# NSGA-II

## Replacement operator

replacement

Gevol.evolution.evoperator.replacement

## Description

A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II introduced by Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal and T. Meyarivan in 2002. The algorithm selects the best individuals regarding many objective functions. Each individual is assigned to proper Pareto front, then the individuals are sorted based on the Pareto front and a crowding distance. Crowding distance describe how different individual is in comparison to other individuals in the same Pareto front.

Pareto front – individual A is in lower (more optimal) Pareto front than individual B only if score for every objective function of individual A is lower than for individual B.

$$I_A <_{front} I_B, if \bigwedge_F F(I_A) < F(I_B)$$

$$I_A =_{front} I_B, if \bigvee_F F(I_A) \leq F(I_B) \wedge \bigvee_F F(I_B) \leq F(I_A)$$

Where

- $I_A$ and $I_B$ – individual A and individual B
- F – all objective function
- $<_{front}$ – belongs to lower (better) Pareto front
- $=_{front}$ – belongs to the same Pareto front

Crowding distance – all individuals in the same Pareto front are sorted based on their crowding distance. The crowding distance describe how different is the individual than his neighbors. If the individual has only one neighbor (the boundary one), it gets maximum (the best) value. The more different the individual is, the better value it gets.

The formula below should be calculated for every objective function. The order is important, so at first step all individuals in the Pareto front should be sorted by current objective function.

$$CrowdingDistance(I_A) = \begin{cases} \infty, & A = 1 \vee A = n \\ CD(I_A) + \dfrac{f(I_{A+1}) - f(I_{A-1})}{f_{max} - f_{min}}, & A \in (1, n) \end{cases}$$

Where

- CD – Crowding Distance (recurrence)
- n – number of individuals in the Pareto front
- $f(I_{A+1})$, $f(I_{A-1})$ – neighbors for the individual A regarding the value of current objective function
- $f_{max}$, $f_{min}$ – maximum and minimum value for current objective function gathered by these individuals

## Parameters

No parameters.

## Pseudocode

```
CalculateParetoFronts
    For each individual i
        For each individual j
            If i dominate j
                i.dominatedIndividuals += j
            If j dominate i
                i.dominatingIndividuals += 1
        If i.dominatingIndividuals == 0
            i = first Pareto front
    //calculate next Pareto fronts
    For each Pareto front p
        For each individual i in p
            For each dominated individual j in i.dominatedIndividuals
                j.dominatingIndividuals -= 1
            If j.dominatingIndividuals == 0
                j = next Pareto front

CalculateCrowdingDistance CD
    For each Pareto front
        For each objective function o
            Sort individuals by o
            CD for first individual = infinity
            CD for last individual = infinity
            CD for other individuals = CD(I)=(Inext-Iprev)/(Fmax-Fmin)

P = sort(P) by Pareto rank asc, crowding distance desc
P' = get first x individuals from P
```

## Implementation details

A special class `NSGA2Structure` has been created for further calculations. At first step children and parents population is merged into the new class.

Calculation of Pareto ranks is based on the algorithm. Last part of the function `CalculateParetoRank()` triggers some kind of recurrence. In the set `HashSet<int> paretoFront` are individuals' indexes of the best Pareto front. `int maxParetoRank = 0` indicates the current Pareto rank. It is increased on every execution.

```
//calculate next Pareto fronts
while (paretoFront.Count != 0)
{
    //add number of individuals in this Pareto front to the list
    paretoFrontCounter.Add(paretoFront.Count);
    paretoFront = findNextParetoFront(++maxParetoRank, paretoFront);
}
```

This implementation will return n the best individuals. n = `parentsPopulation.Count`. So there is no need to calculate the crowding distance for every individual. To remember how many individuals is in every Pareto front:

```
paretoFrontCounter.Add(paretoFront.Count);
```

Check how many Pareto fronts will be used to fulfill new population. The crowding distance will be calculated for only Pareto fronts <= `int index`.

```
//calculate how many Pareto fronts we need to fulfill new population
int sumOfIndividuals = 0;
int index = 0;
while (sumOfIndividuals < parentsPopulation.Count)
{
    sumOfIndividuals += paretoFrontCounter[index++];
}
```

Crowding distance for each Pareto front is calculated separately.

```
for(int i = 0; i <= index; i++)
```

The formula for crowding distance is executed separately for each objective function. Sorting is made via the LINQ language. The `int s` needs to be passed to the function to know for each objective function it executes the calculation.

```
//for each objective function
for (int s = 0; s < unionStructurePopulation[0].Individual.Score.Count; s++)
{
    CalculateCrowdingDistance(
        from individual in unionStructurePopulation
        where individual.ParetoRank == i
        orderby individual.Individual.Score[s]
        select individual.index
    , s);
}
```

Boundary individuals should have the infinity value. Here we use `Double.MaxValue` as the infinity. For one objective function the individual can be boundary, for another not. In the result it should still get the infinity value. To avoid overflow errors calculations are done only for non-infinity individuals.

```
if(unionStructurePopulation[frontIndexes.ElementAt<int>(i)].CrowdingDistance <
Double.MaxValue)
```

Last step is to select individuals with the best Pareto fronts. It is possible that not all individuals will be taken from the worst front that is going to be taken. From that front it takes individuals with the highest crowding distance value.

```
IEnumerable<int> selectedIndividuals = (
    from individual in unionStructurePopulation
    orderby individual.ParetoRank, individual.CrowdingDistance descending
    select individual.index).Take<int>(parentsPopulation.Count
);
```

`selectedIndividuals` is a list of indexes taken from the special class `NSGA2Structure`. The indexes do not corresponds to the real index in `parentsPopulation` as these individuals were taken after `childrenPopulation`. It is required to calculate the proper index of individual and add it to the new population.

```
foreach(int selectedIndex in selectedIndividuals)
{
    if (selectedIndex >= childrenPopulation.Count)
       newPopulation.Add(parentsPopulation[selectedIndex - childrenPopulation.Count]);
    else
       newPopulation.Add(childrenPopulation[selectedIndex]);
}
return newPopulation;
```

## References

1. “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II”, IEEE transactions on evolutionary computation, vol. 6 no. 2, April 2002