Mikołów, January 26th, 2018

Semester: 3rd

Group: 2nd

Section: 1st

COMPUTER PROGRAMMING LABORATORY

Author: Tomasz Wieczorek

E-mail: tomawie972@poczta.student.polsl.pl, tomasz.wieczorek@reta.pl

Tutor: Pablo Ribalta

Table of Contents

1. Task topic....................3
2. Project analysis....................3
3. External specification....................4
4. Internal specification....................6
5. Source code....................8
6. Testing....................8
7. Conclusions....................11

1. *Task topic*

Create the application that will find maximum of the given function, using heurestic algorithm, i.e. by crossing best parts of population and by mutations occuring in new populations.

First population should be created at random. Function should be given in form:

$$f(x)=a_1 \cdot x_1^{n_1}+a_2 \cdot x_2^{n_2}+\ldots+a_k \qquad , x_i \in \langle 0, x_{i,max} \rangle, x_{i,max} - user -$$

2. *Project analysis*

Project like that are used into optimization of various functions, like in Artificial Intelligence to perform function optimization generating as less as possible costs. It assumes finding value of various functions finding their values at random x-values. They are also using mutations to avoid potential local extremals.

To create this application I was looking for best way of solving this problem. I have decided to create list of populations, in which we can find another list of its members, so the problem is two-dimensional.

To proper preparation of needed structures I needed to take into account how this algorithm should work:

1. The smallest "particle" of the problem should be one individual (later also called as "member")
2. Bigger part should be whole population(also called as generation), where we can find all members of that one.
3. The biggest "particle" should be every population.

This lead me to question: How to make it in optimal way, without thoughtless copying the same structures without using C++ capabilities? I noticed that:

1. Individuals should be nodes of population.
2. Population should be a list of individuals, **but also** a node of whole problem.
3. Every population should be a list of generations.

It showed me that in this problem I need to use 2 lists, although they should slightly differ, because nodes are different types and various functions should be used. So I decided to create lists using C++ *std::list* that uses *templates* for setting type of nodes and simultaneously *inheritance*. Knowing this, I could to start creating a solution for the problem.

3. *External specification*

Genetics allows to find maximum of a given function using heuristics algorithm. Functions it can calculate are in the form:

$$f(x)=a_1 \cdot x_1^{n_1}+a_2 \cdot x_2^{n_2}+\ldots+a_k \qquad , x_i \in \langle 0, x_{i,max} \rangle , x_{i,max} - user -$$

Preparation:

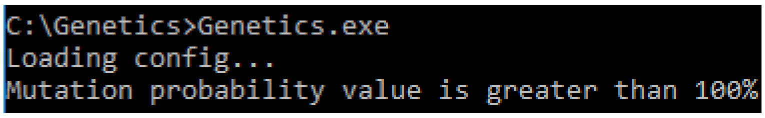
To run appllication, you need to create folder *Genetics* on the disk C:\ . There you have to create two files, e.g. using Notepad:

- params.csv, consisting of following values:
  a_1,x_{1max},n_1
  a_2,x_{2max},n_2
  ...
  a_k,x_{kmax},n_k

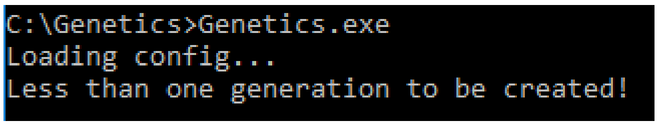
- config.ini, that consists of:
  generation maximum size
  number of generations that should be created
  chance for mutation (without percents, can't be greater than 100).

You can find examples attached to the program.

