2 Software Usage

2.1 Overview

Jocx is a parameterized model inference framework based on hidden Markov models. It's constructed on the foundation of coalescence theory with the key approximation that the distribution of local genealogies is Markovian along the sequence alignment. It combines CoalHMM modeling and several black-box style optimization subroutines with a special focus on heuristic-based evolution optimizers. The software package is available on GitHub at the following URL.

```
https://github.com/jade-cheng/Jocx.git
```

Jocx executes CoalHMM by specifying a model and an optimizer. It uses sequence alignments in the format of ziphmm directories, which is also prepared by Jocx. The program prints to standard output the progression of the estimated parameters and the corresponding log likelihood. The source package contains a set of Python files, and it requires no installation.

2.2 Input Data

Jocs takes pairs of aligned sequences as input. The number of sequence pairs depends on the CoalHMM model specified for a particular execution. We will discuss CoalHMM model specification later in this document. For example, for inference in a two-population isolation scenario, we need a minimal of one pair of aligned sequences, one from each of the two population. The sequences form an alignment so they need to match in length and names.

```
$ ls
a.fasta  b.fasta

$ cat a.fasta | wc -c
1827

$ cat b.fasta | wc -c
1827

$ head *.fasta -n 7
==> a.fasta <==
>1
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaAaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaTTaaaaaaaaaaaaaaaaaaaaaaaa

>2
aaaTaaaaaaaaaaaaaaaaaaaaaaAaCaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaAaaaaaaaaaaaaaaaaaaaaaaaaa
==> b.fasta <==
```

```
>1
aAaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaGaaaaaaGaaaaaaaaaaaaaaaaaaa

>2
aaaaaaaaaaCaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaTaaaaaaaaaTaaaaaa
```

2.3 ZipHMM Preparation

We use the ZipHMM algorithm [ref(admix-coalhmm45)] to calculate forward likelihoods. ZipHMM is shown in previous experiments to give us a speedup in computing the likelihood of one or two orders of magnitude when analyzing full genome alignments. To proceed to CoalHMM analysis, user need to prepare the ZipHMM data directories given pairwise alignments.

This preparation step is customized for each CoalHMM model. In the aforementioned two-population isolation scenario, we need two aligned sequences in Fasta format. Executing the following command given the two sequences results in a ZipHMM directory.

```
$ ls
a.fasta  b.fasta

$ Jocx.py init . iso a.fasta b.fasta
# Creating directory: ./ziphmm_iso_a_b
# creating uncompressed sequence file
# using output directory "./ziphmm_iso_a_b"
# parsing "a.fasta"
# parsing "b.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_iso_a_b/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_iso_a_b/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_iso_a_b/1.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_iso_a_b/2.ziphmm
```

The first command-line argument, init, initializes alignments and zips sequences that are needed to run an experiment. This command must be executed once before using the run command. The second argument specified the location of the output ZipHMM directory.

```
$ ls
a.fasta  b.fasta  ziphmm_iso_a_b

$ find ziphmm_iso_a_b/
ziphmm_iso_a_b/
ziphmm_iso_a_b/1.ziphmm
```

```
ziphmm_iso_a_b/1.ziphmm/data_structure
ziphmm_iso_a_b/1.ziphmm/nStates2seq
ziphmm_iso_a_b/1.ziphmm/nStates2seq/5.seq
ziphmm_iso_a_b/1.ziphmm/original_sequence
ziphmm_iso_a_b/2.ziphmm
ziphmm_iso_a_b/2.ziphmm/nStates2seq
ziphmm_iso_a_b/2.ziphmm/nStates2seq/5.seq
ziphmm_iso_a_b/2.ziphmm/data_structure
ziphmm_iso_a_b/2.ziphmm/original_sequence
```

The third argument, iso, specifies the CoalHMM model of interest. To see the list of all support models, we use the help argument. Here iso represent the two-population two-sequence isolation scenario, shown below.

```
$ Jocx.py --help
:
ISOLATION MODEL (iso)
    *
   / \ tau
  A   B

3 params -> tau, coal_rate, recomb_rate
2 seqs   -> A, B
1 group  -> AB
:
```

