0095_crowding

October 7, 2018

1 When too many multi-objective solutions exist: selecting solutions based on crowding distances

Sometimes in multi-objective algorithms we need to thin out the number of solutions we have.

When we are using a Pareto front to select solutions, 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.

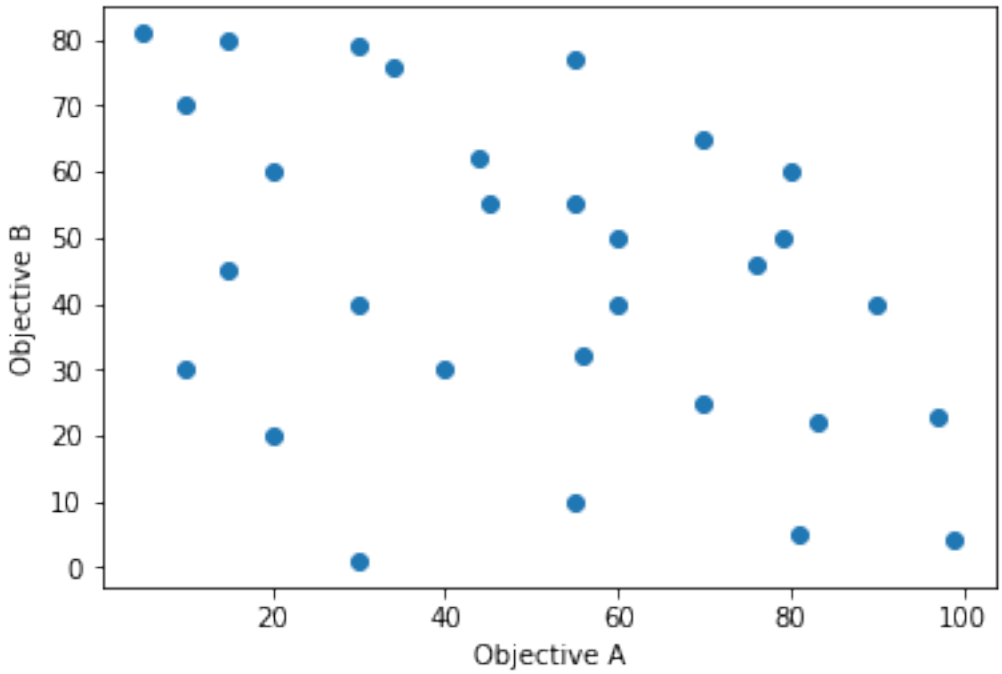
Here we will look at an array of algorithm scores that have two dimensions for each solution (two different scores we wish to optimise). The crowding distance code provided will however work for any number of dimensions/objectives.

Let's make an array of scores - all of which would be on a Pareto front.

```
In [16]: import numpy as np

         test_array = np.array([[12, 0],
                                [11.5, 0.5],
                                [11, 1],
                                [10.8, 1.2],
                                [10.5, 1.5],
                                [10.3, 1.8],
                                [9.5, 2],
                                [9, 2.5],
                                [7, 3],
                                [5, 4],
                                [2.5, 6],
                                [2, 10],
                                [1.5, 11],
                                [1, 11.5],
                                [0.8, 11.7],
                                [0, 12]])
```

We can plot these points. We have cluster of points at the ends and but a more sparse population of points in the middle (though the two end points will have a high crowding distance, because they have no neighbouring points on one side of them).

```
In [17]: import matplotlib.pyplot as plt
         %matplotlib inline

         x = test_array[:, 0]
         y = test_array[:, 1]
         plt.xlabel('Objective A')
         plt.ylabel('Objective B')

         plt.scatter(x,y)
         plt.show()
```



1.1 Code for calculating crowding

Each dimension is normalised between low and high to prevent crowding distances being dominated by scores on larger scales. For each dimension we sort all the scores from low to high. The crowding distance in one dimension for any individual is the distance between the next lowest and next highest score in that dimension. End points have a score of one (which is the range of normalised scores). Each individual will have a crowding score for each dimension (objective). These are summed to give the final crowding score.

```
In [18]: import numpy as np

         def calculate_crowding(scores):
             # Crowding is based on a vector for each individual
             # All dimension is normalised between low and high. For any one dimension, 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
```

