Assessment 02 - Solving Knapsack with NSGA-II

This script is basically the script used for the NSGA-II assignment02 with some small modifications. The main differences are that the fitness computation now needs to compute different things than before.

Basically the value of the items can be taken as is (sum of all item values), the weight has to be negated somehow, because the weight should be minimized. Therefore I took the maximum possible weight and subtracted the actual weight.

Parameter initialization

```
clear;
p.nGenerations = 100;
p.nPopulation = 30;
p.nOffspring = 30;
p.crossoverProbability = 0.9;
```

Read data

Reads the data and stores weight and values to variables

```
data = importdata('items.csv');
p.weights = data.data(:,1);
p.values = data.data(:,2);
p.nGenes = length(p.weights);

% Read names without header
names = data.textdata(2:length(data.textdata),1);

p.mutationProbability = 3 / p.nGenes;
```

Initialize population

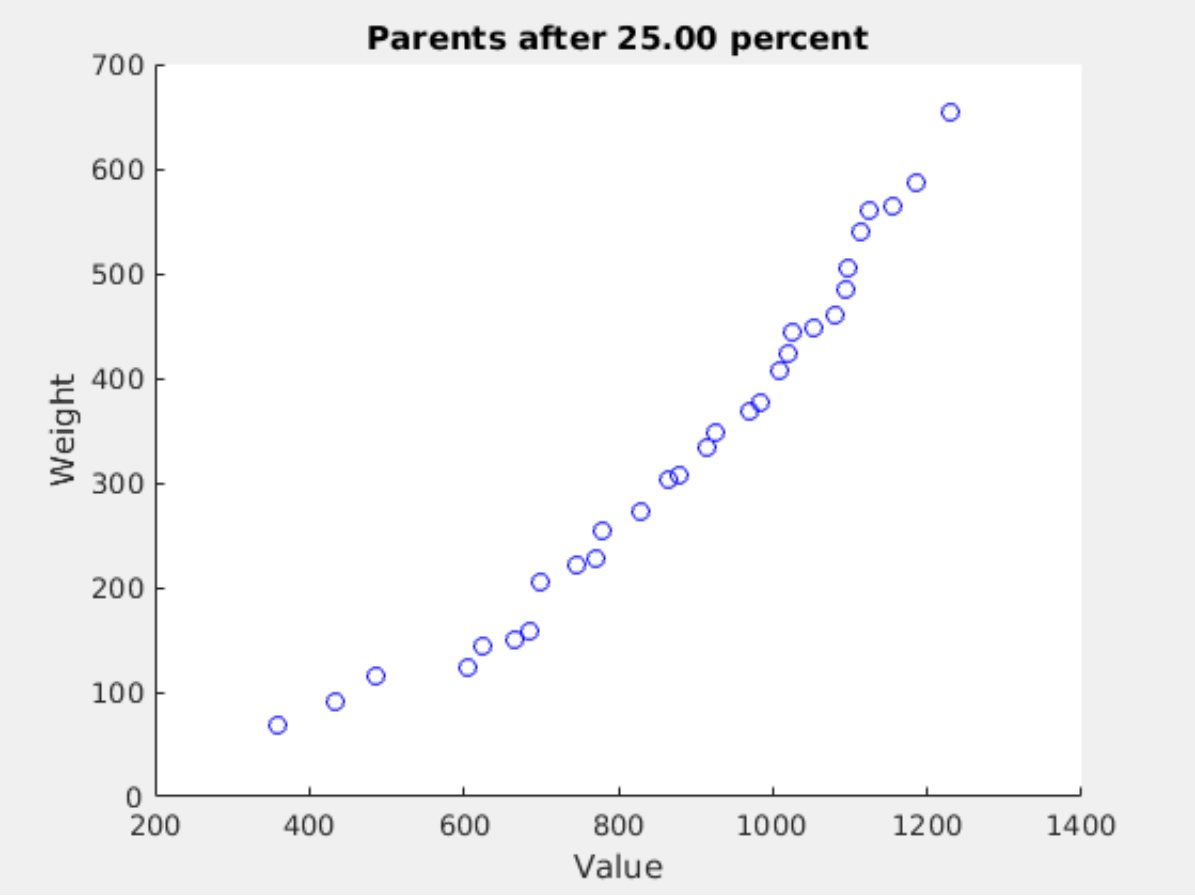
```
p.population = randi([0 1], p.nPopulation, p.nGenes);
```

