Package ‘FLSSS’

August 23, 2024

Type Package

Title Mining Rigs for Problems in the Subset Sum Family

Version 9.1.8

Description Specialized solvers for combinatorial optimization problems in the Subset Sum family. The solvers differ from the mainstream in the options of (i) restricting subset size, (ii) bounding subset elements, (iii) mining real-value multisets with predefined subset sum errors, (iv) finding one or more subsets in limited time. A novel algorithm for mining the one-dimensional Subset Sum induced algorithms for the multi-Subset Sum and the multidimensional Subset Sum. The multi-threaded framework for the latter offers exact algorithms to the multidimensional Knapsack and the Generalized Assignment problems. Historical updates include (a) renewed implementation of the multi-Subset Sum, multidimensional Knapsack and Generalized Assignment solvers; (b) availability of bounding solution space in the multidimensional Subset Sum; (c) fundamental data structure and architectural changes for enhanced cache locality and better chance of SIMD vectorization; (d) option of mapping floating-point instance to compressed 64-bit integer instance with user-controlled precision loss, which could yield substantial speedup due to the dimension reduction and efficient compressed integer arithmetic via bit-manipulations; (e) distributed computing infrastructure for multidimensional subset sum; (f) arbitrary-precision zero-margin-of-error multidimensional Subset Sum accelerated by a simplified Bloom filter. The package contains a copy of xxHash from <https://github.com/Cyan4973/xxHash>. Package vignette (<doi:10.48550/arXiv.1612.04484>) detailed a few historical updates. Functions prefixed with 'aux' (auxiliary) are independent implementations of published algorithms for solving optimization problems less relevant to Subset Sum.

License GPL-3

Encoding UTF-8

ByteCompile true

Imports Rcpp (>= 0.12.13), RcppParallel

LinkingTo Rcpp, RcppParallel

SystemRequirements GNU make

NeedsCompilation yes

Repository CRAN

Author Charlie Wusuo Liu [aut, cre]

Maintainer Charlie Wusuo Liu <liuwusuo@gmail.com>

Date/Publication 2024-08-23 07:50:02 UTC

Contents

`addNumStrings` *Add numeric strings.*

Description

A test function for adding numeric strings.

Usage

```
addNumStrings(s)
```

Arguments

`s`	A vector of numeric strings.

Value

A numeric string.

Examples

```
addNumStrings(c("1.2345345", "-0.34534", "3.1415900"))
```

---

arbFLSSS *Multidimensional exact subset sum in arbitrary precision and magnitude*

Description

Given a multidimensional set and a subset size, find one or more subsets whose elements sum up to a given target.

Usage

```
arbFLSSS(
  len,
  V,
  target,
  givenKsumTable,
  solutionNeed = 1L,
  maxCore = 7L,
  tlimit = 60,
  approxNinstance = 1000L,
  ksumK = 4L,
  ksumTableSizeScaler = 30L,
  verbose = TRUE
  )
```

Arguments

`len`	An integer as the subset size. `1` <= `len` <= `nrow(V)`.
`V`	A string matrix as the superset. Rows are elements.
`target`	A string vector as the target subset sum. `length(target)` == `ncol(V)`.
`givenKsumTable`	Either `NULL` or the return value from `ksumHash()`. See argument `ksumK` for the preliminaries. If `NULL`, the function will compute and hash k-sums depending on `ksumK` before mining the subsets. Otherwise it will use `givenKsumTable` as the lookup table and ignore arguments `ksumK` and `ksumTableSizeScaler`.
`solutionNeed`	An integer. How many solutions are wanted. Default `solutionNeed` = `1`.
`maxCore`	An integer as the maximum threads to invoke. Better not exceed the number of logical processors on the platform. Default `maxCore` = `7`.
`tlimit`	A numeric value as the time limit (seconds). Default `tlimit` = `60`.
`approxNinstance`	An integer. The problem will be decomposed into about `approxNinstance` subproblems solved independently by the threads. Default `approxNinstance` = `1000`. `approxNinstance` is better to be much higher than argument `maxCore` since time costs of the subproblems are unknown and probably vary greatly.

ksumK — An integer. If ksumK < 3, no k-sum accelerator will be built. For example, if ksumK = 5, then the sums of all combinations of 3 elements (3-sums), the sums of all combinations of 4 elements (4-sums), and the sums of all combinations of 5 elements (5-sums) in V, are pre-computed and hashed into a Bloom filter variant. This filter thus contains 3 lookup tables responding to the 3-sums, 4-sums and 5-sums. During the main course of mining, if any set is reduced to one of size 3, 4, or 5, the set's associated target sum will be hashed and looked up in the filter. Not existing would imply the target sum is unreachable and the set can be discredited immediately. This typically generates massive speedup. For ksumK < 3, such filtering is not meaningful and thus not performed. A high ksumK coupled with a large superset V however is prone to memory overflow or extremely time consuming. ksumK will be upper-bounded by subset size len internally. Default ksumK = 4.

ksumTableSizeScaler — An integer for determining size of the k-sum lookup table in the filter described above. For example, a set of size 21 has 1330 3-element subsets. If ksumTableSizeScaler = 10, then around 13300 bits will be allocated for the 3-sum lookup table. The exact number of bits is 14033 + 7, where 14033 is the lowest element greater than 13300 in a prime array defined in GCC's STL of hashing policy, and 7 is to make up the last byte. Default ksumTableSizeScaler = 30. Higher ksumTableSizeScaler means lower chance of hash collision, thus higher efficiency.

verbose — A boolean value. TRUE prints the computing progress. Default TRUE.

Details

New users might want to check out FLSSS() or mFLSSSpar() first.

String matrix V is maximally compressed into an integer set of size nrow(V). Dimensionality of the set will be printed given verbose = TRUE. Each set element is a huge integer comprising many 64-bit buffers. Addition and subtraction of the huge integers call mpn_add_n() and mpn_sub_n() from the GNU Multiple Precision Arithmetic Library (GMP) if the system has it, otherwise they are performed by customized algorithms.

After the initial problem is decomposed, the smaller problems can collectively offer a pair of index lower and upper bounds. The k-subsets outside the bounds are not necessarily considered for building the k-sum accelerator.

See comparisons between this function and FLSSS(), mFLSSSpar() in Examples.

Value

A list of index vectors as solutions.

Examples

```
set.seed(1)
N = 200L # Superset size.
len = 20L # Subset size.
V = sapply(1:N, function(i) # Generate a set where every "number" has at most
  # 100 digits.
```

```
{
  a = 0:9
  left = sample(a, size = sample(50, 1), replace = TRUE)
  right = sample(a, size = sample(50, 1), replace = TRUE)
  x = paste0(paste0(left, collapse = ""), ".", paste0(right, collapse = ""))
  if (runif(1) < 0.5) x = paste0("-", x) # Randomly add a negative sign.
  x
})
str(V)


sol = sample(N, len) # Make a solution.
target = FLSSS::addNumStrings(V[sol]) # An unexposed helper function.


system.time({
  rst = FLSSS::arbFLSSS(
    len, V = as.matrix(V), target, solutionNeed = 1, maxCore = 2,
    tlimit = 10, ksumK = 0, verbose = TRUE)
})


# Validation.
all(unlist(lapply(rst, function(x) FLSSS:::addNumStrings(V[x]))) == target)




# ===============================================================================
# Mine in a multidimensional set.
# ===============================================================================
set.seed(2)
d = 4L # Set dimension.
N = 50L # Set size.
len = 10L # Subset size.
roundN = 4L # For rounding the numeric values before conversion to strings.


V = matrix(round(runif(N * d, -1, 1), roundN), nrow = N) # Make superset.
optionSave = options()
options(scipen = 999) # Ensure numeric-to-string conversion does not
# produce strings like "2e-3".
Vstr = matrix(as.character(V), nrow = N)


sol = sample(N, len) # Make a solution.
target = round(colSums(V[sol, ]), roundN) # Target subset sum.
targetStr = as.character(target)


system.time({
  rst = FLSSS::arbFLSSS(
    len = len, V = Vstr, target, givenKsumTable = NULL, tlimit = 60,
```

```
    solutionNeed = 1e9, maxCore = 2, ksumK = 4, verbose = TRUE)
})


# Validation.
all(unlist(lapply(rst, function(x)
{
  apply(Vstr, 2, function(u) FLSSS:::addNumStrings(u[x]))
})) == targetStr)




# # ===========================================================================
# # Compare arbFLSSS() and FLSSS(). Example takes more than 2 seconds. The
# # section has some analysis of the algorithms.
# # ===========================================================================
# set.seed(3)
# N = 100L # Superset size.
# len = 20L # Subset size.
# roundN = 5L # For rounding the numeric values.
# V = sort(round(100000 * runif(N, -1, 1), roundN)) # Create superset.
# sol = sort(sample(N, len)) # Make a solution.
# target = round(sum(V[sol]), roundN)
# error = 3e-6 # Effectively demands the target sum to be exactly matched
# # since roundN = 5.
#
#
# system.time({
#   FLSSSrst = FLSSS::FLSSS(
#     len, V, target, ME = error, solutionNeed = 2, tlimit = 60)
# })
# # It may seem counter-intuitive that this takes much longer than the instance
# # with N = 1000 and len = 200L --- the 1st example in the help page of
# # FLSSS(). Note the time cost is closely related to the "rarity" of
# # solutions. A larger superset or subset could mean more element combinations
# # that can sum into the given range, thus more solutions and easier to mine.
#
#
# # Validate the results.
# all(abs(unlist(lapply(FLSSSrst, function(x) sum(V[x]))) - target) <= error)
#
#
# options(scipen = 999)
# Vstr = as.matrix(as.character(V))
# targetStr = as.character(target)
# # Use 1 thread for a fair comparison with FLSSS() since the latter is
# # single-threaded. Use no k-sum accelerator.
# system.time({
#   arbFLSSSrst = FLSSS::arbFLSSS(
#     len, V = Vstr, target = targetStr, solutionNeed = 2, maxCore = 1,
#     ksumK = 0, verbose = TRUE, approxNinstance = 1000, tlimit = 60)
# })
```

```
# # Timing is higher than FLSSS() because arbFLSSS()'s objective
# # is not just solving unidimensional problem.
#
#
# # Validation.
# all(abs(unlist(lapply(arbFLSSSrst, function(x) sum(V[x]))) - target) <= error)
#
#
# # Use 4-sum accelerator. Massive speedup.
# system.time({
#   arbFLSSSrst = FLSSS::arbFLSSS(
#     len, Vstr, targetStr, solutionNeed = 2, maxCore = 1, ksumK = 4,
#     verbose = FALSE, approxNinstance = 1000, tlimit = 60)
# })
#
#
# # Validation.
# all(abs(unlist(lapply(arbFLSSSrst, function(x) sum(V[x]))) - target) <= error)




# # ===========================================================================
# # Compare arbFLSSS() and mFLSSSpar(). Example takes more than 2 seconds. The
# # section contains some analysis of the algorithms.
# # ===========================================================================
# set.seed(4)
# d = 5L # Set dimension.
# N = 60L # Set size.
# len = 10L # Subset size.
# roundN = 2L # For rounding the numeric values before conversion to strings.
#
#
# V = matrix(round(runif(N * d, -1e5, 1e5), roundN), nrow = N) # Make superset.
# sol = sample(N, len) # Make a solution.
# target = round(colSums(V[sol, ]), roundN) # Target subset sum.
# error = rep(2e-3, d) # Effectively demands the target sum to be exactly
# # matched since roundN = 2.
#
#
# system.time({
#   mFLSSSparRst = FLSSS::mFLSSSpar(
#     maxCore = 7, len = len, mV = V, mTarget = target, mME = error,
#     avgThreadLoad = 20, solutionNeed = 1, tlimit = 60)
# })
#
#
# # Validation.
# all(unlist(lapply(mFLSSSparRst, function(x)
#   abs(colSums(V[x, , drop = FALSE]) - target) <= error)))
#
#
# options(scipen = 999) # Ensure numeric => string conversion does not
```

```
# # produce strings like 2e-3.
# Vstr = matrix(as.character(V), nrow = N) # String version of V.
# targetStr = as.character(target)
#
#
# # Use no k-sum accelerator.
# system.time({
#   arbFLSSSrst = FLSSS::arbFLSSS(
#     len = len, V = Vstr, target = targetStr, givenKsumTable = NULL,
#     tlimit = 60, solutionNeed = 1, maxCore = 7, ksumK = 0, verbose = TRUE)
# })
#
#
# # Validation.
# all(unlist(lapply(arbFLSSSrst, function(x)
#   abs(colSums(V[x, , drop = FALSE]) - target) <= error)))
#
#
# # Use 5-sum accelerator. Massive speedup.
# system.time({
#   arbFLSSSrst = FLSSS::arbFLSSS(
#     len = len, V = Vstr, target = targetStr, givenKsumTable = NULL,
#     tlimit = 60, solutionNeed = 1, maxCore = 7, ksumK = 5, verbose = TRUE)
# })


options(optionSave)
```

`arbFLSSSobjRun` *Run an* `arbFLSSS` *instance*

Description

Run an `arbFLSSS` instance decomposed from `decomposeArbFLSSS()`.

Usage

```
arbFLSSSobjRun(
  X,
  solutionNeed = 1L,
  tlimit = 60,
  maxCore = 7L,
  ksumK = 0L,
  ksumTableSizeScaler = 30L,
  verbose = TRUE
  )
```

Arguments

X	An arbFLSSS object from decomposeArbFLSSS().
solutionNeed	See the same argument in arbFLSSS().
tlimit	See the same argument in arbFLSSS().
maxCore	See the same argument in arbFLSSS(). Mining subsets is single-threaded, but if X has no k-sum accelerator, users have the option of computing one on the fly, which is multithreaded. Will be ignored if X already has a k-sum accelerator.
ksumK	See the same argument in arbFLSSS(). Will be ignored if X already has a k-sum accelerator.
ksumTableSizeScaler	See the same argument in arbFLSSS(). Will be ignored if X already has a k-sum accelerator.
verbose	See the same argument in arbFLSSS(). Will be ignored if X already has a k-sum accelerator.

Details

The rationale follows mFLSSSobjRun(). The pair decomposeArbFLSSS() and arbFLSSSobjRun() makes up the distributed computing counterpart of arbFLSSS().

Value

Has the same return from arbFLSSS().

Examples

```
set.seed(42)
d = 5L # Set dimension.
N = 30L # Set size.
len = 10L # Subset size.
roundN = 2L # For rounding the numeric values before conversion to strings.


V = matrix(round(runif(N * d, -1e5, 1e5), roundN), nrow = N) # Make superset.
sol = sample(N, len) # Make a solution.
target = round(colSums(V[sol, ]), roundN) # Target subset sum.


optionSave = options()
options(scipen = 999) # Ensure numeric => string conversion does not
# produce strings like 2e-3.
Vstr = matrix(as.character(V), nrow = N) # String version of V.
targetStr = as.character(target)


system.time({
  theDecomposed = FLSSS::decomposeArbFLSSS(
    len = len, V = Vstr, target = targetStr, approxNinstance = 1000,
    maxCore = 2, ksumTable = NULL, ksumK = 4, verbose = TRUE)
```

```
})


# Run the objects sequentially.
rst = unlist(lapply(theDecomposed$arbFLSSSobjects, function(x)
{
  FLSSS::arbFLSSSobjRun(x, solutionNeed = 1e9, tlimit = 5, verbose = FALSE)
}), recursive = FALSE)
str(rst)


options(optionSave)
```

---

auxGAPbb *Multithreaded generalized assignment problem solver via branch and bound*

Description

Multithreaded exact solver for the generalized assignment problem via decomposition to binary knapsack problems (branch), and Lagrangian relaxation (bound).

Usage

```
auxGAPbb(
  cost,
  profitOrLoss,
  budget,
  maxCore = 7,
  tlimit = 60,
  ub = "MT",
  greedyBranching = TRUE,
  optim = "max",
  multhreadOn = "nodes",
  threadLoad = 32
)
```

Arguments

`cost`	A numeric matrix. Dimensionality = N(agents) x N(tasks).
`profitOrLoss`	A numeric matrix of the same dimensionality of `cost`. Profit for maximum GAP. Loss for minimum GAP.
`budget`	A numeric vector. Size = N(agents).
`maxCore`	Maximal threads to invoke. No greater than the number of logical CPUs on machine.
`tlimit`	Return the best exsisting solution in `tlimit` seconds.
`ub`	Upper bound function. "MT" or "HS". See `auxKnapsack01bb()`.

greedyBranching
: If TRUE, branch and bound in a greedy manner. See Details.

optim
: A string. optim = "max" ("min") solves the maximum (minimum) GAP.

multhreadOn
: A string. The default multhreadOn = "nodes" multithreads over branching nodes. Internally, a single-threaded miner runs at first and stops once there are no less than maxCore * threadLoad latent trees stored in stack. The miner realizes those branches and distribute them to threads. Threads work on the subproblems sequentially and update the optimum supervised by a light mutex lock.

  Other values of multhreadOn assign threads to knapsack problems at each branching node, which is a historical remain and should always be avoided. It has overwhelming overheads because knapsack problems at those nodes are typically trivial.

threadLoad
: An integer. Each thread is loaded with threadLoad sub-problems on average.

Details

A popular library of GAP instances can be found here: https://github.com/WhateverLiu/gapInstances.

This algorithm is based on a foundational paper by Ross and Soland (1975) and is carefully engineered towards speed. Implementation highlights include (i) decomposition for multithreading; (ii) a new branching method (greedyBranching) that pushes all candidate branching variables at each node into stack instead of pushing only those that have the highest desirabilities and would not affect the subsequent branching after being pushed; (iii) the return of current best solutions in time; (iv) the capability of taking real costs and profits. greedyBranching may considerably lower the number of nodes having the same series of knapsack problems to solve, thus accelerate the convergence speed.

Value

A list of 5:

totalProfitOrLoss
: Total profit or loss generated from the assignment.

agentCost
: A numeric vector of total costs for each agent.

assignment
: An integer vector. assignment[i] indexes the agent assigned to the ith task.

nodes
: The number of branching nodes generated in mining.

bkpSolved
: The number of binary knapsack problems solved in mining.

Note

The C++ implementation is fully independent and borrows no code from any commercial or open source.

Examples

```
# ===========================================================================
# Data source: http://people.brunel.ac.uk/~mastjjb/jeb/orlib/gapinfo.html,
# gap1 c515-1, 5 agents 15 tasks. Parsed instances from the library can be
# found here: https://github.com/WhateverLiu/gapInstances
```

```
# ===========================================================================
profit = c(17,21,22,18,24,15,20,18,19,18,16,22,24,24,16,23,16,21,16,17,16,19,
           25,18,21,17,15,25,17,24,16,20,16,25,24,16,17,19,19,18,20,16,17,21,
           24,19,19,22,22,20,16,19,17,21,19,25,23,25,25,25,18,19,15,15,21,25,
           16,16,23,15,22,17,19,22,24)
profit = t(matrix(profit, ncol = 5))
cost = c(8,15,14,23,8,16,8,25,9,17,25,15,10,8,24,15,7,23,22,11,11,12,10,17,16,
         7,16,10,18,22,21,20,6,22,24,10,24,9,21,14,11,14,11,19,16,20,11,8,14,
         9,5,6,19,19,7,6,6,13,9,18,8,13,13,13,10,20,25,16,16,17,10,10,5,12,23)
cost = t(matrix(cost, ncol = 5))
budget = c(36, 34, 38, 27, 33)


sol = FLSSS::auxGAPbb(cost, profit, budget, maxCore = 2, tlimit = 4,
                      ub = "MT", greedyBranching = TRUE, optim = "max")




# ===========================================================================
# Data source: http://support.sas.com/documentation/cdl/en/ormpug/65554/HTML
# /default/viewer.htm#ormpug_decomp_examples02.htm, an example made by SAS
# corporation. 24 tasks assigned to 8 agents.
# ===========================================================================
cost = t(matrix(c(
  8,18,22,5,11,11,22,11,17,22,11,20,13,13,7,22,15,22,24,8,8,24,18,8,24,14,11,
  15,24,8,10,15,19,25,6,13,10,25,19,24,13,12,5,18,10,24,8,5,22,22,21,22,13,
  16,21,5,25,13,12,9,24,6,22,24,11,21,11,14,12,10,20,6,13,8,19,12,19,18,10,21,
  5,9,11,9,22,8,12,13,9,25,19,24,22,6,19,14,25,16,13,5,11,8,7,8,25,20,24,20,11,
  6,10,10,6,22,10,10,13,21,5,19,19,19,5,11,22,24,18,11,6,13,24,24,22,6,22,5,14,
  6,16,11,6,8,18,10,24,10,9,10,6,15,7,13,20,8,7,9,24,9,21,9,11,19,10,5,23,20,5,
  21,6,9,9,5,12,10,16,15,19,18,20,18,16,21,11,12,22,16,21,25,7,14,16,10),
  ncol = 8))
profit = t(matrix(c(
  25,23,20,16,19,22,20,16,15,22,15,21,20,23,20,22,19,25,25,24,21,17,23,17,16,
  19,22,22,19,23,17,24,15,24,18,19,20,24,25,25,19,24,18,21,16,25,15,20,20,18,
  23,23,23,17,19,16,24,24,17,23,19,22,23,25,23,18,19,24,20,17,23,23,16,16,15,23,
  15,15,25,22,17,20,19,16,17,17,20,17,17,18,16,18,15,25,22,17,17,23,21,20,24,22,
  25,17,22,20,16,22,21,23,24,15,22,25,18,19,19,17,22,23,24,21,23,17,21,19,19,17,
  18,24,15,15,17,18,15,24,19,21,23,24,17,20,16,21,18,21,22,23,22,15,18,15,21,22,
  15,23,21,25,25,23,20,16,25,17,15,15,18,16,19,24,18,17,21,18,24,25,18,23,21,15,
  24,23,18,18,23,23,16,20,20,19,25,21), ncol = 8))
budget = c(36, 35, 38, 34, 32, 34, 31, 34)


# Intel CPU i7-4770 3.4GHz, g++ '-Ofast', 64-bit Windows 7:
system.time({sol = FLSSS::auxGAPbb(
  cost, profit, budget, maxCore = 2, tlimit = 4, ub = "MT",
  greedyBranching = FALSE, optim = "max")})
# user  system elapsed
# 0.02    0.00    0.01
# The elapsed time is about 1% of that reported by the SAS proc with 8
# threads, although its hardware configuration is unknown.
```

```
system.time({sol2 = FLSSS::auxGAPbb(
  cost, profit, budget, maxCore = 2, tlimit = 4, ub = "MT",
  greedyBranching = TRUE, optim = "max")})
sol[c("nodes", "bkpSolved")]  # 4526, 14671, can be different.
sol2[c("nodes", "bkpSolved")] # 4517, 13115, can be different.
# Greedy branching may lower the numbers of branching nodes and
# knapsack problems to solve.




# ===============================================================================
# Play random numbers.
# ===============================================================================
set.seed(22) # A nontrivial instance searched via changing random seeds.
             # RNG in R 3.5.1 for Windows.
Nagent = 20L; Ntask = 200L
cost = matrix(runif(Nagent * Ntask, 1e3, 1e6), nrow = Nagent)
profit = matrix(abs(rnorm(Nagent * Ntask, 1e6, 1e6)) + 1000, nrow = Nagent)
budget = apply(cost, 1, function(x) runif(1, min(x), sum(x) / 2))


# Intel CPU i7-4770 3.4GHz, g++ '-Ofast', 64-bit Windows 7.
system.time({sol1 = FLSSS::auxGAPbb(
  cost, profit, budget,
  maxCore = 1, multhreadOn = "KPs",
  tlimit = 3600, ub = "MT", greedyBranching = TRUE, optim = "max")})
# user  system elapsed
# 9.17    0.00    9.19


# Multithread knapsack problems at each branching node.
# This does not accelerate the speed at all because threading overheads
# are overwhelming.
system.time({sol2 = FLSSS::auxGAPbb(
  cost, profit, budget,
  maxCore = 7, multhreadOn = "KPs",
  tlimit = 3600, ub = "MT", greedyBranching = TRUE, optim = "max")})
# user   system elapsed
# 39.02    5.24   11.12


# Multithread nodes.
system.time({sol3 = FLSSS::auxGAPbb(
  cost, profit, budget,
  maxCore = 7, multhreadOn = "nodes", threadLoad = 32L,
  tlimit = 3600, ub = "MT", greedyBranching = TRUE, optim = "max")})
#  user  system elapsed
# 14.62    0.00    2.13
```