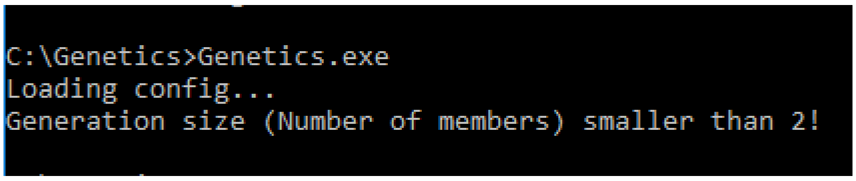
If there's errors while reading files, you can observe following informations, such as:



```
C:\Genetics>Genetics.exe
Loading config...
Mutation probability value is greater than 100%
```



```
C:\Genetics>Genetics.exe
Loading config...
Less than one generation to be created!
```

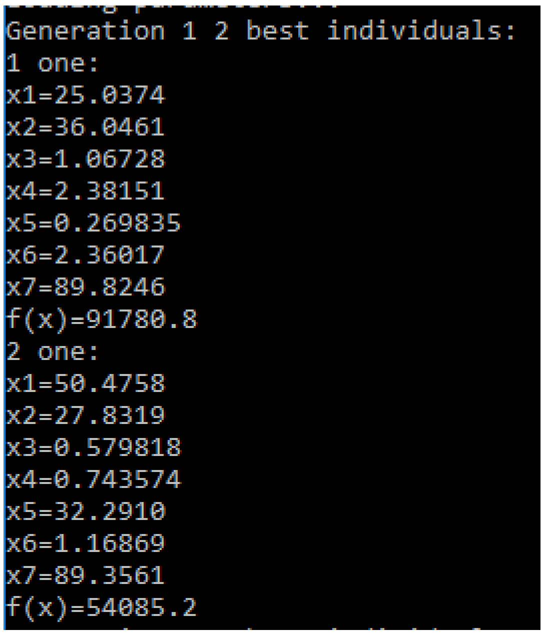


```
C:\Genetics>Genetics.exe
Loading config...
Generation size (Number of members) smaller than 2!
```

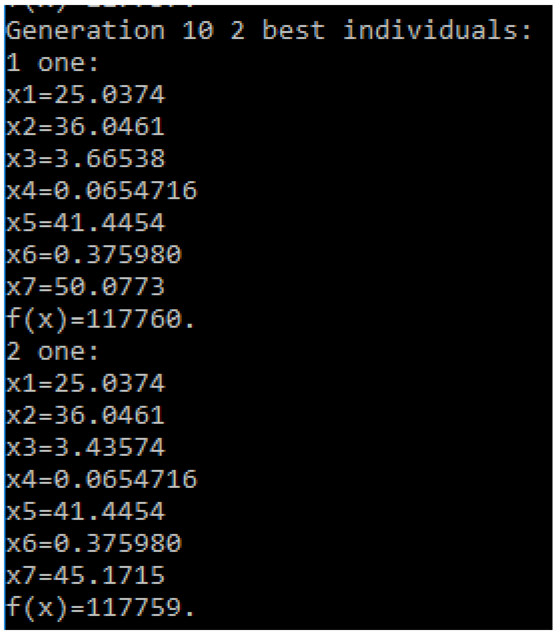
After that information application will exit.

Calculations

After switching on the program, when files are created it will draw k x_i values, that are in the range of $<0, x_{imax}>$. Then it will calculate its first generation values and choose two best of them.


```
Generation 1 2 best individuals:
1 one:
x1=25.0374
x2=36.0461
x3=1.06728
x4=2.38151
x5=0.269835
x6=2.36017
x7=89.8246
f(x)=91780.8
2 one:
x1=50.4758
x2=27.8319
x3=0.579818
x4=0.743574
x5=32.2910
x6=1.16869
x7=89.3561
f(x)=54085.2
```

Then, after creating first generation and choosing best one of them, Genetics will cross these individuals to get new ones. There's also a chance of mutation, set by user. This will be done until number of generations will get to set in *config.ini*.


```
Generation 10 2 best individuals:
1 one:
x1=25.0374
x2=36.0461
x3=3.66538
x4=0.0654716
x5=41.4454
x6=0.375980
x7=50.0773
f(x)=117760.
2 one:
x1=25.0374
x2=36.0461
x3=3.43574
x4=0.0654716
x5=41.4454
x6=0.375980
x7=45.1715
f(x)=117759.
```

In these example we can see that maximum found value has increased more or less two times. Application will be more precisely when number of individuals in generation and number of generations are greater.