2.3.1 Sequence Order

The two-population isolation demographic model is symmetric, so the order of input Fasta sequences do not matter. This is not always the case. For example, in a three-population admix model, shown below, the roles populations take are different. Population C is admixed, and it's formed from ancestral siblings of the two source populations, A and B. The order of input Fasta sequences, therefore, needs to match.

```
$ Jocx.py --help
:
THREE POP ADMIX 2 3 MODEL (admix23)
                  *
                 / \     greedy1_time_1a
buddy23_time_1a /\  \
               /  \_/\   buddy23_time_2a
   admix_prop /  <-|  \  iso_time
             A      C   B

7 params -> iso_time,        buddy23_time_1a,
            buddy23_time_2a, greedy1_time_1a,
            coal_rate, recomb_rate, admix_prop
3 seqs   -> A, B, C
3 groups -> AC, BC, AB
:
```

The init command's execution takes aligned Fasta sequences following the order specified above, i.e. one sequence from population A, followed by one sequence from population B, followed by one sequence from population C.

```
$ ls
a1.fasta  b1.fasta  c1.fasta

$ Jocx.py init . admix23 a1.fasta b1.fasta c1.fasta
# Creating directory: ./ziphmm_admix23_a_c
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23_a_c"
# parsing "a1.fasta"
# parsing "c1.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23_a_c/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23_a_c/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23_a_c/2.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23_a_c/1.ziphmm
# Creating directory: ./ziphmm_admix23_b_c
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23_b_c"
# parsing "b1.fasta"
# parsing "c1.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23_b_c/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23_b_c/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23_b_c/1.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23_b_c/2.ziphmm
# Creating directory: ./ziphmm_admix23_a_b
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23_a_b"
# parsing "a1.fasta"
# parsing "b1.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23_a_b/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23_a_b/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23_a_b/2.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23_a_b/1.ziphmm

$ ls
a1.fasta  b1.fasta  c1.fasta
ziphmm_admix23_a_b  ziphmm_admix23_a_c  ziphmm_admix23_b_c

$ find ziphmm_admix23_*
ziphmm_admix23_a_b
```

```
ziphmm_admix23_a_b/2.ziphmm
ziphmm_admix23_a_b/2.ziphmm/data_structure
ziphmm_admix23_a_b/2.ziphmm/nStates2seq
ziphmm_admix23_a_b/2.ziphmm/nStates2seq/5.seq
ziphmm_admix23_a_b/2.ziphmm/original_sequence
ziphmm_admix23_a_b/1.ziphmm
ziphmm_admix23_a_b/1.ziphmm/nStates2seq
ziphmm_admix23_a_b/1.ziphmm/nStates2seq/5.seq
ziphmm_admix23_a_b/1.ziphmm/data_structure
ziphmm_admix23_a_b/1.ziphmm/original_sequence
ziphmm_admix23_a_c
ziphmm_admix23_a_c/2.ziphmm
ziphmm_admix23_a_c/2.ziphmm/data_structure
ziphmm_admix23_a_c/2.ziphmm/nStates2seq
ziphmm_admix23_a_c/2.ziphmm/nStates2seq/5.seq
ziphmm_admix23_a_c/2.ziphmm/original_sequence
ziphmm_admix23_a_c/1.ziphmm
ziphmm_admix23_a_c/1.ziphmm/data_structure
ziphmm_admix23_a_c/1.ziphmm/nStates2seq
ziphmm_admix23_a_c/1.ziphmm/nStates2seq/5.seq
ziphmm_admix23_a_c/1.ziphmm/original_sequence
ziphmm_admix23_b_c
ziphmm_admix23_b_c/2.ziphmm
ziphmm_admix23_b_c/2.ziphmm/data_structure
ziphmm_admix23_b_c/2.ziphmm/nStates2seq
ziphmm_admix23_b_c/2.ziphmm/nStates2seq/5.seq
ziphmm_admix23_b_c/2.ziphmm/original_sequence
ziphmm_admix23_b_c/1.ziphmm
ziphmm_admix23_b_c/1.ziphmm/nStates2seq
ziphmm_admix23_b_c/1.ziphmm/nStates2seq/5.seq
ziphmm_admix23_b_c/1.ziphmm/data_structure
ziphmm_admix23_b_c/1.ziphmm/original_sequence
```

