MaCuMBA

Marine Microorganisms: Cultivation Methods for Improving their Biotechnological Applications

Project number: 311957
Start of the project (duration): August 1st, 2012 (48 months)

Collaborative Project
Seventh Framework Programme
Cooperation, KBBE

Deliverable D3.11

Genetic algorithms to improve the culture efficiency of bacterial strains

Organisation name of lead contractor: UvA (2)

Due date of deliverable: M48
Actual submission date: M46

Revision: V.1

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)	
Dissemination Level	
PU Public	**X**
PP Restricted to other programme participants (including the Commission Services)	
RE Restricted to a group specified by the consortium (including the Commission Services)	
CO Confidential, only for members of the consortium (including the Commission Services)	

List of reviewers

Issue	Date	Implemented by
v.1	01/06/2016	Gerard Muyzer

Indicate any document related to this deliverable (report, website, ppt etc) and give file name
** Please attach deliverable documents and any additional material if needed.*

Summary

A bacterial culture medium consists of various components. The concentrations of these components for optimal growth of a bacterium are often unknown and have to be tested experimentally (Garcia-Camacho *et al.*, 2011). Especially for industry and in scientific research, money and time for lab experiments are restricted parameters. Time is money and a minimum of lab experiments to test a specific objective, is highly desired (Patil *et al.*, 2002). The number of possible experimental points N in an n-dimensional variable space is given by N=Ln, where L represents the possible concentrations of each medium component (Weuster-Botz, 2000). This means that for a culture medium with 10 components, each in 5 different concentrations, the number of possible lab-experiments would exceed 9 million. Genetic algorithms can lower the number of necessary lab-experiments to a minimum (Muffler *et al.*, 2007). A genetic algorithm is a computational optimization method for a specific function (e.g. biomass) and is based on the natural selection principles after Darwin. The algorithm is run in iterative steps of predefined functions, but combined with genetic operators, such as selection, crossover and mutation, in order to receive individuals with highest fitness and high genetic variation (Fig. 1; Roubos *et al.*, 1999).

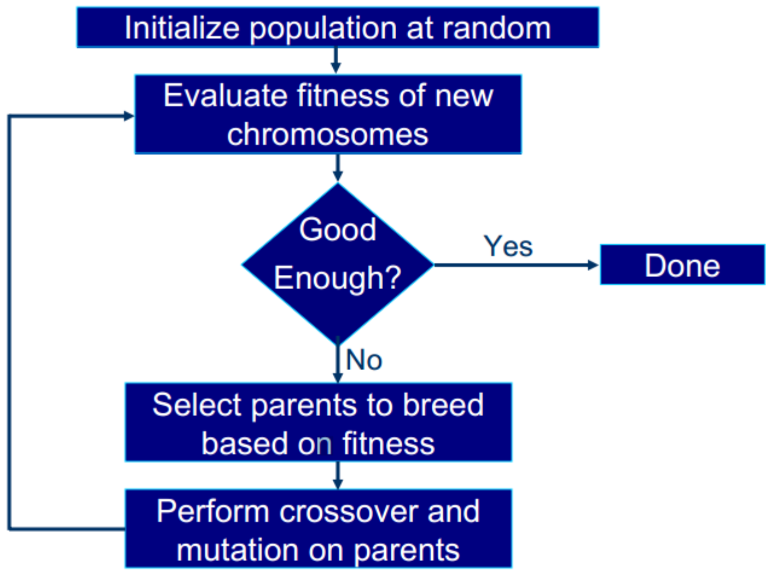
Initialize population at random
Evaluate fitness of new chromosomes
Good Enough?
Yes
Done
No
Select parents to breed based on fitness
Perform crossover and mutation on parents

Figure 1: Flowchart of the genetic algorithm in 5 basic steps (Goodman, 2008).

Here we have developed and tested a genetic algorithm (see appendix) to optimize the cultivation of the cyanobacterium *Synechocystis* PCC6803 in BG11 culture medium (see Table 1 for reference concentrations and tested concentration ranges of medium components). Figure 2 shows the results after 5 generations (blue line). All generations produced culture media that resulted in a biomass increase of at least 12%. The most optimized culture medium gave a biomass increase of 32% in comparison to the amount of biomass obtained with the standard BG11 culture medium. The best-tested medium consisted of components all with higher concentrations than in the standard medium (see Table 1).

