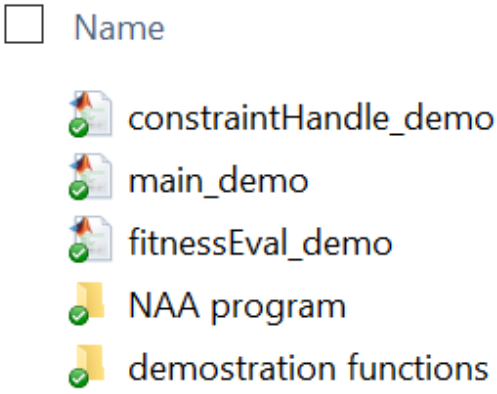
Tutorial of NAA Solver

Fengji Luo, Ph.D Aug/2016

1. Introduction

NAA algorithm is a population-based heuristic optimization method, aims to find a set of optimal decision variables which minimize a given fitness function. NAA mimics the self-aggregation intelligence of group-living animals, and use heuristic strategies to perform stochastic search in the problem space.

Firstly, unzip the NAA solver, the root directory of NAA solver should look like this:



a. [demonstration functions]: include some pre-defined benchmark functions to test NAA;

b. [NAA program]: include the sourced codes of NAA algorithm;

c. [constaintHandle_demo.m]: An example of user-specified constraint handling function;

d. [fitnessEval_demo.m]: An example of user-specified fitness calculation function;

e. [main_demo.m]: An example of main file of runing NAA.

In the evolution process, NAA will generate a population of individuals, where each individual is coded as a *D*-dimension vector, representing a potential solution for the given problem (where *D* is the number of dimensions of the given problem). NAA then repeatedly evaluates their fitness values to driven the evolution. Practical problems may have 1 or multiple constraints, thus before evaluating the fitness values, NAA needs to firstly handle the constraint to make the solutions feasible. Since different practical problems often have different forms of constraints and fitness evaluation logics, the user should implement his/her own constraint handling and fitness evaluation logics in separate functions first, and then NAA will automatically invoke these functions in its evolution process (in software engineering domain, this structure is called “call-back”).

To sum up, to use the NAA solver on your own problem, you need to specify 3 things: (b) if your problem consists of 1 or more constraints, how to handle them; (b) how to calculate the fitness value of your problem; (c) invoke NAA solver to solve your problem. In other words, you need to implement 3 modules on top of the NAA solver:

a. The fitness function;

b. The constraint handling function;

c. The main execution file.

In the following parts of this tutorial, we will use a simple constrained, mixed-integer optimization problem as an example to learn how to use the NAA solver. The test problem is assumed to have 10 variables, where the first 5 variables are continuous variables in the range of [0, 100]; the 6th, 7th, and 8th variables are integers in the rage of [-10, 10]; and the 9th and 10th variables are binary variables. The objective of the problem is to simply minimize the sum of the squares of the variables while satisfying a constraint. The problem can be mathematically formulated as follows:

$$\min \sum_{d=1}^{10} x_d^2$$

Subjected to:

$$x_1 + x_3 = 80$$

$$0 \le x_d \le 100 \quad d = 1,2,3,4,5$$

$$x_d \in \{0,1,2,3,4,5,6,7,8,9,10\} \quad d = 6,7,8$$

$$x_d \in \{0,1\} \quad d = 9,10$$

2. Specify Your Constraint Handling Function

Firstly, you need to implement your own constraint handling logics in a separate function, which should follow following prototype:

function [newInd, newUserObj] = constraintHandle(ind, userObj)

The input parameters:

a. *ind*: The target individual, which will be passed by NAA automatically;

b. *userObj*: A customized structure which stores some other necessary data that will be used in the constraint handling and/or fitness evaluation process. This structure is defined in the main file, as

introduced later. For example, for a power system economic dispatch problem, each dimension represents the power output of a generator, and the sum of the generator outputs must be constrained to be equal to the system load. Therefore, the information of system load needs to be encapsulated into the *userObj* structure and passed into the constraint handling function:

userObj.systemLoad = 200;

The constraint handling function should return two outputs to NAA:

a. *newInd*: The new individual after the constraint adjustment;

b. *newUserobj*: The new userObj structure after the constraint handling process. In most cases, the constraint handling process would alter the original *userObj* structure. In this case, you can just return *newUserObj=userObj*.

There are basically two ways to handle the constraint. The simplest way is that if the individual is checked to violate the constraints, then simply tell NAA to ignore it. This is achieved by setting the output value *newInd* to be '-1':

newInd = -1;

For the test problem, if we use the "check and abandon" strategy, the *constraintHandle_demo.m* file should be written as follows:

```
function [newInd, newUserObj] = constraintHandle_demo(ind, userObj)

limit = userObj.threshold;

if (ind(1)+ind(3)~=limit)
    newInd = -1;
else
    newInd = ind;
end

newUserObj = userObj;

end
```

The second kind of strategy is that if the individual is checked to violate the constraints, then manually adjust it to satisfy the constraints. For example, for the test problem, we need to constraint that the sum of the 1st and 3rd dimensional values of the individual is equal to 120. Therefore, we can use following adjustment strategy:

```
function [newInd, newUserObj] = constraintHandle_demo(ind, userObj)

limit = userObj.threshold;

if (ind(1)+ind(3)~=limit)
    if(ind(1)>limit)
        ind(1)=limit;
        ind(3)=0;
    else
        ind(3) = limit-ind(1);
    end

end

newInd = ind;
newUserObj = userObj;

end
```

The above codes of the second kind of strategy are used in the *constraintHandle_demo.m* file of the NAA solver package.

2. Specify Your Fitness Evaluation Function

After handling the constraints, NAA will invoke the user-specified fitness evaluation function to evaluate the fitness values of the individuals. The fitness evaluation which should follow following prototype:

function [fitnessValue] = constraintHandle(ind, userObj)

Input parameter:

a. *ind*: The target individual, which will be passed by NAA automatically;

b. *userObj*: A customized structure which stores some other necessary data that will be used in the constraint handling and/or fitness evaluation process.

Output:

a. *fitnessValue*: The fitness value of an individual;

For the test problem, the *fitnessEval_demo.m* file is just like follows. In this simple example, we do not need any extra information to calculate the fitness, and thus do not use the *userObj* parameter.

```
function fitness = fitnessEval_demo(ind, userObj)

dimension = length(ind);

fitness = 0;
for i=1:dimension
    v = ind(i);
    fitness = fitness + v*v;
end

end
```

3. Run NAA

Now it is time to implement the main file and run NAA to solve the problem. We follow the steps in the *main_demo.m* file to explain the procedures.

(1) Specify the population size and maximum generation time:

```
popSize = 40;
generation = 500;
```

(2) Specify the value bounds of each dimension. This is a matrix with size of $2 \times D$. The entry [1, *d*] and [2, *d*] indicate the lower and upper limits of the *d*th dimension, respectively. For the test problem, the value bounds are defined as below:

```
D = 10;
bounds = [0, 0, 0, 0, 0, -10, -10, -10, 0, 0;
          100, 100, 100, 100, 100, 10, 10, 10, 1, 1];
```

(3) Specify the data types of each dimension. It should be a vector with the size of $1 \times D$, and the *d*th element represents the data type of the *d*th dimension. In NAA, 3 flags are used for indicating different data types: 0-continuous variable; 1-binary variable; 2-integer variable. For the test problem, the data types are specified as:

```
types = [0,0,0,0,0,2,2,2,1,1];
```

(4) Specify the control parameter settings of NAA:

```
controlParam.shelterNum =2;
avg = popSize/(controlParam.shelterNum);
controlParam.shelterCap = avg;
controlParam.scale_local = 1;
controlParam.Cr_local =0.9;
controlParam.Cr_global = 0.1;
controlParam.alpha = 1;
```

(5) Specify the file names of the user-implemented constraint handling function and fitness evaluation function:

```
fitnessFuncName = 'fitnessEval_demo';
adjustIndFuncName = 'contraintHandle_demo';
```

(6) Specify the *userObj* structure. For the test problem, we only need 1 piece of extra information, which will be used in the constraint handling:

```
userObj.threshold = 80;
```

(7) Specify whether you want to display the iteration information during the evolution on the screen: 0-no display; 1-display:

```
verbose = 0;
```

(8) Invoke NAA to do optimization. Pass the parameters to NAA, and retrieve the results:

```
[bestFitness, bestInd, historicalFitness] = NAA(D, bounds, types, popSize,...
                                                generation, adjustIndFuncName, fitnessFuncName,...
                                                userObj, controlParam, verbose);
```

After the optimization, NAA returns 3 results:

a. *bestFitness*: The final optimal fitness value;

b. *bestInd*: The final optimal solution which yields the *bestFitness*;

c. *historicalFitness*: A column vector with the size of $1\times G$, where G is the maximum generation time. The element [1, g] is the searches best fitness value at the gth generation.

Figure 1 shows the convergence curve by running *main_demo.m* file, and the optimal solution is [40, 0, 40, 0, 0, 0, 0, 0, 0, 0], with the fitness value of 3200.

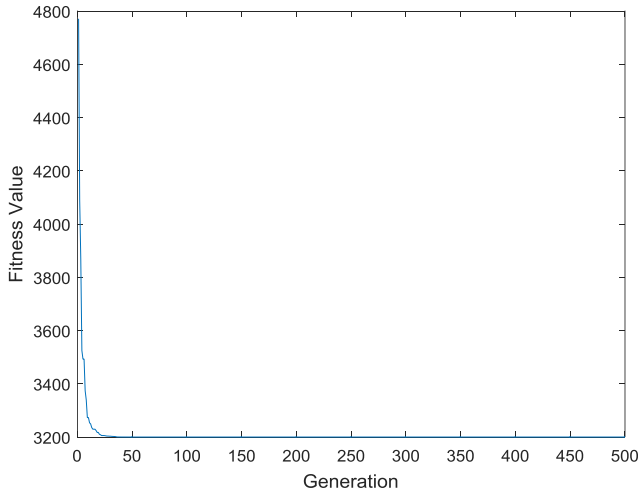


Figure 1 Convergence profile of the test problem

References

Fengji Luo, Junhua Zhao, and Zhao Yang Dong, “A new metaheuristic algorithm for real-parameter optimization: Natural Aggregation Algorithm,” in *Proc. IEEE Congress on Evolutionary Computation*, Vancouver, Canada, 2016.