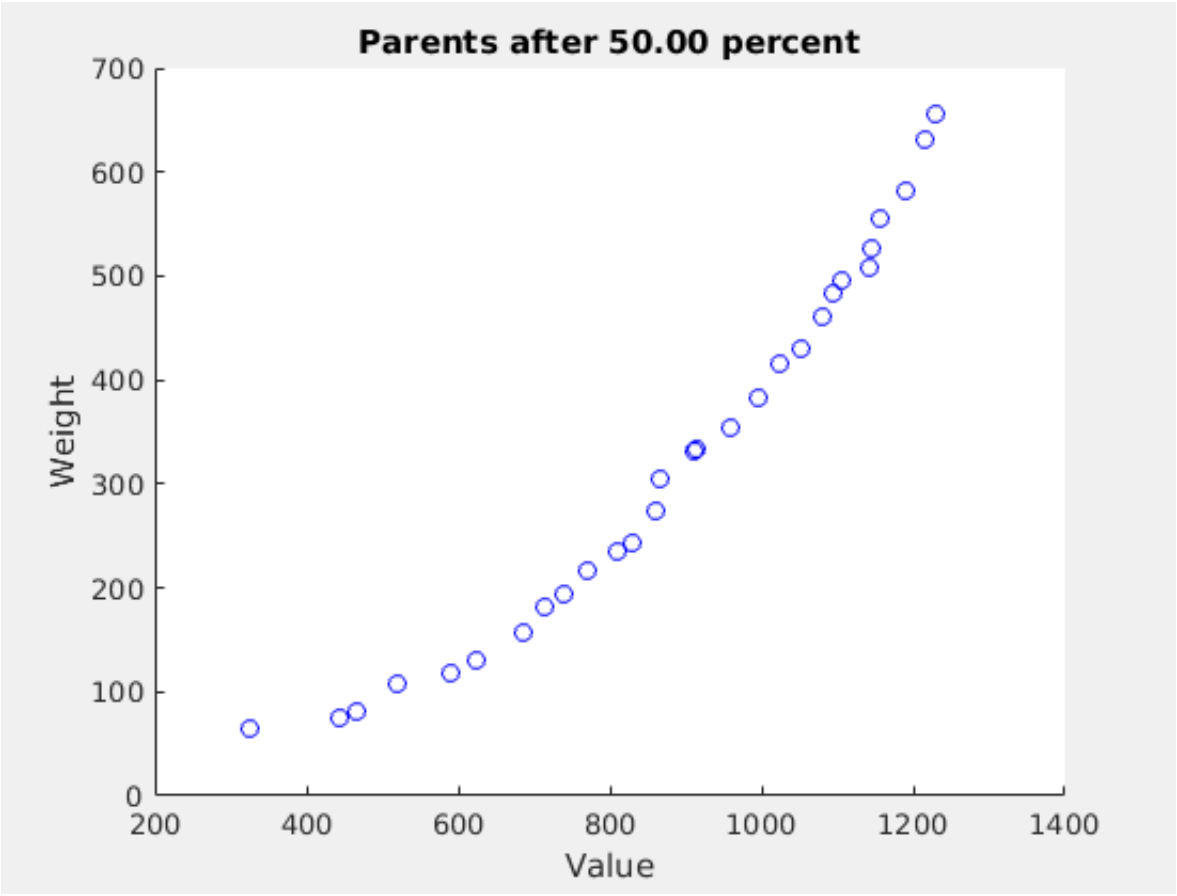
Evolution Loop

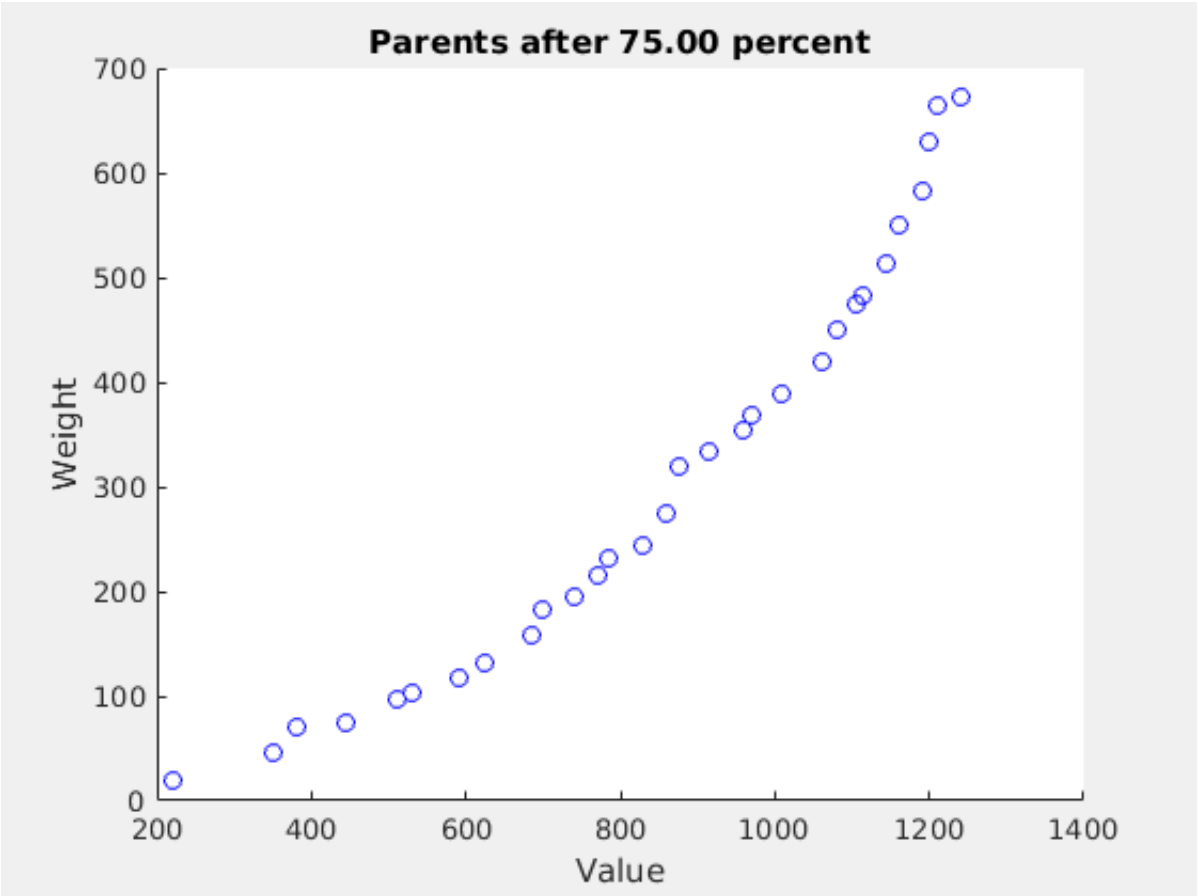
```
visualizationStep = 1;
for generation=1:p.nGenerations

    % Evaluation
    %   Computes the weight and the value of the chosen items. Value is
    %   stored as is, weight fitness value is computed as 803 -
    %   actualWeight, where 803 is the maximum weight. This way it is
    %   possible to maximize the weight-value.
    fitness = computeFitness(p.population, p.values, p.weights);
    p.value = fitness.value;
    p.weight = fitness.weight;

    % Visualization
    %   Plots the parents every generation and at 25, 50. 75 and 100
```