Table 1: BG11 medium and concentration ranges used in the genetic algorithms (Franco-Laura *et al.*, 2006)

Medium component	Reference concentration (g/L)	Tested concentration range (g/L)	Concentrations in the best-tested medium (g/L)
$NaNO_3$	1.5	0.15 - 15	7.46
K_2HPO_4	0.031	0.0031 - 0.31	0.21
$MgSO_4 \cdot 7H_2O$	0.075	0.0075 - 0.75	0.5
$CaCl_2 \cdot 2H_2O$	0.037	0.0037 - 0.37	0.28
Citric acid	0.006	0.0006 - 0.06	0.026
Ferric ammonium citrate (18% Fe)	0.006	0.0006 - 0.06	0.01
Na_2CO_3	0.04	0.004 - 0.4	0.058
NaEDTA	0.006	0.0006 - 0.06	

The concentrations of trace metals were kept constant.

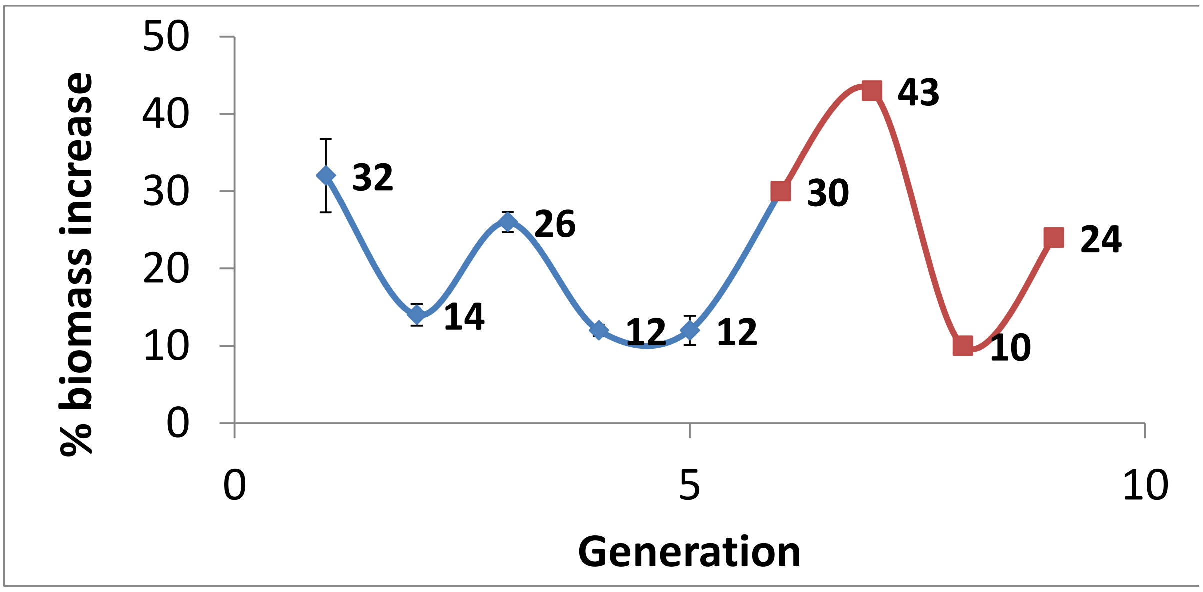

Figure 2. Optimization of culture medium for the growth of the cyanobacterium *Synechocystis* PCC6803 after using a genetic algorithm. The blue line represents the obtained results; the red line is an extrapolation to future generations.

References

Franco-Lara E., Link H., Weuster-Botz D. (2006) Evaluation of artificial neural networks for modelling and optimization of medium composition with a genetic algorithm, Process Biochemistry 41: 2200-2206.

Garcia-Camacho F., Gallardo-Rodriguez J.J., Sanchez-Miron A., Chisti Y., Molina-Grima E. (2011) Genetic algorithm-based medium optimization for a toxic dinoflagellate, Elsevier, Harmful Algae 10: 697-701.

Goodman E.D. (2008) Introduction to genetic algorithms. Proceedings of the 2008 GECCO conference companion on genetic and evolutionary computation, 2277.