2.3.2 Sequences per Population

In the two examples above, each population contributes one and only one sequence to the CoalHMM model's construction. Jocx also has models that support two sequences per population. This is to better take advantage of the genomic data that's available to the researcher.

```
$ Jocx.py --help
  :
  THREE POP ADMIX 2 3 MODEL 6 HMM (admix23-6hmm)
                     *
                    / \     greedy1_time_1a
  buddy23_time_1a /\  \
                 /  \_/\   buddy23_time_2a
     admix_prop /  <-|  \  iso_time
                A1   C1   B1
                A2   C2   B2

  7 params -> iso_time,        buddy23_time_1a,
```

```
            buddy23_time_2a, greedy1_time_1a,
            coal_rate, recomb_rate, admix_prop
  6 seqs   -> A1, A2, B1, B2, C1, C2
  6 groups -> A1C1, B1C1, A1B1, A1A2, B1B2, C1C2
  :
```

In this example, we demonstrate the same admixture demographic model but with each population contributing two sequences to form six pairwise alignments, which are then used to construct six HMMs for the inference.

```
$ ls
a1.fasta  a2.fasta  b1.fasta  b2.fasta  c1.fasta  c2.fasta

$ Jocx.py init . admix23-6hmm a1.fasta a2.fasta b1.fasta b2.fasta c1.fasta c2.fasta
# Creating directory: ./ziphmm_admix23-6hmm_a1_c1
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23-6hmm_a1_c1"
# parsing "a1.fasta"
# parsing "c1.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_a1_c1/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_a1_c1/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_a1_c1/2.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_a1_c1/1.ziphmm
# Creating directory: ./ziphmm_admix23-6hmm_b1_c1
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23-6hmm_b1_c1"
# parsing "b1.fasta"
# parsing "c1.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_b1_c1/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_b1_c1/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_b1_c1/1.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_b1_c1/2.ziphmm
# Creating directory: ./ziphmm_admix23-6hmm_a1_b1
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23-6hmm_a1_b1"
# parsing "a1.fasta"
# parsing "b1.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_a1_b1/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_a1_b1/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_a1_b1/2.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_a1_b1/1.ziphmm
# Creating directory: ./ziphmm_admix23-6hmm_a1_a2
```

```
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23-6hmm_a1_a2"
# parsing "a1.fasta"
# parsing "a2.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_a1_a2/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_a1_a2/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_a1_a2/1.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_a1_a2/2.ziphmm
# Creating directory: ./ziphmm_admix23-6hmm_b1_b2
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23-6hmm_b1_b2"
# parsing "b1.fasta"
# parsing "b2.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_b1_b2/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_b1_b2/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_b1_b2/1.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_b1_b2/2.ziphmm
# Creating directory: ./ziphmm_admix23-6hmm_c1_c2
# creating uncompressed sequence file
# using output directory "./ziphmm_admix23-6hmm_c1_c2"
# parsing "c1.fasta"
# parsing "c2.fasta"
# comparing sequence "1"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_c1_c2/1.ziphmm"
# comparing sequence "2"
# sequence length: 900
# creating "./ziphmm_admix23-6hmm_c1_c2/2.ziphmm"
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_c1_c2/1.ziphmm
# Creating 5-state alignment in directory: ./ziphmm_admix23-6hmm_c1_c2/2.ziphmm

$ ls
a1.fasta  b1.fasta  c1.fasta
a2.fasta  b2.fasta  c2.fasta
ziphmm_admix23-6hmm_a1_a2  ziphmm_admix23-6hmm_a1_c1  ziphmm_admix23-6hmm_b1_c1
ziphmm_admix23-6hmm_a1_b1  ziphmm_admix23-6hmm_b1_b2  ziphmm_admix23-6hmm_c1_c2
```

2.4 MLE Optimization

Jocx implements three optimization subroutines, Nelder-Mead (NM), Genetic Algorithm (GA), and Particle Swarm Optimization (PSO). After preparing the ZipHMM

directories, user can choose to proceed to CoalHMM parametrized model inference using one of these three black-box optimizers.