auxGAPbbDp *Multithreaded generalized assignment problem solver via a hybrid of branch-and-bound and dynamic programming.*

Description

Multithreaded exact solver for the generalized assignment problem via decomposition to binary knapsack problems (branch), and Lagrangian relaxation (bound). Knapsack problems are solved via dynamic programming.

Usage

```
auxGAPbbDp(
  cost,
  profitOrLoss,
  budget,
  maxCore = 7L,
  tlimit = 60,
  greedyBranching = TRUE,
  optim = "max",
  multhreadOn = "nodes",
  threadLoad = 32
)
```

Arguments

`cost`	An integer matrix. Dimensionality = N(agents) x N(tasks).
`profitOrLoss`	A numeric matrix of the same dimensionality of `cost`. Profit for maximum GAP. Loss for minimum GAP.
`budget`	An integer vector. Size = N(agents).
`maxCore`	Maximal threads to invoke. No greater than the number of logical CPUs on machine.
`tlimit`	Return the best exsisting solution in `tlimit` seconds.
`greedyBranching`	See `greedyBranching` in `auxGAPbb()`.
`optim`	A string. `optim = "max"` (`"min"`) solves the maximum (minimum) GAP.
`multhreadOn`	See `multhreadOn` in `auxGAPbb()`.
`threadLoad`	See `threadLoad` in `auxGAPbb()`.

Details

For instances with integral `cost` and `budget` of small magnitudes, knapsack problems from the decomposition could be solved faster via dynamic programming than branch and bound. See `auxKnapsack01dp()`. Implementation highlights include (i) only `maxCore` many lookup matrices exist in memory; (ii) a lookup matrix is recycled if it is sufficiently large to support solving the current knapsack problem, so as to minimize potential contentious memory allocations in multithreading. These management rules for economical memories propagate through all package functions. See more details in `auxGAPbb()`.

Value

See Value of `auxGAPbb()`.

Note

`cost` and `budget` are integers. The C++ implementation is fully independent and borrows no code from any commercial or open source.

