```
from __future__ import division
from searcher import *
from models import *
import sys, sk
from decimal import *
import numpy as np
from anzeigen import *
from time import gmtime, strftime
import sys, random, math, datetime, time,re
sys.dont_write_bytecode = True
rdivDemo=sk.rdivDemo

def what2say(k,modelName):
  hi, lo, kooling, indepSize, thresh, iterations = k.eigenschaften()
  if modelName.__doc__≡"Simulated Annealing":
    return {’Max: ’: hi, ’Min: ’: lo, ’Cooling Factor: ’:kooling,
            ’Iterations: ’: iterations}
  elif modelName.__doc__≡"Max Walk−SAT":
    return {’Max: ’: hi, ’Min: ’: lo, ’Retries: ’: 100,
            ’Iterations: ’: 100}
  elif modelName.__doc__≡’Genetic Algorithm’:
    return {’Max: ’: hi, ’Min: ’: lo, ’Population: ’: 50,
            ’Generations: ’: 400, ’crossover: ’: 0.6}
  elif modelName.__doc__≡’Differential Evolution’:
    return {’Max: ’: hi, ’Min: ’: lo, ’Iterations: ’: 100,
            ’NP: ’:100, ’f: ’:0.75, ’cf: ’:0.3}
  elif modelName.__doc__≡’Particle Swarm Optimization’:
    return {’Max: ’: hi, ’Min: ’: lo, ’Iterations: ’: 100,
            ’Number of Particles: ’:30, ’phi1: ’:1.3, ’phi2: ’:2.6}

#===============================================================================
# Baselining
#===============================================================================
emin=emax=0;
for x in [Schaffer, Kursawe,
          Fonseca, ZDT1, ZDT3, Viennet3, DTLZ7]:
  for y in [PSO, GA, diffEvolve, SimulatedAnnealer, MaxWalkSat]:
    k=modelBasics(x)
    eMax, eMin = k.baselining(x)
    emax= eMax if eMax>emax else emax
    emin= eMin if eMin<emin else emin
print ’Baselining...’



for x in [Schaffer, Kursawe,
          Fonseca, ZDT1, ZDT3, Viennet3, DTLZ7]:
  early=True
  E=[]
  for i in xrange(50): sys.stdout.write(’_’)
  print ’\n’
  print ’Model: ’, x.__doc__
  for i in xrange(50): sys.stdout.write(’−’)
  print ’\n’
  print strftime("%a, %d %b %Y %H:%M:%S ", gmtime()), ’GMT’, ’\n’

  for y in [PSO, diffEvolve, GA, SimulatedAnnealer, MaxWalkSat]:
    eb=30*[0]
    print ’Searcher: ’, y.__doc__
    k=x()
    reps=30
    dspl=anzeigen();
    hi, lo, kooling, indepSize, thresh, iterations = k.eigenschaften()
    print ’Settings:’
    toprint=what2say(k,y);
    for k in toprint:
      print k, toprint[k]
    #if early: print ’Early Termination!’  , ’\n’
    for r in xrange(reps):
      a=y(x,disp=False,early=early)
      eb[r] =  a.runSearcher(emax, emin)
    eb.insert(0,y.__doc__)
    E.append(eb)
    #print dspl.xtile(eb[1:])
    """for r in xrange(reps):
  print dspl.xtile(eb[r:r+50], lo=lo, hi=hi)"""
    print ’Energy: ’, "{:.3E}".format(Decimal(str(np.sum(eb[1:])/reps))), ’\n’

  def _rDiv():
    rdivDemo(E)
  _rDiv()

  #

  sys.stdout.write(’\n’)
```

```
# −*− coding: utf−8 −*−
"""
Created on Mon Sep 15 03:04:43 2014

@author: rkrsn
"""
from __future__ import division
import sys
import math, random, numpy as np, scipy as sp
from math import ceil
sys.dont_write_bytecode = False
from models import *
from anzeigen import *
from dynamikliste import *
# from sk import Num
import analyzer
import types

# Define some aliases.
rand = random.uniform
randi = random.randint
exp = math.exp

class SimulatedAnnealer(object):
  "Simulated Annealing"
  def __init__(self, modelName, disp=False, early=False):
    self.modelName = modelName
    self.disp = disp
    self.early = early
  def runSearcher(self,emax,emin):
    modelbasics = modelBasics(self.modelName);
    modelFunction = self.modelName()
    anz = anzeigen();
    hi, lo, kooling, indepSize, thresh, iterations = \
     modelFunction.eigenschaften()
    #emax, emin = modelbasics.baselining(self.modelName)
    sb = s = [randi(lo, hi) for z in xrange(indepSize)];
    eb = e = modelbasics.energy(s, emax, emin)
    enRec = dynamikliste()  # Creates a growing list.