We use the two-population isolation CoalHMM as an example in this sections. The first command-line argument, run, runs an experiment. Before running an experiment, the appropriate directories must be initialized using the init command. The second argument specifies the directory where the ZipHMM data directories can be found. The third argument, iso, specified the CoalHMM model of interest. The ZipHMM directories found in the given directory need to match with the model specification. The fourth argument, nm, ga, or pso, specifies the optimization method used for the execution.

The rest of the arguments are used to determine the initial parameter condition. In NM they are the initial simplex vertices. In PSO and GA, the first generation of solutions are sampled from the range +/- 100x of the given values. The length of these parameters matches the specified CoalHMM model.

```
$ Jocx.py --help
:
3 params -> tau, coal_rate, recomb_rate
:
```

In the two-population isolation model, represented by the iso argument, we infer three model parameters, the population split time, the coalescent rate, and the recombination rate. Populations are modeled to have the same coalescent rate. The order of these three parameters follows the ASCII diagram printed using the help argument, shown above. For this model we first have the split time, then the coalescent rate, and finally the recombination rate.

2.4.1 NM

NM was introduced by John Nelder and Roger Mead in 1965 [ref(thesis25)] as a technique to minimize an objective function in a many-dimensional space. This method uses several algorithm coefficients to determine the amount of effect of possible actions. They are the reflection coefficient r, the expansion coefficient c, the contraction coefficient g, and the shrinkage coefficient s. Standard values recommended in [ref(thesis3)] are r = 1, c = 2, g = 1/2, and s = 1/2.

```
$ Jocx.py run . iso nm 0.0001 1000 0.1
# algorithm            = _NMOptimiser
# timeout              = None
# max_executions       = 1
#
# 2017-10-11 11:29:08.069462
:
# execution state score param0 param1 param2
0 init -38.2023478685 0.000376954454165 7480.36836670 0.337649514816
1 fmin-in -40.5337262711 0.000385595244114 661.208520686 0.920281817958
1 fmin-cb -40.3804021200 0.000385595244114 694.268946721 0.920281817958
:
```

```
1 fmin-cb -37.8927822292 0.000695082517418 200504630.601 32081.6528250
Optimization terminated successfully.
     Current function value: 37.892782
     Iterations: 262
     Function evaluations: 533
1 fmin-out -37.8927822292 0.000695082517418 200504630.601 32081.652825
```

In the output of NM's execution, we have a final report of whether or not the execution was successful together with the optimal solution. Different reasons contribute to the failure of an execution. The number of parameters to solve is a major reason. When CoalHMM models become complex, the number of parameters increase to a level beyond NM's capability. Initial parameters is another major reason for failed NM executions.

2.4.2 GA

GA was introduced by John Holland first introduced in the 1970s [ref(thesis13)]. The idea is to encode each solution as a chromosome-like data structure and operate on them through actions analogous to genetic alterations, which usually involves selection, recombination, and mutation. For each type of alteration, people have developed different techniques.

```
$ Jocx.py run . iso ga 0.0001 1000 0.1
# algorithm            = _GAOptimiser
# timeout              = None
# elite_count          = 1
# population_size      = 50
# initialization       = UniformInitialisation
# selection            = TournamentSelection
# tournament_ratio     = 0.1
# selection_ratio      = 0.75
# mutation             = GaussianMutation
# point_mutation_ratio = 0.15
# mu                   = 0.0
# sigma                = 0.01
#
# 2017-10-23 10:31:32.821761
#
# param0 = (1.0000000000000016e-05, 0.001)
# param1 = (99.99999999999996, 10000.0)
# param2 = (0.009999999999999995, 1.0)
#
#
# POPULATION FOR GENERATION 1
# average_fitness = -5.32373335161
# min_fitness     = -10.7962322739
# max_fitness     = -0.613544122419
#
# gen idv       fitness      param0         param1      param2
    1   1   -0.61354412  0.00002825  6305.95175380  0.04139445
    1   2   -1.38710619  0.00004282  2182.61708962  0.03027973
```

```
    1   3   -4.45085424  0.00001133   254.73764392  0.01081756
    1   4   -9.37092993  0.00067074   116.84983427  0.13757425
    1   5  -10.79623227  0.00071728   142.34535478  0.81564586
    :
#
# POPULATION FOR GENERATION 2
# average_fitness = -5.83495296756
# min_fitness     = -10.5697879572
# max_fitness     = -0.613544122419
#
# gen idv       fitness      param0         param1      param2
    2   1   -0.61354412  0.00002825  6305.95175380  0.04139445
    2   2   -0.61382451  0.00002825  6305.95175380  0.13757425
    2   3   -6.89850999  0.00002825   116.84983427  0.14110664
    2   4  -10.47909826  0.00067074   145.01523656  0.81564586
    2   5  -10.56978796  0.00067074   142.34535478  0.81564586
    :
:
```