Examples

```
# ===========================================================================
# Data source: http://support.sas.com/documentation/cdl/en/ormpug/65554/HTML
# /default/viewer.htm#ormpug_decomp_examples02.htm, an example made by SAS
# corporation. 24 tasks assigned to 8 agents.
# ===========================================================================
cost = t(matrix(as.integer(c(
  8,18,22,5,11,11,22,11,17,22,11,20,13,13,7,22,15,22,24,8,8,24,18,8,24,14,11,
  15,24,8,10,15,19,25,6,13,10,25,19,24,13,12,5,18,10,24,8,5,22,22,21,22,13,
  16,21,5,25,13,12,9,24,6,22,24,11,21,11,14,12,10,20,6,13,8,19,12,19,18,10,21,
  5,9,11,9,22,8,12,13,9,25,19,24,22,6,19,14,25,16,13,5,11,8,7,8,25,20,24,20,11,
  6,10,10,6,22,10,10,13,21,5,19,19,19,5,11,22,24,18,11,6,13,24,24,22,6,22,5,14,
  6,16,11,6,8,18,10,24,10,9,10,6,15,7,13,20,8,7,9,24,9,21,9,11,19,10,5,23,20,5,
  21,6,9,9,5,12,10,16,15,19,18,20,18,16,21,11,12,22,16,21,25,7,14,16,10)),
  ncol = 8))
profit = t(matrix(c(
  25,23,20,16,19,22,20,16,15,22,15,21,20,23,20,22,19,25,25,24,21,17,23,17,16,
  19,22,22,19,23,17,24,15,24,18,19,20,24,25,25,19,24,18,21,16,25,15,20,20,18,
  23,23,23,17,19,16,24,24,17,23,19,22,23,25,23,18,19,24,20,17,23,23,16,16,15,23,
  15,15,25,22,17,20,19,16,17,17,20,17,17,18,16,18,15,25,22,17,17,23,21,20,24,22,
  25,17,22,20,16,22,21,23,24,15,22,25,18,19,19,17,22,23,24,21,23,17,21,19,19,17,
  18,24,15,15,17,18,15,24,19,21,23,24,17,20,16,21,18,21,22,23,22,15,18,15,21,22,
  15,23,21,25,25,23,20,16,25,17,15,15,18,16,19,24,18,17,21,18,24,25,18,23,21,15,
  24,23,18,18,23,23,16,20,20,19,25,21), ncol = 8))
budget = as.integer(c(36, 35, 38, 34, 32, 34, 31, 34))


system.time({sol = FLSSS::auxGAPbbDp(
  cost, profit, budget,
  maxCore = 2, tlimit = 4, greedyBranching = TRUE, optim = "max")})
sol[c("nodes", "bkpSolved")] # 2630, 8102




set.seed(8) # A nontrivial instance searched via changing random seeds.
            # RNG in R 3.5.1 for Windows.
```

```
Nagent = 20L; Ntask = 200L
cost = matrix(as.integer(runif(Nagent * Ntask, 1, 50)), nrow = Nagent)
budget = as.integer(apply(cost, 1, function(x) runif(1, min(x), sum(x) / 2)))
profit = matrix(abs(rnorm(Nagent * Ntask, 1e6, 1e6)) + 1000, nrow = Nagent)


# Intel CPU i7-4770 3.4GHz, g++ '-Ofast', 64-bit Windows 7.
system.time({sol1 = FLSSS::auxGAPbb(
  cost, profit, budget,
  maxCore = 7, multhreadOn = "nodes",
  tlimit = 3600, greedyBranching = TRUE, optim = "max")})
# user  system elapsed
# 14.43    0.00    2.11


system.time({sol2 = FLSSS::auxGAPbbDp(
  cost, profit, budget,
  maxCore = 7, multhreadOn = "nodes",
  tlimit = 3600, greedyBranching = TRUE, optim = "max")})
# user  system elapsed
# 5.77    0.00    0.87
# Dynamic programming for solving knapsack problems could be faster
# for integral costs and budgets of small magnitudes.
```

`auxGAPga` *Multithreaded generalized assignment problem solver via genetic algorithm*

Description

A genetic algorithm with local heuristics for GAP.

Usage

```
auxGAPga(
  cost,
  profitOrLoss,
  budget,
  trials,
  populationSize,
  generations,
  randomSeed = NULL,
  maxCore = 7,
  optim = "max"
)
```

Arguments

- `cost` — A numeric matrix. Dimensionality = N(agents) x N(tasks).
- `profitOrLoss` — A numeric matrix of the same dimensionality of `cost`. Profit for maximum GAP. Loss for minimum GAP.
- `budget` — A numeric vector. Size = N(agents).
- `trials` — An integer. Number of trials, aka the number of population sets.
- `populationSize` — An integer. Size of each population.
- `generations` — An integer. As reproduction iterates, if there have been `generations` many children produced and accepted in population but no update on the current optimum, function quits.
- `randomSeed` — An integer or `NULL`. `randomSeed` seeds the random number generator in R, generates `trials` many integers to seed the `mt19937_64` (Mersenne Twister) engine for each trial.
- `maxCore` — Maximal threads to invoke. No greater than the number of logical CPUs on machine. The algorithm multithreads over trials.
- `optim` — A string. `optim` = `"max"` (`"min"`) solves the maximum (minimum) GAP.

Details

This algorithm is based on a foundational paper by Chu and Beasley (1997) and is carefully engineered towards speed. Besides the standard cross-over and mutation operations, the algorithm applies two local heuristics for educating the new borns. The first is to randomly pick a task from each overloaded agent and reassign the task to the next budget-sufficient agent — if there is any. The second is to raise the total profit by reassigning another agent for each task — if the reassignment would not result in overload. The algorithm outperforms most peer metaheuristics such as variants of simulated annealing and tabu search (Osman), and is highly effective for large and hard instances.