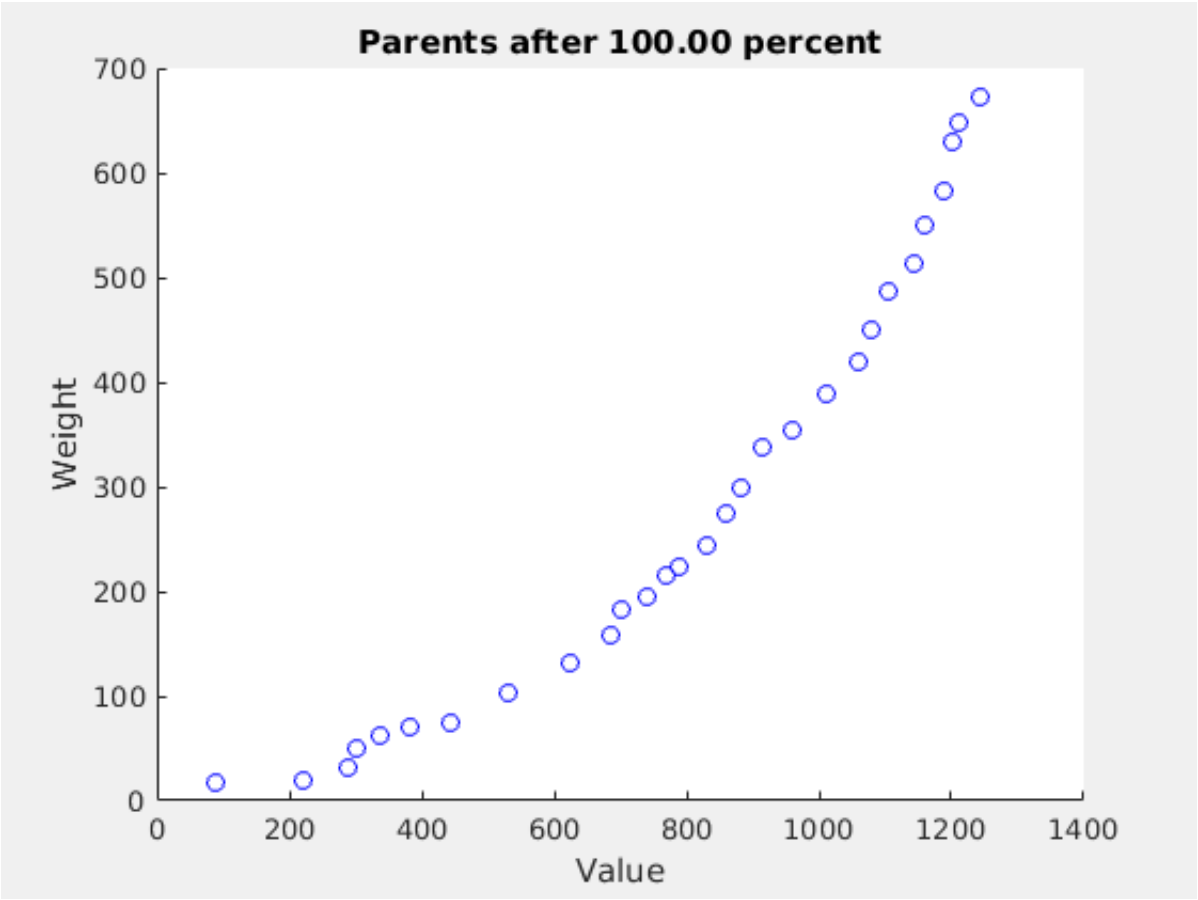
```
    %   percent of maximum generations.
    if generation >= p.nGenerations*(0.25 * visualizationStep)
        visualizationStep = visualizationStep + 1;
        figure(visualizationStep);clf;hold on;
        title(sprintf('Parents after %.2f percent',...
            (visualizationStep - 1) * 25));
        plot(p.value, 803 - p.weight,'bo');
        xlabel('Value');
        ylabel('Weight');
%         axis([0,p.nGenes,0,p.nGenes]);
    end

    figure(1);clf;hold on;
    subplot(1,2,1);
    plot(p.value, 803 - p.weight,'bo');
    title(sprintf('Parents at generation %d', generation));
    xlabel('Value');
    ylabel('Weight');
%     axis([0,p.nGenes,0,p.nGenes]);

    % Create and evaluate offspring
    %   Generates offspring for the given population by using tournament
    %   selection, mutation and crossover as has been used in oneMax.
    %   Elitism is not used because the pareto fronts takes care of the
    %   best individuals.
    offspring = generateOffspring(p);
    fitness = computeFitness(offspring, p.values, p.weights);
    p.value = [p.value fitness.value];
    p.weight = [p.weight fitness.weight];
    p.population = [p.population;offspring];

    %   Computes the domination sets and counts for every individual and
    %   assigns each individual to the pareto front it belongs to.
    %   This is done by checking for each individual those individuals it
    %   dominates and is dominated by. If there is no dominating
    %   individual, the individual goes into the front.
    %   For every next front the previous fronts get subtracted from each
    %   remaining individual's domination counter. Again, if one of the
    %   remaining individual's counter decreases to zero, it belongs to the
    %   next front.
    %   More details can be found in dominationSort.m
    paretoFronts = dominationSort(p);

    % Choose individuals for next generation using pareto fronts.
    %   As long as the next front fits into next generation, add...
    nextPopulation = [];
    i = 1;
    while size(nextPopulation,1) + length(paretoFronts{i}) <=...
            p.nPopulation
        indizes = paretoFronts{i};
        paretoElements = p.population(indizes,:);
        nextPopulation = [nextPopulation; paretoElements];
        i = i + 1;
    end

    %   If next generation is not full yet,...
    if size(nextPopulation,1) ~= p.nPopulation
        %   ...fill it with best spreaded individuals from next front.

        %   Given a population and its values for leadingZeros and
        %   tailingOnes, this function computes the crowding distance for
```