So, calculating crowding distances for our original array:

```
In [19]: crowding_distance = calculate_crowding(test_array)

         print(crowding_distance)
```

```
[2.         0.16666667 0.11666667 0.08333333 0.09166667 0.125
 0.16666667 0.29166667 0.45833333 0.625      0.75       0.5
 0.20833333 0.11666667 0.125      2.        ]
```

We can see our end points have maximum crowding distance (as they have no neighbour on one side). Points close to the end have low crowding score (i.e. close proximity to other solutions), and points in the middle have high crowding scores.

We may then use a selection method such as Tournament to choose between any two (or more) individuals when selecting solutions to take forward in an iterative algorithm (such as a genetic algorithm).

1.2 Selecting individuals based on crowding score

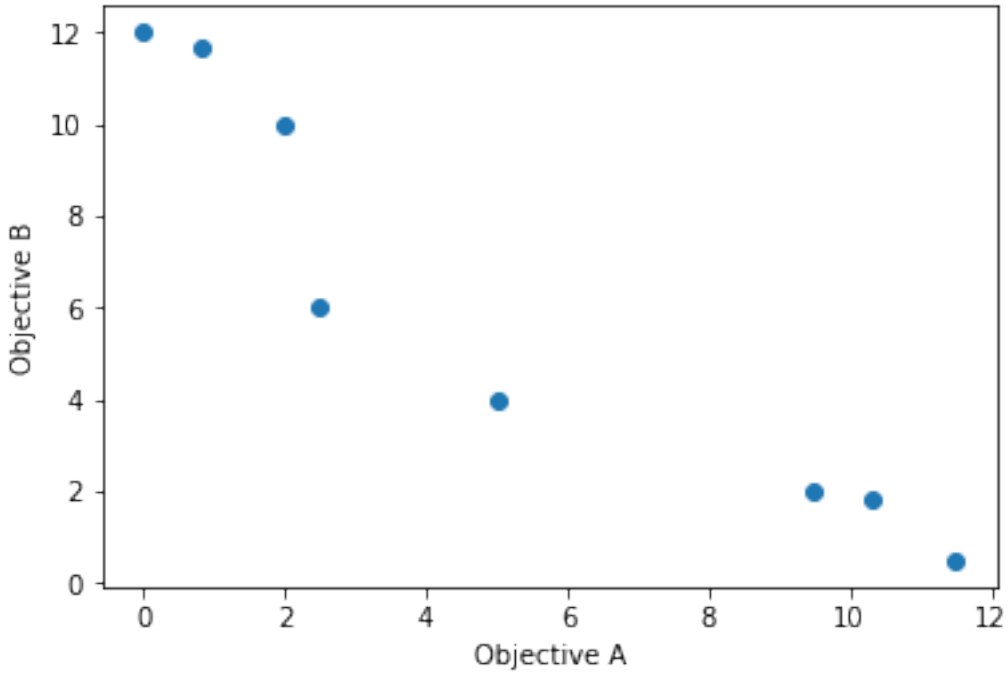
The code below uses Tournament selection to pick a given number of solutions based on crowding scores. In each iteration of a loop two individuals solutions are picked at random. The one with the highest crowding score (greatest distance from neighbouring solutions) is selected and removed from the population that may be picked from.

```
In [20]: import numpy as np
         import random as rn

         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)
```

Let's use the function to pick eight solutions from our original array (half of our orginal solutions), and plot them. This method is based on random picks, so the results will vary between runs, but you should see your population is 'thinned out' more where points are closely crowded.

```
In [24]: selected_solutions_ids = reduce_by_crowding(test_array, 8)
         selected_solutions = test_array[selected_solutions_ids]

         x = selected_solutions[:, 0]
         y = selected_solutions[:, 1]
         plt.xlabel('Objective A')
         plt.ylabel('Objective B')

         plt.scatter(x,y)
         plt.show()
```



In [ ]: