Package ‘Rmalschains’

November 29, 2016

Maintainer  Christoph Bergmeir <c.bergmeir@decsai.ugr.es>
License  GPL-3 | file LICENSE
Title  Continuous Optimization using Memetic Algorithms with Local Search Chains (MA-LS-Chains) in R
LinkingTo  Rcpp
Type  Package
LazyLoad  yes
Author  Christoph Bergmeir, Daniel Molina, José M. Benítez
Description  An implementation of an algorithm family for continuous optimization called memetic algorithms with local search chains (MA-LS-Chains). Memetic algorithms are hybridizations of genetic algorithms with local search methods. They are especially suited for continuous optimization.
Version  0.2-3
URL  http://sci2s.ugr.es/dicits/software/Rmalschains
Date  2016-11-29
Depends  Rcpp (>= 0.9.10)
Suggests  inline
Encoding  UTF-8
Collate  'malschains.R' 'Rmalschains-package.R'
RoxygenNote  5.0.1
NeedsCompilation  yes
Repository  CRAN
Date/Publication  2016-11-29 08:09:54

R topics documented:

  Rmalschains-package .............................................. 2
  malschains .......................................................... 5
  malschains.control .................................................. 6
  print.malschains ..................................................... 8
Description

This package implements an algorithm family for continuous optimization called memetic algorithms with local search chains (MA-LS-Chains).

Details

One of the main issues to optimize a real-coded function is the capability of the algorithm to realize a good exploration of the search space and, at the same time, to exploit the most promising region to obtain accurate solutions.

Memetic algorithms are hybridizations of genetic algorithms with local search methods. They are especially suited for continuous optimization, as they combine the power of evolutionary algorithms to explore the search space with a local search method to find the local optimum of a promising region. In these algorithms, it is recommended to increase the effort invested in the local search (measured in number of evaluations, called intensity) in the improvement of the most promising solution. However, it is not easy to decide the right intensity for each solution.

MA-LS-Chains is a steady-state memetic algorithm, which combines a steady-state genetic algorithm with various different local search methods. In contrast to the generational approach, where all individuals are substituted in an iteration, in the steady-state genetic algorithm in each iteration only one solution, the worst one, is substituted in the population. This makes it possible to not lose the improvement obtained by the local search over the individuals.

For MA-LS-Chains, the current state of the local search algorithm is stored along with the individuals. So, it becomes possible to run the local search a fixed number of iterations, stop it, and possibly later continue the previous local search over the same individual. In this way, MA-LS-Chains controls the application of the local search to the most promising solutions.

The package implements various different local search strategies:

- CMA-ES The Covariance Matrix Adaptation Evolution Strategy
- SW A Solis Wets solver
- SSW Subgrouping Solis Wets
- Simplex

CMA-ES is a very effective local search strategy, but quite complicated, and it does not scale well if the amount of parameters to optimize is huge. The Solis Wets solver is pretty simple and therewith fast. The SSW strategy is an adapted version of the Solis Wets solver for high dimensional data, so that the algorithm with this type of local search scales well with the dimensionality of the data. It applies the Solis Wets solver to randomly chosen subgroups of variables (Subgrouping Solis Wets).

All the local search methods can also be used directly, without making use of the evolutionary algorithm.

The package contains some demos illustrating its use. To get a list of them, type:
library(Rmalschains)
demo()

The demos currently available are claw, rastrigin, sphere, rastrigin_highDim, and rastrigin_inline. So in order to, e.g., execute the claw demo, type
demo(claw)

All algorithms are implemented in C++, and they run pretty fast. A usual processing to speed up optimization is to implement the objective function also in C/C++. However, a bottleneck in this approach is that the function needs to be passed as an R function, so that the optimizer needs to go back from C++ to R to C/C++ in each call of the target function. The package provides an interface which allows to pass the C/C++ target function directly as a pointer. See the rastrigin_inline demo for how to do that. The demo also shows how an environment can in this approach be used to pass additional parameters to the target function.