In the output of GA's execution, we have multiple generations of solutions, and multiple solutions per generation. Solutions in each generation is ordered by the fitness, i.e. best solution is at the top. The final solution is, therefore, the first solution in the last generation.

2.4.3 PSO

PSO was introduced by Eberhart and Kennedy in 1995 [ref(thesis6)] as an optimization technique relying on stochastic processes, similar to GA. As its name implies, each individual solution mimics a particle in a swarm. Each particle holds a velocity and keeps track of the best positions it has experienced and best position the warm has experienced. The former encapsulates the social influence, i.e. a force pulling towards the swarm's best. The latter encapsulates the cognitive influence, i.e. a force pulling towards the particle's best. Both forces act on the velocity and drive the particle through a hyper parameter space.

```
$ Jocx.py run . iso pso 0.0001 1000 0.1
# algorithm            = _PSOptimiser
# timeout              = None
# max_iterations       = 50
# particle_count       = 50
# max_initial_velocity = 0.02
# omega                = 0.9
# phi_particle         = 0.3
# phi_swarm            = 0.1
#
# 2017-10-23 10:32:29.123305
#
# param0 = (1.0000000000000016e-05, 0.001)
# param1 = (99.99999999999996, 10000.0)
# param2 = (0.009999999999999995, 1.0)
```

```
#
#
# PARTICLES FOR ITERATION 1
# swarm_fitness           = -0.832535308472
# best_average_fitness    = -4.40169918533
# best_minimum_fitness    = -9.77654933959
# best_maximum_fitness    = -0.832535308472
# current_average_fitness = -4.40169918533
# current_minimum_fitness = -9.77654933959
# current_maximum_fitness = -0.832535308472
#
#                                          best-   best-     best-     best-
# gen idv  fitness  param0   param1  param2 fitness param0    param1    param2
    1   0  -0.83  0.000044  4619.31  0.20   -0.83   0.000044  4619.31   0.20
    1   1  -0.86  0.000048  4502.80  0.26   -0.86   0.000048  4502.80   0.26
    1   2  -0.89  0.000061  4669.48  0.58   -0.89   0.000061  4669.48   0.58
    1   3  -1.10  0.000035  2970.77  0.31   -1.10   0.000035  2970.77   0.31
    1   4  -1.46  0.000057  2148.93  0.15   -1.46   0.000057  2148.93   0.15
    :
#
# PARTICLES FOR ITERATION 2
# swarm_fitness           = -0.810479293858
# best_average_fitness    = -4.02436023707
# best_minimum_fitness    = -9.12434788412
# best_maximum_fitness    = -0.810479293858
# current_average_fitness = -4.02984771812
# current_minimum_fitness = -9.12434788412
# current_maximum_fitness = -0.810479293858
#
#                                          best-   best-     best-     best-
# gen idv  fitness  param0   param1  param2 fitness param0    param1    param2
    2   0  -0.81  0.000045  4854.87  0.25   -0.81   0.000045  4854.87   0.25
    2   1  -0.82  0.000040  4622.38  0.21   -0.82   0.000040  4622.38   0.21
    2   2  -0.91  0.000064  4599.97  0.59   -0.89   0.000061  4669.48   0.58
    2   3  -1.12  0.000038  2917.40  0.29   -1.10   0.000035  2970.77   0.31
    2   4  -1.39  0.000058  2308.29  0.14   -1.39   0.000058  2308.29   0.14
    :
:
```

In the output of the PSO's execution, we have multiple generations and multiple particles (solutions) per generation. Each particle contains two sets of solutions, the current solution and the best solution that this particle has encountered throughout the PSO's execution. The latter is always better than the former. Similar to GA, each generation is ordered by the particles' fitness. The final solution is, therefore, the second solution of the first particle in the last generation.