Muffler K., Retzlaff M., Van Pee K.-H., Ulber R. (2007) Optimisation of halogenase enyzme activity by application of a genetic algorithm, Journal of Biotechnology 127: 425-433.

Patil S.V., Jayaraman V.K., Kulkarni B.D. (2002) Optimization of media by evolutionary algorithms for production of polyols, Applied biochemistry and biotechnology, Volume 102-103.

Roubos J.A., Van Straten G., Van Boxtel A.J.B. (1999) An evolutionary strategy for fed-batch bioreactor optimization; concepts and performance, Journal of Biotechnology 67: 173-187.

Weuster-Botz: Experimental Design for fermentation media development: Statistical design or global random search? Journal of Bioscience and Bioengineering, Volume 90, Number 5, 473-483, 2000.

Appendix: Genetic algorithm model (Python 2.7.5) consisting of (1) random selection, (2) crossover, (3) mutation, and (4) remainder sampling

To run the genetic algorithm

To get random parent individuals, run Selection 1-73

Select intermediate population, run Selection 73-end

Perform crossover on selected individuals, run Crossover, POP2_c is population after crossover

Perform mutation on individuals, run mutation code 10 times for 10 individuals, output will be position and new value

Codes are dependent on each other: First run Selection, then Crossover, than Mutation scripts.

1. Selection.py

```
#RANDOM SELECTION:

#random selection of parent individuals out of pre-defined concentration ranges for optimization of
BG11 medium
#8 modified macronutrients x1-x8 and 6 constant trace elements x9-x14

#x1=random.uniform(0.15,15)  #NaNO3
#x2=random.uniform(0.0031,0.31)  #K2HPO4
#x3=random.uniform(0.0075,0.75) #MgSO4*7H2O
#x4=random.uniform(0.0037,0.37)  #CaCl2*2H2O
#x5=random.uniform(0.0006,0.06) #Citric acid
#x6=random.uniform(0.0006,0.06)  #Ferric ammonium citrate (18% Fe)
#x7=random.uniform(0.004,0.04)  #Na2CO3
#x8=random.uniform(0.0006,0.06)  #Na2EDTA (for iron binding)

x9=2.86 #H3BO3 (constant)
x10=1.81  #MnCl2*4H2O(constant)
x11=0.222  #ZnSO4*7H2O (constant)
x12= 0.39  #Na2MoO4*2H2O (constant)
x13= 0.079 #CuSO4*5H2O (constant)
x14=0.0494 #Co(NO3)2*6H2O (constant)

# random selection of parent individuals, gives a list called "Medium" with concentration values for
each individual

import random

Medium=[]
for i in xrange(10):
    Medium.append(random.uniform(0.15,15))
    Medium.append(random.uniform(0.0031,0.31))
    Medium.append(random.uniform(0.0075,0.75))
    Medium.append(random.uniform(0.0037,0.37))
    Medium.append(random.uniform(0.0006,0.06))
    Medium.append(random.uniform(0.0006,0.06))
    Medium.append(random.uniform(0.004,0.04))
    Medium.append(random.uniform(0.0006,0.06))
    Medium.append(x9)
    Medium.append(x10)
    Medium.append(x11)
    Medium.append(x12)
    Medium.append(x13)
    Medium.append(x14)
```

```
#splitting up Medium list into values corresponding to different individuals
#definition of individuals

ind0= Medium[0:14]
ind1= Medium[14:28]
ind2= Medium[28:42]
ind3= Medium[42:56]
ind4= Medium[56:70]
ind5= Medium[70:84]
ind6= Medium[84:98]
ind7= Medium[98:112]
ind8= Medium[112:126]
ind9= Medium[126:140]

#definition of population, consisting of 10 different individuals
Pop=[ind0,ind1,ind2,ind3,ind4,ind5,ind6,ind7,ind8,ind9]


#SELECTION, tournament simple (tournament size = 2): 2 individuals are randomly chosen and fitter
one wins the tournament with 80% chance.
#fitter individuals are being selected for intermediate population, others die out
#optical density (OD) values for each individual (new after each generation)

#random numbers as example to run GA
OD_750_ind0=1.834
OD_750_ind1=1.2498
OD_750_ind2=1.345
OD_750_ind3=2.97
OD_750_ind4=1.2358
OD_750_ind5=1.4684
OD_750_ind6=0.8080
OD_750_ind7=1.5008
OD_750_ind8=1.004
OD_750_ind9=1.3056


#list of ODs

OD_Pop=[OD_750_ind0,OD_750_ind1,OD_750_ind2,OD_750_ind3,OD_750_ind4,OD_750_ind5,OD_75
0_ind6,OD_750_ind7,OD_750_ind8,OD_750_ind9]


#conversion of ODs into a representative for biomass concentration
#from Franco-Lara et. al, 2000, mathematically optional due to multiplication of all values with the
same empirical factor
def biomass_conc(x):
    return 0.2699*x


#list of representative values for biomass concentrations of individuals
biomass_Pop=[]
for x in OD_Pop:
    print (biomass_conc(x))
    biomass_Pop.append(biomass_conc(x))


#list of selected individuals
selected_inds=[]

#sort list of representative values of biomasses in ascending order
sort=sorted(biomass_Pop)

# "Weak Elitism" strategy: individual with highest biomass is immediately selected into intermediate
population (population before crossover and mutation)
selected_inds.append([i for i,x in enumerate(biomass_Pop) if x == sort[9]])
```

```
# Also: conversion of list in integer number
selected_inds.append(selected_inds[0][0])
selected_inds.remove(selected_inds[0])


#"Selection tournament simple" (tournament size = 2)
def selection_tournament_simple (biomass_Pop):
#choose a pair of individuals for tournament, randomly chosen position out of the population
    fit_index1 = random.randint(0, len(biomass_Pop) - 1)
    fit_index2 = random.randint(0, len(biomass_Pop) - 1)

    fitter = 0
    weaker = 0
    # defining which individual is fitter and which is weaker, in respect of representative values of
biomass population
    if biomass_Pop[fit_index1] < biomass_Pop[fit_index2]:
        fitter = fit_index1
        weaker = fit_index2
    else:
        fitter = fit_index2
        weaker = fit_index1

#fitter individual wins tournament with a chance of 80%
    if random.random() > 0.8:
        return weaker
    return fitter
#do the tournament simple strategy for 9 individuals and append the selected individuals to the list
selected_inds
#now there are 10 individuals labelled with numbers in the list selected_inds (9 from the tournaments,
1 from elitism (at the beginning of list))
for i in xrange(9):
    print(selection_tournament_simple(biomass_Pop))
    selected_inds.append(selection_tournament_simple(biomass_Pop))


#convert list of numbers of individuals back to list with nutrient component concentrations for each
individual
#intPop=intermediate Population (population before crossover and mutation)
intPop=[]
for i in selected_inds:
    y= 'ind{}'.format(int(i))
    str=''
    print((eval(str.join(y))))
    intPop.append((eval(str.join(y))))
```