4. Internal specification

Files functions:

```
bool loadConfig(std::string, int *, int *, int *); //loads config.ini file

bool loadParams(std::string, std::vector<double>[3]); //loads params.csv file
```

Generation member (individual) class:

```
class GenerationMember {
protected:

       std::vector <double> x_values;                 //x values of member

       long double function_value;                    //function (individual) value

public:
       GenerationMember();                            //default constructor (empty)
       GenerationMember(bool, std::vector<double> *, std::mt19937 *);
                                                      //constructor for first
                                                      generation
       GenerationMember(std::vector<double> *,
std::list<GenerationMember>::iterator[2], std::mt19937 *, int);
                                                      //constructor for second and
                                                      later members

       ~GenerationMember();                           //default destructor (empty)
       double randVal(double, std::mt19937 *);        //function that draw x-values
for member
       double mutate(double, std::mt19937 *, int);    //function that draw if
mutation should occur
       bool operator>(const GenerationMember &);      //compare operator, not used at
this moment
       bool operator<(const GenerationMember &);      //compare operator

       std::vector <double> *get_x_values();          //function that gets values of
x
       long double get_f_value();                     //function that gets function
                                                      value

};
```

One generation members list:

```
class GenerationMembersList
{
protected:
       std::list<GenerationMember> GenMembList;        //list of members
public:
       GenerationMembersList();                        //default constructor (empty)
       GenerationMembersList(bool, std::vector<double> *, const int &, std::mt19937
*);
                                                       //constructor for first
                                                       generation
       GenerationMembersList(std::vector<double> *, const int &,
std::list<GenerationMember>::iterator [2], std::mt19937 *, int);
                                                       //constructor for second and
                                                       later generations
       ~GenerationMembersList();                       //default destructor, clears
                                                       list

       std::list<GenerationMember> *getGenMembList();
                                                       //function that gets list of
                                                       individuals
};
```

List of generations:

```
std::list <GenerationMembersList> GenerationsList;
```

Other important things:

```
       int genSize=-1;                          //generations size
       int gensNumber=-1;                       //generations number
       int mutationChance=-1;                   //mutation chance
                                                //default as -1 as to be loaded from
                                                file(value used for exceptions)
       std::string cfg_filename = "C://Genetics/config.ini";
       std::string par_filename = "C://Genetics/params.csv";
                                                //locations of config and params files
       //RANDOM ENGINES
       std::mt19937 eng(static_cast<unsigned long>(time(0))); //used for real x-values

       std::srand(time(NULL));                                //used for integers
       //END OF ENGINES


       GenerationsList.push_back(GenerationMembersList(1, params, genSize, &eng));
                                                       //first generation

       std::list<GenerationMembersList>::iterator prev;//iterator to previous
                                                       generation (before creating new
                                                       one)

       std::list<GenerationMember>::iterator best[2];  //iterator for 2 best
                                                       individuals
       std::vector <double> *mem_ptr;
                                                       //pointer to obtain values of x
                                                       of 2 best individuals
```

5. Source code

Source code is attached to project in archive “Source code”.

6. Testing

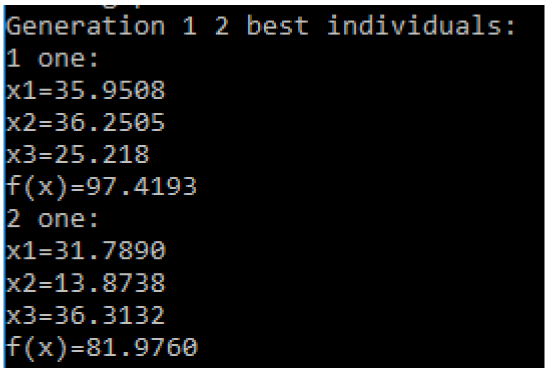
Project was tested for wrong config.ini data, as can be observed in external specification (occurence of exceptions).

For exemplary testing data:

a_i	$x_{i,max}$	ni
1	40	1
1	40	1
1	40	1

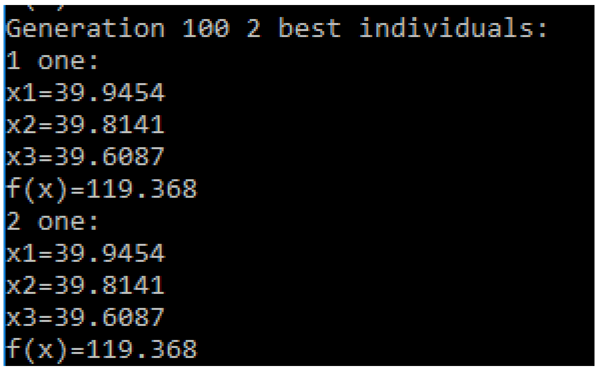
We can simply evaluate that it’s linear function and we know that it’s maximum should be for every x_i=$x_{i,max}$. Let’s check initial values generated by program and check it’s final solution for following config.ini data: 20,200,10.