Value

A list of 4:

- `totalProfitOrLoss` — Total profit or loss generated from the assignment. Negative infinity if no solution found.
- `agentCost` — A numeric vector of total costs for each agent. Empty if no solution found.
- `assignment` — An integer vector. `assignment[i]` indexes the agent assigned to the `i`th task. Empty if no solution found.
- `populationInfo` — A list of 3:
- `allGenes` — An N(task) x (`populationSize` x `trials`) integer matrix recording genes in all population sets upon completion. Each column represents a gene, namely a tentative assignment.
- `allBudgetExceedance` — A numeric vector of the size of `populationSize` x `trials`. `allBudgetExceedance[i]` equals the total budget exceedance of tentative assignment `allGenes[, i]`.

allProfitOrLoss
: A numeric vector of the size of `allBudgetExceedance`. `allProfitOrLoss[i]` equals the total profit or loss of tentative assignment `allGenes[, i]`.

Note

The C++ implementation is fully independent and borrows no code from any commercial or open source.

Examples

```
# =============================================================================
# A trivial instance
# =============================================================================
profit = c(17,21,22,18,24,15,20,18,19,18,16,22,24,24,16,23,16,21,16,17,16,19,
           25,18,21,17,15,25,17,24,16,20,16,25,24,16,17,19,19,18,20,16,17,21,
           24,19,19,22,22,20,16,19,17,21,19,25,23,25,25,25,18,19,15,15,21,25,
           16,16,23,15,22,17,19,22,24)
profit = t(matrix(profit, ncol = 5))
cost = c(8,15,14,23,8,16,8,25,9,17,25,15,10,8,24,15,7,23,22,11,11,12,10,17,16,
         7,16,10,18,22,21,20,6,22,24,10,24,9,21,14,11,14,11,19,16,20,11,8,14,
         9,5,6,19,19,7,6,6,13,9,18,8,13,13,13,10,20,25,16,16,17,10,10,5,12,23)
cost = t(matrix(cost, ncol = 5))
budget = c(36, 34, 38, 27, 33)


Nagent = 5L; Ntask = 15L
rst = FLSSS::auxGAPga(
  cost, profit, budget, trials = 2, populationSize = 100, generations = 10000,
  randomSeed = 42, maxCore = 2, optim = "max")


# =============================================================================
# A relatively hard instance.
# =============================================================================
# Download gapInstances.Rdata from
# https://github.com/WhateverLiu/gapInstances. Load it in R.
if (FALSE)
{
  cost = gapC[[3]]$cost
  loss = gapC[[3]]$loss
  budget = gapC[[3]]$budget
  # Intel CPU i7-4770 3.4GHz, g++ '-Ofast', 64-bit Windows 7.
  system.time({rst = FLSSS::auxGAPga(
    cost, loss, budget, trials = 7, randomSeed = 42, populationSize = 100,
    generations = 500000, optim = "min", maxCore = 7)})
  rst$totalProfitOrLoss # 1416
  #  user  system elapsed
  # 69.24    0.17   11.61
  # The known optimum equals 1402 as the total loss.
}
```

auxKnapsack01bb *Multithreaded binary knapsack problem solver via branch and bound*

Description

Given items' weights and values, concurrently solve 0-1 knapsack problems to optimality via branch and bound for multiple knapsacks of different capacities.

Usage

```
auxKnapsack01bb(
  weight,
  value,
  caps,
  itemNcaps = integer(0),
  maxCore = 7L,
  tlimit = 60,
  ub = "MT",
  simplify = TRUE
)
```

Arguments

`weight`	A numeric vector.
`value`	A numeric vector. The size equals that of `weight`.
`caps`	A numeric vector of knapsack capacities.
`itemNcaps`	An integer vector of upper bounds on the number of selected items. `itemNcaps[i]` corresponds to instance `caps[i]`. Empty `itemNcaps` implies no size restriction.
`maxCore`	Maximal threads to invoke. No greater than the number of logical CPUs on machine.
`tlimit`	Return the best exsisting solution in `tlimit` seconds.
`ub`	Upper bound function.
`simplify`	If `length(caps) == 1`, simplify the output.

Details

The algorithm takes the Horowitz-Sahni (1974) and the Martello-Toth (1977) upper bound functions and is carefully engineered towards speed. Implementation highlights include (i) an extra option of upper bounding the number of selected items, which only adds trivial overhead; (ii) the return of existing best solutions in time; (iii) the capability of taking numeric weights and values.

Value

A list of 2:

`maxValue`: a numeric vector. `maxValue[i]` equals the sum of values of items selected for capacity `caps[i]`.

`selection`: a list of integer vectors. `selection[i]` indexes the items selected for capacity `caps[i]`.

Note

The function is not to solve the 0-1 multiple knapsack problem. The C++ implementation is fully independent and borrows no code from any open or commercial source.

Examples

```
set.seed(42)
weight = runif(100, min = 1e3, max = 1e6)
value = weight ^ 0.5 * 100 # Higher correlation between item weights and values
                           # typically implies a harder knapsack problem.
caps = runif(10, min(weight), sum(weight))
rst = FLSSS::auxKnapsack01bb(weight, value, caps, maxCore = 2, tlimit = 2)
str(rst)
```

`auxKnapsack01dp` *Multithreaded binary knapsack problem solver via dynamic programming*

Description

Given items' weights and values, concurrently solve 0-1 knapsack problems to optimality via dynamic programming for multiple knapsacks of different capacities.