2.Crossover.py

```
#CROSSOVER, single point: increases genetic variability of the population through exchange of genes
within the individuals

#crossover probability 80%
P_c=0.8

#pc is random number between 0 and 1
def pc():
    return random.random()

#c_point is crossover point, random number between 1 and 6 (at which gene/ nutrient component)
does crossover take place? ends at 6 because after this position, nutrients are constant so crossover
not useful
def c_point():
    return random.choice(np.arange(0,7,1))
```

```
#randomly mix individuals within intermediate Population, break up order to form random crossover
pairs
random.shuffle(intPop)


#POP2_c is a list of new individuals after crossover
POP2_c=[]


#for each crossover-pair: if random number between 0 and 1 is smaller 0.8, crossover takes place at
position c_point
#if random number between 0 and 1 is higher 0.8, individuals of the crossover pair are copied into the
Pop2_c list
for i in range(0, 9, 2) :
    if (pc()<0.8):
        y=c_point()
        POP2_c.append(intPop[i][:y]+intPop[i+1][y:])
        POP2_c.append(intPop[i+1][:y]+intPop[i][y:])
    else:
        POP2_c.append(intPop[i])
        POP2_c.append(intPop[i+1])
```

3. Mutation.py

```
#MUTATION: exchange of a concentration value of a nutrient wiht 15% possibility and at random
position

#mutation probability for each nutrient and individual is 15%
P_m=0.15

#random number between 0 and 1
def pm(x):
    return random.random()

#m_point is point of mutation
def m_point():
     return random.choice(np.arange(0,7,1))


#8 nutrients, where mutation is possible
z0=random.uniform(0.15,15)  #NaNO3
z1=random.uniform(0.0031,0.31)  #K2HPO4
z2=random.uniform(0.0075,0.75) #MgSO4*7H2O
z3=random.uniform(0.0037,0.37)  #CaCl2*2H2O
z4=random.uniform(0.0006,0.06) #Citric acid
z5=random.uniform(0.0006,0.06)  #Ferric ammonium citrate (18% Fe)
z6=random.uniform(0.004,0.04)  #Na2CO3
z7=random.uniform(0.0006,0.06)  #Na2EDTA (for iron binding)



# do mutation with 15% probability and print position and new value
#this is for one individual, do times x for 10 individuals
for i in xrange(8):
    if pm(x) < 0.15:
        print(i)
        y='z{}'.format(i)
        str=''
        print(eval(str.join(y)))
```

