length*100
         plt.xlabel('Objective A - % maximum obtainable')
         plt.ylabel('Objective B - % maximum obtainable')

         plt.scatter(x,y)
         plt.savefig('pareto.png')
         plt.show()
```


Objective B - % maximum obtainable
Objective A - % maximum obtainable