    enRec[0] = 0;
    # Since iterations start from 1, lets initialize enRec[0] to 0
    analyser = analyzer.analyser()
    epochs = 5 if self.early else iterations;
    k = 1;
    while epochs ∧ k < iterations:
      sn = modelbasics.neighbour(s, hi, lo)
      en = modelbasics.energy(sn, emax, emin)
      t = k / iterations
      if en < eb:
        eb, sb, enRec[k] = en, sn, en;
        if self.disp:
          modelbasics.say(’!’)

      if en < e:
        s, e, enRec[k] = sn, en, en;
        if self.disp:
          modelbasics.say(’+’)

      if modelbasics.do_a_randJump(en, e, t, kooling):
        # The cooling factor needs to be really low for some reason!!
        s, e, enRec[k] = sn, en, en;
        if self.disp:
          modelbasics.say(’?’)
      else:
        enRec[k] = en
      if self.disp:
        modelbasics.say(’.’)
      if k % 50 ≡ 0 ∧ k > 50:
        # print enRec[:−10]
        proceed = analyser.isItGettinBetter(enRec[k − 100:])
        if proceed:
          epochs += 1;
        else:
          epochs −= 1;
        # print enRec[k−40:] #
      k = k + 1
      if k % 40 ≡ 0:
        if self.disp:
          modelbasics.say(’\n’)  # sa.say(format(sb,’0.2f’))

    if self.disp:
      modelbasics.say(’\n’),
  # Print Energy and best value.
    for i in xrange(k):
      if self.disp:
        if i % 50 ≡ 0:
```

```
          print anz.xtile(enRec[i − 50:])
    if self.disp:
      modelbasics.say(’\n’)
    return eb

class MaxWalkSat(object):
  "Max Walk−SAT"
  def __init__(self, modelName, disp=False, early=True, maxTries=100,
               maxChanges=100):
    self.modelName = modelName
    self.disp = disp
    self.maxTries = maxTries
    self.maxChanges = maxChanges
  def runSearcher(self, emax, emin):
    modelbasics = modelBasics(self.modelName);
    modelFunction = self.modelName()
    hi, lo, kooling, indepSize, thresh, iterations = \
    modelFunction.eigenschaften()
    #emax, emin = modelbasics.baselining(self.modelName)
    for i in xrange(self.maxTries):
        # Lets create a random assignment, I’ll use list comprehesions here.
        x = xn = xb = [rand(lo, hi) for z in xrange(indepSize)]
        # Create a threshold for energy,
        # let’s say thresh=0.1% of emax (which is 1) for starters
        for j in xrange(self.maxChanges):