For theoretical background of the algorithm, the reader may refer to the cited literature, where the algorithms where originally proposed.

Additional information is also available at the project website:
http://sci2s.ugr.es/dicits/software/Rmalschains

Author(s)
Christoph Bergmeir <c.bergmeir@decsai.ugr.es>
Daniel Molina <dmolina@decsai.ugr.es>
José M. Benítez <j.m.benitez@decsai.ugr.es>

Additional information is also available at our group’s website of continuous optimization:
http://sci2s.ugr.es/EAMHCO/

References

See Also
malschains, malschains.control
Examples

# Example for maximization of the claw function

claw <- function(xx) {
  x <- xx[1]
  y <- (0.46 * (dnorm(x, -1, 2/3) + dnorm(x, 1, 2/3)) +
       (1/300) * (dnorm(x, -0.5, 0.01) + dnorm(x, -1, 0.01) + dnorm(x, -1.5, 0.01)) +
       (7/300) * (dnorm(x, 0.5, 0.07) + dnorm(x, 1, 0.07) + dnorm(x, 1.5, 0.07))
  return(y)
}

# Use MA-CMA-Chains
res.claw <- malschains(function(x) {claw(x)}, lower=c(-3), upper=c(3),
                       maxEvals=50000, control=malschains.control(popsize=50,
                                                     istep=300, ls="cmaes", optimum=-5))

## Not run:
# Use only the CMA-ES local search
res.claw2 <- malschains(function(x) {claw(x)}, lower=c(-3), upper=c(3), verbosity=0,
                        maxEvals=50000, control=malschains.control(ls="cmaes",
                                                      lsOnly=TRUE, optimum=-5))

# Use only the Simplex local search
res.claw3 <- malschains(function(x) {claw(x)}, lower=c(-3), upper=c(3), verbosity=0,
                        maxEvals=50000, control=malschains.control(ls="simplex",
                                                      lsOnly=TRUE, optimum=-5))

x <- seq(-3, 3, length=1000)
claw_x <- NULL
for (i in 1:length(x)) claw_x[i] <- claw(x[i])

plot(x, claw_x, type="l")
points(res.claw$sol, -res.claw$fitness, col="red")
points(res.claw2$sol, pch=3, -res.claw2$fitness, col="blue")
points(res.claw3$sol, pch=3, -res.claw3$fitness, col="green")

# Example for the rastrigin function

rastrigin <- function(x) {
  dimension <- length(x)
  res <- 0.0
  for (i in 1:dimension) {
    res <- res + (x[i]^2 - 10*cos(2*pi*x[i]) + 10)
  }
  return(res)
}
Perform optimization with the MA-LS-Chains algorithm

Description

This is the main function of the package. It minimizes the output of the function fn (for maximization, change the sign of the output of fn).

Usage

malschains(fn, lower, upper, dim, maxEvals = 10 * control$istep, verbosity = 2, initialpop = NULL, control = malschains.control(), seed = NULL, env)

Arguments

- **fn**: The function to minimize.
- **lower**: The lower bound (or bounds) of the search domain.
- **upper**: The upper bound (or bounds) of the search domain.
- **dim**: The dimension of the problem (if lower and upper are vectors it is not needed).
- **maxEvals**: The maximal number of evaluations of the fitness function.
- **verbosity**: Set the verbosity level. Currently, meaningful values are 0, 1, 2
- **initialpop**: An initial population for the evolutionary algorithm can be submitted (as a matrix). Here, prior knowledge can be introduced to get better results from the algorithm.
control A list containing the main options of the algorithm. See \texttt{malschains.control}.

seed A seed value for the random number generator.

env The environment in which to evaluate the fitness function. If not given, it is generated.