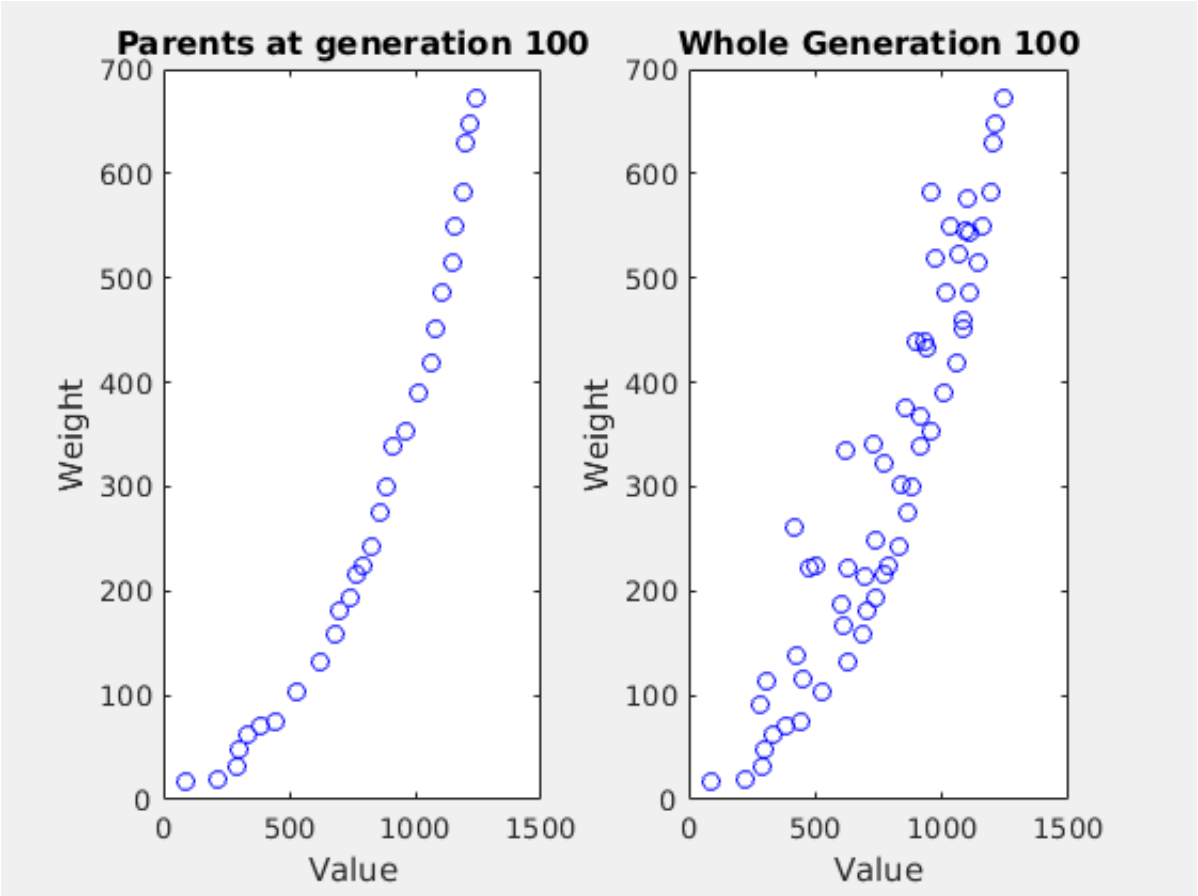
```
        %   every individual.
        x = p;
        x.population = p.population(paretoFronts{i},:);

        distances = computeCrowdingDistance(x);

        %   Sort individuals according to descending distance...
        [sortedDistances, iSortedDistances] = sort(distances,'descend');

        %   ... and carry over the individuals with the highest values.
        sortedPareto = paretoFronts{i}(iSortedDistances);
        nIndividualsLeft = p.nPopulation - size(nextPopulation,1);
        bestSpacedIndividuals = p.population(...
            sortedPareto(1:nIndividualsLeft),:);
        nextPopulation = [nextPopulation;bestSpacedIndividuals];
    end

    % Visualization
    subplot(1,2,2);
    plot(p.value, 803 - p.weight,'bo');
    title(sprintf('Whole Generation %d', generation));
    xlabel('Value');
    ylabel('Weight');
%     axis([0,p.nGenes,0,p.nGenes]);
    pause(0.1);

    p.population = nextPopulation;

end
```




Parents after 50.00 percent
Weight
Value
0
100
200
300
400
500
600
700
200
400
600
800
1000
1200
1400


Parents after 75.00 percent
Weight
Value
0
100
200
300
400
500
600
700
200
400
600
800
1000
1200
1400






```
population = p.population;
```

Results

Because the first pareto front is the first that is being added to the next generation, the first ten individuals of the population are also one of the ten best solutions.

```
bestWeights = unique(p.weight(1:20), 'stable');
bestValues = unique(p.value(1:20),'stable');
result = [803 - bestWeights(1:10)',bestValues(1:10)']
```

```
result = 10x2 double

         673        1242
          18          90
          20         220
         389        1010
          74         445
         103         530
         354         960
         582        1190
         419        1060
         514        1145
```