Initial best values:



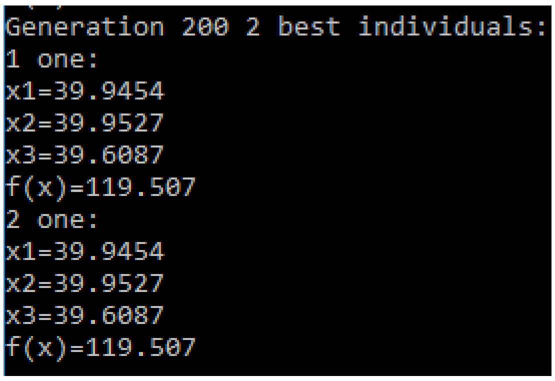
```
Generation 1 2 best individuals:
1 one:
x1=35.9508
x2=36.2505
x3=25.218
f(x)=97.4193
2 one:
x1=31.7890
x2=13.8738
x3=36.3132
f(x)=81.9760
```

Half-way best individuals:



```
Generation 100 2 best individuals:
1 one:
x1=39.9454
x2=39.8141
x3=39.6087
f(x)=119.368
2 one:
x1=39.9454
x2=39.8141
x3=39.6087
f(x)=119.368
```

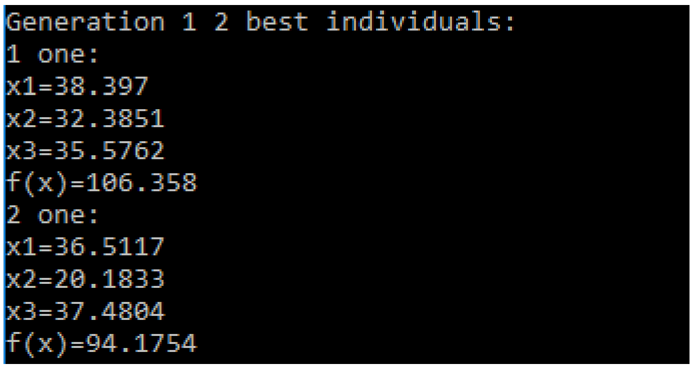
Values of x are equally the same, an f-values are near expected, but let’s check what after next 100 steps:



```
Generation 200 2 best individuals:
1 one:
x1=39.9454
x2=39.9527
x3=39.6087
f(x)=119.507
2 one:
x1=39.9454
x2=39.9527
x3=39.6087
f(x)=119.507
```

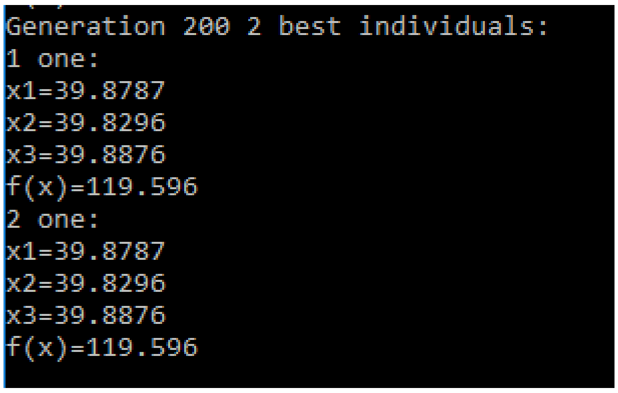
Differences are now not so great, but visible thanks to mutation algorithm.