Details

The output of the function when run with \texttt{verbosity}=2 is the following:

- \texttt{EA::PopFitness} The fitness of the best, the one at the 1st quartile, the one at the 3rd quartile, and the worst individual.
- \texttt{EA::Improvement} Improvement of the individuals at the according ranked positions in the population (best, 1st quartile, 3rd quartile, worst).
- \texttt{LS} The number of the individual which is improved on (in braces), its fitness before and after application of the LS procedure, and their difference.
- \texttt{EABest} If the best fitness present in the population changed: same as LS.

Value

The function returns a list containing the best individual, sol, and its fitness. Furthermore, it contains some information on the optimization process, which can be seen using \texttt{print.malschains}.

References


\textbf{malschains.control} \hspace{1cm} \textit{Sets and initializes the main parameters of the algorithm}

\section*{Description}

This is a function that initializes and sets the parameters of the algorithm. It generates a list of parameters, to be used with the \texttt{malschains} function.

\section*{Usage}

\begin{verbatim}
  malschains.control(popsize = 50, ls = "cmaes", istep = 500,
                      effort = 0.5, alpha = 0.5, optimum = -Inf, threshold = 1e-08,
                      lsOnly = FALSE, lsParam1 = 0, lsParam2 = 0)
\end{verbatim}
malschains.control

Arguments

- **popsize**: The population size of the evolutionary algorithm.
- **ls**: The local search method. Currently implemented are cmaes, sw, simplex, and ssw. Usually, the cmaes local search strategy will give good results. However, it does not scale well with the problem size. So, if performance is needed, the sw strategy is a better choice. If the problem is high-dimensional, the ssw strategy is promising, which selects randomly 20% of the variables for optimization.
- **istep**: The number of iterations of the local search. I.e., if the local search is started or re-started on an individual, it will be executed for an istep number of iterations. This parameter depends on the local search used. For cmaes, usually an istep of 300 is a good choice. For the other local search methods, an istep of 100 performs better.
- **effort**: A value between 0 and 1 which gives the ratio between the amount of evaluations that are used for the local search and for the evolutionary algorithm, respectively. A higher effort means more evaluations for the evolutionary algorithm. So, if exploration of the search space is more important than finding local optima, effort is to be chosen higher.
- **alpha**: The alpha parameter from crossover BLX-alpha. A lower value (< 0.3) reduces diversity, a higher value increases diversity.
- **optimum**: The optimum to achieve. The default is zero, as in many minimization problems a value of zero can be considered optimal.
- **threshold**: A threshold which defines for the local search how much improvement is considered as no improvement. If this value is chosen too low (zero), then the local search will usually always try to improve on the best individual, even if it is already located very close to a local optimum.
- **lsOnly**: Apply only the local search algorithm, and not MA-LS-Chains
- **lsParam1**: First (optional) parameter. Currently, if local search is cmaes, this is the parameter popsize/lambda of cmaes. If it is not set or not positive, cmaes will calculate this automatically using a heuristic.
- **lsParam2**: Second (optional) parameter. Currently, if local search is cmaes, this is the parameter parentssize/mu of cmaes. If it is not set, not positive, or not smaller lambda, cmaes will calculate this automatically using a heuristic.

References


Description

Print out some characteristics of a `malschains` result. The result shows besides the best solution and its fitness the total number of evaluations spent for both EA and LS, the ratio of the spent evaluations (also called effort), the ratio of total improvement of the fitness, the percentage of times that application of the EA/LS yielded improvement, and some timing results in milliseconds.

Usage

```r
## S3 method for class 'malschains'
print(x, ...)
```

Arguments

- `x`: the `malschains` result
- `...`: additional function parameters (currently not used)
Index

*Topic MA-LS-Chains
  Rmalschains-package, 2

*Topic optimization,
  Rmalschains-package, 2

malschains, 3, 5, 6, 8
malschains.control, 3, 6, 6

print.malschains, 6, 8

Rmalschains (Rmalschains-package), 2
Rmalschains-package, 2