4. Remainder_sampling.py

```
#remainder stochastic sampling selection method

#RANDOM SELECTION:

#random selection of parent individuals out of pre-defined concentration ranges
#concentration ranges for optimization of BG11 medium
#8 modified macronutrients x1-x8 and 6 constant trace elements x9-x14

#x1=random.uniform(0.15,15)  #NaNO3
#x2=random.uniform(0.0031,0.31)  #K2HPO4
#x3=random.uniform(0.0075,0.75) #MgSO4*7H2O
#x4=random.uniform(0.0037,0.37)  #CaCl2*2H2O
#x5=random.uniform(0.0006,0.06) #Citric acid
#x6=random.uniform(0.0006,0.06)  #Ferric ammonium citrate (18% Fe)
#x7=random.uniform(0.004,0.04)  #Na2CO3
#x8=random.uniform(0.0006,0.06)  #Na2EDTA (for iron binding)

x9=2.86 #H3BO3 (constant)
x10=1.81  #MnCl2*4H2O(constant)
x11=0.222  #ZnSO4*7H2O (constant)
x12= 0.39  #Na2MoO4*2H2O (constant)
x13= 0.079 #CuSO4*5H2O (constant)
x14=0.0494 #Co(NO3)2*6H2O (constant)

import random

Medium=[]
for i in xrange(10):
    Medium.append(random.uniform(0.15,15))
    Medium.append(random.uniform(0.0031,0.31))
    Medium.append(random.uniform(0.0075,0.75))
    Medium.append(random.uniform(0.0037,0.37))
    Medium.append(random.uniform(0.0006,0.06))
    Medium.append(random.uniform(0.0006,0.06))
    Medium.append(random.uniform(0.004,0.04))
    Medium.append(random.uniform(0.0006,0.06))
    Medium.append(x9)
    Medium.append(x10)
    Medium.append(x11)
    Medium.append(x12)
    Medium.append(x13)
    Medium.append(x14)


#splitting up Medium list into values corresponding to different individuals
#definition of individuals

ind0= Medium[0:14]
ind1= Medium[14:28]
ind2= Medium[28:42]
ind3= Medium[42:56]
ind4= Medium[56:70]
ind5= Medium[70:84]
ind6= Medium[84:98]
ind7= Medium[98:112]
ind8= Medium[112:126]
ind9=Medium[126:140]

#definition of population, consistin of 10 different individuals
Pop=[ind0,ind1,ind2,ind3,ind4,ind5,ind6,ind7,ind8,ind9]


#SELECTION, tournament simple (tournament size = 2): 2 individuals are randomly chosen and fitter
one wins the tournament with 80% chance.
```

```
#fitter individuals are being selected for intermediate population, others die out
#optical density (OD) values for each individual (new after each generation)

OD_750_ind0=0.7962
OD_750_ind1=1.3522
OD_750_ind2=1.3901
OD_750_ind3=1.3592
OD_750_ind4=1.0227
OD_750_ind5=1.3088
OD_750_ind6=1.0526
OD_750_ind7=1.2448
OD_750_ind8=0.7724
OD_750_ind9=1.0624



#list of ODs

OD_Pop=[OD_750_ind0,OD_750_ind1,OD_750_ind2,OD_750_ind3,OD_750_ind4,OD_750_ind5,OD_75
0_ind6,OD_750_ind7,OD_750_ind8,OD_750_ind9]


#conversion of ODs into biomass concentration

def biomass_conc(x):
    return 0.2699*x


#list of biomass concentrations of individuals

biomass_Pop=[]
for x in OD_Pop:
    print (biomass_conc(x))
    biomass_Pop.append(biomass_conc(x))


y=biomass_Pop

#average biomass is the sum of all biomass values divided by the amount of individuals (10)
def average_biomass (x):
    return sum(y)/len(y)

#selection probability is fitness function: individual biomass value divided by average biomass value
def selection_prob(x):
    return (x/average_biomass(x))