Let’s run again the same algorithm. Now our initial values are:



```
Generation 1 2 best individuals:
1 one:
x1=38.397
x2=32.3851
x3=35.5762
f(x)=106.358
2 one:
x1=36.5117
x2=20.1833
x3=37.4804
f(x)=94.1754
```

And final value:



```
Generation 200 2 best individuals:
1 one:
x1=39.8787
x2=39.8296
x3=39.8876
f(x)=119.596
2 one:
x1=39.8787
x2=39.8296
x3=39.8876
f(x)=119.596
```

Now we can observe that final value is slightly closer to the expected, but still similar to the previous one.

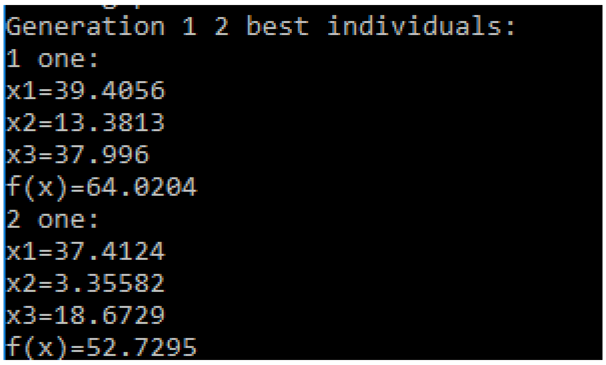
Ratio of these two final values to the expected is close to 99.6-99.7 %

Next testing data:

a_i	$x_{i,max}$	ni
1	40	1
-1	40	1
1	40	1

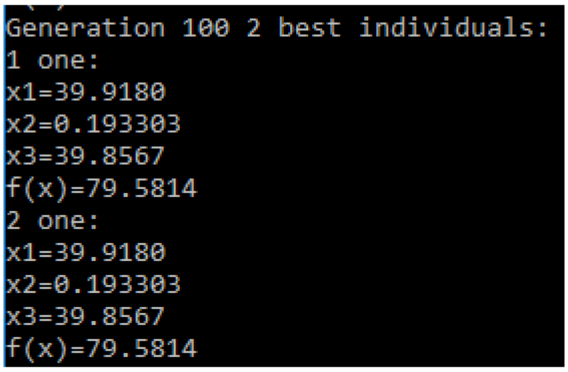
Now we expect values: 40,0,40, let’s check the program.

Initial:



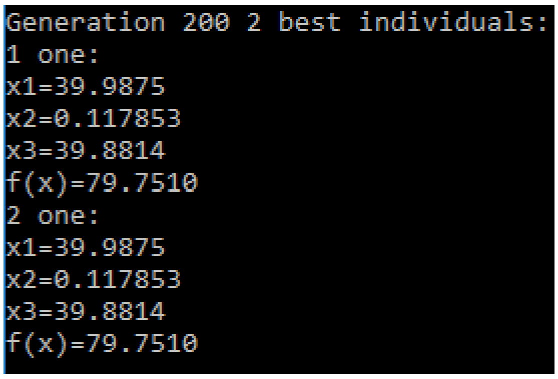
```
Generation 1 2 best individuals:
1 one:
x1=39.4056
x2=13.3813
x3=37.996
f(x)=64.0204
2 one:
x1=37.4124
x2=3.35582
x3=18.6729
f(x)=52.7295
```

Half-way:



```
Generation 100 2 best individuals:
1 one:
x1=39.9180
x2=0.193303
x3=39.8567
f(x)=79.5814
2 one:
x1=39.9180
x2=0.193303
x3=39.8567
f(x)=79.5814
```

Final:



```
Generation 200 2 best individuals:
1 one:
x1=39.9875
x2=0.117853
x3=39.8814
f(x)=79.7510
2 one:
x1=39.9875
x2=0.117853
x3=39.8814
f(x)=79.7510
```

Again, final ratio is about 99.7%.

7. Conclusions

My program works, although there are many improvements that can be implemented, such us:

1. Searching for the minimum or maximum (defined by user).
2. Searching for more values of one type x (e.g. $a_1 \cdot x_1 + a_2 \cdot x_1^2 + a_3 \cdot x_1^3$)
3. Searching for different lowest values of x.
4. Preparing more protection against potential errors.

5. Searching for the maximum also for other functions, like trigonometric functions, e.g. sine or cosine.

6. It probably can be also more optimized.

Implementation of first and second ideas shouldn't be hard, but t he other ones might probably need a bit more work to make them work.