            # Let’s check if energy has gone below the threshold.
            # If so, look no further.
            if modelbasics.energy(xn, emax, emin) < thresh:
                xb=xn
            else:
              # Choose a random part of solution x
                randIndx = randi(0, indepSize − 1)
                if rand(0, 1) > 1 / (indepSize + 1):  # Probablity p=0.33
                    y = xn[randIndx]
                    xn[randIndx] = modelbasics.simpleneighbour(y, hi, lo)
                    # print ’Random change on’, randIndx
                else:
                    # xTmp is a temporary variable
                    xBest = emax;
                    # Step from xmin to xmax, take 10 steps
                    Step = np.linspace(lo, hi, 10)
                    for i in xrange(np.size(Step)):
                        xNew = xn; xNew[randIndx] = Step[i];
                        if modelbasics.energy(xNew, emax, emin) < xBest:
                            xBest = modelbasics.energy(xNew, emax, emin)
                            xn = xNew

            if modelbasics.energy(xn, emax, emin) < modelbasics.energy(xb,
                                                                      emax,
                                                                      emin):
              xb = xn
              print modelbasics.energy(xn, emax, emin)
    return modelbasics.energy(xb, emax, emin)

class GA(object):
  "Genetic Algorithm"
  def __init__(self, modelName, disp=False, early=True, popcap=50,
               generations=400, crossover=0.6):
    self.modelName = modelName
    self.disp = disp
    self.popcap = popcap
    self.generations = generations
    self.crossover= crossover
  def runSearcher(self, emax, emin):
    modelbasics = modelBasics(self.modelName);
    modelFunction = self.modelName()
    hi, lo, kooling, indepSize, thresh, iterations = \
    modelFunction.eigenschaften()
    #emax, emin = modelbasics.baselining(self.modelName)
    #−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    def init_pop(indepSize, lo, hi, N=self.popcap):
      return [[rand(lo,hi) for _ in xrange(indepSize)] for _ in xrange(N)]

    #−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    def evalPop(Pop, emax, emin):
      score=[];
      for individual in Pop:
        score.append(modelbasics.energy(individual,emax,emin))
      indices=[i[0] for i in sorted(enumerate(score), key=lambda x:x[1],
                                    reverse=False)]
      scores=[i[1] for i in sorted(enumerate(score), key=lambda x:x[1],
                                   reverse=False)]
      return [Pop[z] for z in indices], scores

    #−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    def evolve(Pop, emax, emin, hi, lo, indepSize, retain=0.2, randSelect=0.05,
               crossover=self.crossover, mutate=1/(indepSize*(hi-lo))):
      parents, score=evalPop(Pop, emax, emin)
      parents=parents[:int(len(score)*retain)]
      # Increase diversity by selecting bad parents
      for indv in parents[int(len(score)*retain):]:
        if rand(0,1)<randSelect:
          parents.append(indv)

      # Crossover parents to create children
      childern=[]
      numChildren=len(Pop)-len(parents)

      while len(childern)<numChildren:
        he=randi(0,len(parents)-1);
        she=randi(0,len(parents)-1);
        #print parents
        if he≠she:
          he=parents[he]; she=parents[she]
          if indepSize≡1:
            flatten = lambda x: x if ¬ isinstance(x, list) else x[0]
            #print he, she
            child=0.5*(flatten(he)+flatten(she)) \
            if mutate<rand(0,1) else rand(lo,hi)
          else:
            #print he, she
            child=he[:int(0.5*indepSize)]+she[int(0.5*indepSize):]
            if mutate>rand(0,1): child[randi(0,indepSize-1)]=rand(lo,hi)
          childern.append(child)

      parents.extend(childern)

      return parents

    #--------------------------------------------------------------------------
    Pop=init_pop(indepSize, lo, hi, self.popcap)
    pn, en= evalPop(Pop, emax, emin)
    eb=en[0]
    pBest=pn[0]
    for i in xrange(self.generations):
      Pop=evolve(Pop, emax, emin, hi, lo, indepSize)
      # Spit out the magic variables please
      pn, en= evalPop(Pop, emax, emin)
      if en[0]<eb:
        eb=en[0]; pBest=pn[0]
    #print pBest
    return eb

class diffEvolve(object):
  "Differential Evolution"
  def __init__(self, modelName, disp=False, early=False,
               maxIter=100, NP=100, f=0.75, cf=0.3):
    self.modelName = modelName
    self.disp=disp
    self.early=early
    self.maxIter=maxIter
    self.NP,self.f,self.cf=NP,f,cf