#Selection1 is a list with all the selection probabilites of the individuals
Selection1=[]
for x in biomass_Pop:
    print (selection_prob(x))
    Selection1.append(x)

#make a list which shows which individual belongs to which selection probability
keys=['ind1','ind2','ind3','ind4','ind5','ind6','ind7','ind8','ind9','ind10']
values=Selection1
sel_prob_list= (zip(keys,values))


#normalize selection probabilities to 10 gives list3

list3=[]
for x in Selection1:
    list3.append((x*10)/sum(Selection1))
```

```
#if the normalized value is high (>1) then return the integer portion
def highno_sel_copies(x):
    return x-(x%1)
# first "tournament": get all the integer values
def no_copies_gen0_sel1(x):
    if x>=1:
        return highno_sel_copies(x)
    else:
        return 'no copy'
# "second tournament": get all the remaining float values
def remain_copies_gen0_sel1(x):
    if x >=1:
        return (x%1)
    else:
        return (x)
#list with all the integer values after first "tournament"
GEN0_SEL1a=[]
for x in list3:
   GEN0_SEL1a.append(no_copies_gen0_sel1(x))
#list with all the float values after first "tournament"
GEN0_SEL1b=[]
for x in list3:
    GEN0_SEL1b.append(remain_copies_gen0_sel1(x))



#sum of all the integer portions in the integer list
sum_1a= sum([x for x in GEN0_SEL1a if type(x) == float])
#enumerated list of integers
GEN0_SEL1a_en=list(enumerate(GEN0_SEL1a))


#new list which shows the individual and how many copies are selected for the intermediate
population
POP2a =[]

#if the sum of copies in intermediate population is not 10, then take integer portion as representatives
for
#the amount of copies to the POP2a list
import operator
for x in sorted(GEN0_SEL1a_en, key=operator.itemgetter(1), reverse=True):
    if sum_1a==10:
        if x[1] != 'no copy':
            POP2a.append(x)
    elif sum_1a <10:
        if x[1]!= 'no copy':
            POP2a.append(x)
    else:
        while sum(POP2a) <10:
            if x[1]!= 'no copy':
                POP2a.append(x)


#the remainder float portion in GEN0_SEL1b is normalized again to 10 in list4
list4=[]
for x in GEN0_SEL1b:
    list4.append((x*10)/sum(GEN0_SEL1b))

#same procedure as before! 2nd "tournament"

#list of integer portions of copies
GEN0_SEL2a=[]
for x in list4:
    GEN0_SEL2a.append(no_copies_gen0_sel1(x))
#list of remaining float portions of copies
```

```
GEN0_SEL2b=[]
for x in list4:
    GEN0_SEL2b.append(remain_copies_gen0_sel1(x))


#enumerated list of integer portions of copies
GEN0_SEL2a_en=list(enumerate(GEN0_SEL2a))


#x_new is a new value of copies, which is added to POP2a (intermediate Population), if the sum of
POP2a exceeds 10

def x_new(x):
    return (x[0], x[1]-10)

for x in sorted(GEN0_SEL2a_en, key=operator.itemgetter(1), reverse=True):
    if x[1] != 'no copy' and (sum([pair[1] for pair in POP2a]) <10):
        POP2a.append(x)
    else:pass


def x_new(x):
    return (x[-1][0]), (10-(sum([pair[1] for pair in x[:-1]])))


if (sum([pair[1] for pair in POP2a]))>10:
    POP2a.append(x_new(POP2a))
    del(POP2a[-2])
```