  def runSearcher(self, emax, emin):
    modelbasics = modelBasics(self.modelName);
    modelFunction = self.modelName()
    hi, lo, __, indepSize, thresh, __ = modelFunction.eigenschaften()
    #emax, emin = modelbasics.baselining(self.modelName)
    #--------------------------------------------------------------------------
    def inititalPopultaion(indepSize, lo, hi, N=self.NP):
      return [[lo+(hi-lo)*rand(0,1) for _ in xrange(indepSize)]
              for _ in xrange(N)]

    #--------------------------------------------------------------------------
    def evalFront(Pop, emax, emin):
      score=[];
      for individual in Pop:
        score.append(modelbasics.energy(individual,emax,emin))
      indices=[i[0] for i in sorted(enumerate(score), key=lambda x:x[1],
                                    reverse=False)]
      scores=[i[1] for i in sorted(enumerate(score), key=lambda x:x[1],
                                   reverse=False)]
      return Pop[indices[0]], scores[0]

    #--------------------------------------------------------------------------
    def spawn(P0, Frontier, hi, lo, NP=self.NP, cf=self.cf, f=self.f):
      """
  Create a new member for the frontier using some new values and by
  extrapolating P0 (the old value)
  """
      first = P0
      second, third, fourth = first, first, first
```

```
      while second≡first:
        second=Frontier[randi(0,len(Frontier)-1)]
      while third≡second ∨ third≡first:
        third=Frontier[randi(0,len(Frontier)-1)]
      while fourth≡second ∨ fourth≡first ∨ fourth≡third:
        fourth=Frontier[randi(0,len(Frontier)-1)]
      trim = lambda x: max(lo, min(x, hi))
      return [first[z] if cf<rand(0,1) else trim(second[z]+f*(third[z]-fourth[z]))
           for z in xrange(len(first))]

    Frontier=inititalPopultaion(indepSize, lo, hi)
    gBest, eBest = evalFront(Frontier, emax, emin)
    maxIter=self.maxIter
    while maxIter ∧ (eBest>thresh):
      newFrontier=[]
      for F_i in Frontier:
        newSamp=spawn(F_i, Frontier, hi, lo)
        if modelbasics.energy(newSamp,emax,emin) < modelbasics.energy(newSamp,
                                                                     emax,emin):
          newFrontier.append(newSamp)
        else:
          newFrontier.append(F_i)
      Frontier=newFrontier
      gBest, eBest = evalFront(Frontier, emax, emin)
      maxIter-=1
    return eBest
    #--------------------------------------------------------------------------
class PSO(object):
  "Particle Swarm Optimization"
  def __init__(self, modelName, disp=False, early=True, numPart=30, phi1=1.3, phi2=2.8):
    self.numPart=numPart
    self.phi1=phi1
    self.phi2=phi2
    self.modelName=modelName
  def runSearcher(self, emax, emin):
    modelbasics = modelBasics(self.modelName);
    modelFunction = self.modelName()
    score = lambda x: modelbasics.energy(x,emax,emin)
    hi, lo, __, indepSize, thresh, maxIter = modelFunction.eigenschaften()
    #emax, emin = modelbasics.baselining(self.modelName)
    def velocity(Pos, Vel, pBest, gBest, hi, phi1=self.phi1, phi2=self.phi2):
      k=2/abs(2-phi1-phi2-math.sqrt(phi1**2+phi2**2)-4*(phi1+phi2))
      Vel= [1*(Vel[r]+phi1*rand(0,1)*(pBest[r]-Pos[r])\
               +phi2*rand(0,1)*(gBest[r]-Pos[r])) for r in xrange(indepSize)]
      return [v if v<hi else 0 for v in Vel]
    #==========================================================================
    # Initialize particle values
    #==========================================================================
    pPos=[] # Position of the particles
    pVel=[] # Velocity of the particles
    pBest=[];
    gBest=[rand(lo,hi) for j in xrange(indepSize)]
    for i in xrange(self.numPart):
      pVel.append([0 for j in xrange(indepSize)])
      pPos.append([rand(lo,hi) for j in xrange(indepSize)])
      pBest.append(pPos[i])
      if score(pBest[i])<score(gBest):
        gBest=pBest[i]
    #==========================================================================
    # Run PSO
    #==========================================================================
    maxIter=1000;
    while maxIter:
      for i in xrange(self.numPart):
        pVel[i] = velocity(pPos[i], pVel[i], pBest[i], gBest, hi)
        pPos[i] = [j+k for j,k in zip(pPos[i], pVel[i])]
        pPos[i] = [hi if p>hi else lo if p<lo else p for p in pPos[i]]
        if score(pPos[i])<score(pBest[i]):
          pBest[i]=pPos[i]
          if score(pBest[i])<score(gBest):
            gbest=pBest[i]
      maxIter-=1
    return score(gBest)

if __name__ ≡ ’main’:
  SimulatedAnnealer(Schaffer)
```

```
"""
A models file that can be imported to run optimizers
"""
from __future__ import division
import sys, types
import math, random, numpy as np, scipy as sp
from math import sin
sys.dont_write_bytecode = False
# Define some aliases.
rand=random.uniform
randi=random.randint
exp=math.e
sin=math.sin
sqrt=math.sqrt
pi=math.pi

class modelBasics(object):
  def __init__(i,model):
    i.model=model()
    i.name=model.__name__
  def do_a_randJump(i, e, en, t, k):
    p=exp**(−(e−en)/(t**k))<rand(0,1)
    return p
  def simpleneighbour(self,x,xmax,xmin):
      return xmin+(xmax−xmin)*rand(0,1)
  def neighbour(i,x,xmax,xmin):
    def __new(x,z):
        return xmin+(xmax−xmin)*rand(0,1) if rand(0,1)<1/(i.model.indepSize) \
          else x[z]
    x_new=[__new(x,z) for z in xrange(i.model.indepSize)]
    return x_new
  def energy(i,x,emax,emin,sigmoid=False):
    if ¬ sigmoid:
      ener=i.model.score(x);
      e_norm= ((ener−emin)/(emax−emin))
    else:
      ener=i.model.score(x)
      e_norm=1/(1+exp**(−ener/1e4))
    return e_norm
  def baselining(i,model):
    emax=0;emin=0;
    indepSize=i.model.indepSize;
    for _ in xrange(int(1e3)):
      x_tmp=[rand(i.model.baselo,i.model.basehi) for _ in xrange(indepSize)]
      ener=i.model.score(x_tmp);
      if ener>emax:
        emax=ener
      elif ener<emin:
        emin=ener
    return emax,emin
  f=open(’log_sa_schaffer.txt’,’w’)
  def say(i,x):
    sys.stdout.write(str(x));
    sys.stdout.flush()

class Schaffer(object):
  "Schaffer"
  def __init__(i,hi=100,lo=−100, basehi=1000, baselo=−1000, kooling=0.7,
               indepSize=1, thresh=1e−2, iterations=2000):
    i.hi, i.lo, i.basehi, i.baselo=  hi, lo, basehi, baselo
    i.thresh=thresh
    i.kooling, i.indepSize, i.iterations= kooling, indepSize, iterations
    random.seed()
  def f1(i,x):
    return x*x
  def f2(i,x):
    return (x−2)**2
  def score(i,x):
    from compiler.ast import flatten
    flatten = lambda x: x if ¬ isinstance(x, list) else x[0]
    return i.f1(flatten(x))+i.f2(flatten(x))
  def eigenschaften(i):
    return i.hi, i.lo, i.kooling, i.indepSize, i.thresh, i.iterations

class Kursawe(object):
  "Kursawe"
  def __init__(i,hi=5,lo=−5,kooling=0.6, a=0.8, b=3, indepSize=3, basehi=1000,
               baselo=−1000, thresh=1e−2, iterations=2000):
    i.hi, i.lo, i.basehi, i.baselo, i.kooling = hi, lo, basehi, baselo, kooling
    i.thresh=thresh
    i.a, i.b, i.indepSize, i.iterations= a, b, indepSize, iterations
    random.seed()
  def f1(i,x):
    return np.sum([−10*exp**(−0.2*sqrt(x[z]**2+x[z+1]**2)) \
                   for z in xrange(i.indepSize−1)])
  def f2(i,x):
```

```
    return np.sum([abs(x[z])**i.a+5*sin(x[z]**i.b) \
                   for z in xrange(i.indepSize)])
  def score(i,x):
    return i.f1(x)+i.f2(x)
  def eigenschaften(i):
    return i.hi, i.lo, i.kooling, i.indepSize, i.thresh, i.iterations

class Fonseca(object):
  "Fonseca"
  def __init__(i,hi=4,lo=−4, basehi=5, baselo=−5, kooling=1.99, indepSize=3,
               thresh=1e−2, iterations=2000):
    i.hi, i.lo, i.basehi, i.baselo, i.kooling, i.indepSize, i.thresh, i.iterations= \
    hi, lo, basehi, baselo, kooling, indepSize, thresh, iterations
    random.seed()
  def f1(i,x):
    return (1−exp**np.sum([(x[z]−1/((i.indepSize)**0.5)) \
                           for z in xrange(i.indepSize)]))
  def f2(i,x):
    return (1−exp**np.sum([(x[z]+1/((i.indepSize)**0.5)) \
                           for z in xrange(i.indepSize)]))
  def score(i,x):
    return i.f1(x)−i.f2(x)
  def eigenschaften(i):
    return i.hi, i.lo, i.kooling, i.indepSize, i.thresh, i.iterations

class ZDT1(object):
  "ZDT1"
  def __init__(i,hi=1,lo=0, basehi=1, baselo=0, kooling=7e−3, indepSize=30,
               thresh=1e−2, iterations=2000):
    i.hi, i.lo, i.basehi, i.baselo, i.thresh= hi, lo, basehi, baselo, thresh
    i.kooling, i.indepSize, i.iterations= kooling, indepSize, iterations
    random.seed()
  def f1(i,x):
    return x[0]
  def g(i,x):
    return (1+9*(np.sum(x[1:]))/(i.indepSize−1))
  def f2(i,x):
    return i.g(x)*(1−sqrt(x[0]/i.g(x)))
  def score(i,x):
    return (i.f1(x)−i.f2(x))
  def eigenschaften(i): # German for features
    return i.hi, i.lo, i.kooling, i.indepSize, i.thresh, i.iterations

class ZDT3(object):
  "ZDT3"
  def __init__(i,hi=1,lo=0, basehi=1, baselo=0, kooling=7e−3, indepSize=30,
               thresh=1e−2, iterations=2000):
    i.hi, i.lo, i.basehi, i.baselo, i.thresh = hi, lo, basehi, baselo, thresh
    i.kooling, i.indepSize, i.iterations = kooling, indepSize, iterations
    random.seed()
  def f1(i,x):
    return x[0]
  def g(i,x):
    return (1+9*(np.sum(x[1:]))/(i.indepSize−1))
  def f2(i,x):
    return i.g(x)*(1−(x[0]/i.g(x))**0.5−(x[0]/i.g(x))*sin(10*math.pi*x[0]))
  def score(i,x):
    return (i.f1(x)−i.f2(x))
  def eigenschaften(i): # German for features
    return i.hi, i.lo, i.kooling, i.indepSize, i.thresh, i.iterations


class Viennet3(object):
  "Viennet3"
  def __init__(i,hi=1,lo=0, basehi=1, baselo=0, kooling=7e−3, indepSize=2,
               thresh=1e−2, iterations=2000):
    i.hi, i.lo, i.basehi, i.baselo, i.thresh =  hi, lo, basehi, baselo, thresh
    i.kooling, i.indepSize, i.iterations= kooling, indepSize, iterations
    random.seed()
  def f1(i,x):
    return 0.5*x[0]**2+x[1]**2+sin(x[0]**2+x[1]**2)
  def f2(i,x):
    return (3*x[0]−2*x[1]+4)**2/8+(x[0]−x[1]+1)**2/27+15
  def f3(i,x):
    return 1/(x[0]**2+x[1]**2+1)−1.1*exp**(−x[0]**2−x[1]**2)
  def score(i,x):
    return (i.f1(x)−i.f2(x)−i.f3(x))
  def eigenschaften(i): # German for features
    return i.hi, i.lo, i.kooling, i.indepSize, i.thresh, i.iterations

class DTLZ7(object):
  "DTLZ7"
  def __init__(self,hi=1,lo=0, basehi=1, baselo=0, kooling=7e−3, indepSize=20,
               thresh=1e−2, iterations=2000):
    self.hi, self.lo = hi, lo
    self.basehi, self.baselo, self.thresh =  basehi, baselo, thresh
    self.kooling, self.indepSize, self.iterations= kooling, indepSize, iterations
    random.seed()
  def g(self,x):
    return 1+9/(self.indepSize)*np.sum(x)
  def h(self,x):
    return self.indepSize-np.sum([x[z]*(1+math.sin(3*math.pi*x[z]))/(1+self.g(x))
                                  for z in xrange(self.indepSize-2)])
  def f(self,x):
    F=x[:-1]
    F.append((1+self.g(x))*self.h(x))
    return F
  def score(self,x):
    return np.sum(self.f(x))
  def eigenschaften(self): # German for features
    return self.hi, self.lo, self.kooling, self.indepSize, self.thresh, self.iterations
```