Usage

```
auxKnapsack01dp(
  weight,
  value,
  caps,
  maxCore = 7L,
  tlimit = 60,
  simplify = TRUE
)
```

Arguments

`weight`	An integer vector.
`value`	A numeric vector. The size equals that of `weight`.
`caps`	An integer vector of knapsack capacities.

maxCore	Maximal threads to invoke. No greater than the number of logical CPUs on machine.
tlimit	Return the best exsisting solution in `tlimit` seconds.
simplify	If `length(caps) == 1`, simplify the output.

Details

Implementation highlights include (i) lookup matrix is only of space complexity O(N * [max(C) - min(W)]), where N = the number of items, max(C) = maximal knapsack capacity, min(W) = minimum item weight; (ii) threads read and write the same lookup matrix and thus accelerate each other; (iii) the return of existing best solutions in time.

Value

A list of 3:

`maxValue`: a numeric vector. `maxValue[i]` equals the sum of values of items selected for capacity `caps[i]`.

`selection`: a list of integer vectors. `selection[i]` indexes the items selected for capacity `caps[i]`.

`lookupTable`: a numeric matrix.

Note

The function is not to solve the 0-1 multiple knapsack problem. `weight` and `caps` are integers. Be cautioned that dynamic programming is not suitable for problems with weights or capacities of high magnitudes due to its space complexity. Otherwise it could outperform branch-and-bound especially for large instances with highly correlated item weights and values.

Examples

```
# Examples with CPU (user + system) or elapsed time > 5s
#                  user system elapsed
# auxKnapsack01dp 6.53      0    3.33
# CRAN complains about computing time. Wrap it.
if (FALSE)
{
  set.seed(42)
  weight = sample(10L : 100L, 600L, replace = TRUE) # Dynamic programming
  # solution requires integer
  # weights.
  value = weight ^ 0.5 * 100 # Higher correlation between item weights and values
  # typically implies a harder knapsack problem.
  caps = as.integer(runif(10, min(weight), 600L))
  system.time({rstDp = FLSSS::auxKnapsack01dp(
    weight, value, caps, maxCore = 2, tlimit = 4)})
  system.time({rstBb = FLSSS::auxKnapsack01bb(
    weight, value, caps, maxCore = 2, tlimit = 4)})
  # Dynamic programming can be faster than branch-and-bound for integer weights
  # and capacity of small magnitudes.
}
```

decomposeArbFLSSS *arbFLSSS decomposition*

Description

Decompose an arbFLSSS instance into sub-problems for distributed computing.

Usage

```
decomposeArbFLSSS(
  len,
  V,
  target,
  approxNinstance = 1000L,
  maxCore = 7L,
  ksumTable = NULL,
  ksumK = 4L,
  ksumTableSizeScaler = 30L,
  verbose = TRUE
  )
```

Arguments

len	See the same argument in arbFLSSS().
V	See the same argument in arbFLSSS().
target	See the same argument in arbFLSSS().
approxNinstance	See the same argument in arbFLSSS().
maxCore	See the same argument in arbFLSSS(). The decomposition is single-threaded, but building the k-sum accelerator is multithreaded.
ksumTable	Either NULL or the return value from ksumHash(). ksumTable is not necessary for the decomposition. The function merely store a reference to it in every arbFLSSS object.
ksumK	See the same argument in arbFLSSS(). If ksumK >= 3 and ksumTable == NULL, the function will build a k-sum accelerator and store a reference in every arbFLSSS object.
ksumTableSizeScaler	See the same argument in arbFLSSS().
verbose	See the same argument in arbFLSSS().

Details

The rationale follows decomposeMflsss(). The pair decomposeArbFLSSS() and arbFLSSSobjRun() makes up the distributed computing counterpart of arbFLSSS().

Value

A list of two:

`$arbFLSSSobjects`: a list. Each element is an `arbFLSSS` object that would be supplied to `arbFLSSSobjRun()`.

`$solutionsFound`: a list. Solutions found during decomposition.

Examples

```
set.seed(42)
d = 5L # Set dimension.
N = 60L # Set size.
len = 10L # Subset size.
roundN = 2L # For rounding the numeric values before conversion to strings.


V = matrix(round(runif(N * d, -1e5, 1e5), roundN), nrow = N) # Make superset.
sol = sample(N, len) # Make a solution.
target = round(colSums(V[sol, ]), roundN) # Target subset sum.


optionSave = options()
options(scipen = 999) # Ensure numeric => string conversion does not
# produce strings like 2e-3.
Vstr = matrix(as.character(V), nrow = N) # String version of V.
targetStr = as.character(target)


system.time({
  theDecomposed = FLSSS::decomposeArbFLSSS(
    len = len, V = Vstr, target = targetStr, approxNinstance = 1000,
    maxCore = 2, ksumTable = NULL, ksumK = 4, verbose = TRUE)
})


# Check if any solution has been found during decomposition.
str(theDecomposed$solutionsFound)


# Check the first arbFLSSS object.
str(theDecomposed$arbFLSSSobjects[[1]])


options(optionSave)
```

`decomposeMflsss` *mFLSSS decomposition*

Description

Decompose an `mFLSSS` instance into sub-problems for distributed computing.

Usage

```
decomposeMflsss(
  len,
  mV,
  mTarget,
  mME,
  solutionNeed = 1L,
  dl = ncol(mV),
  du = ncol(mV),
  useBiSrchInFB = FALSE,
  approxNinstance = 50000L
  )
```

Arguments

`len`	See the same argument in `mFLSSSpar()`.
`mV`	See the same argument in `mFLSSSpar()`.
`mTarget`	See the same argument in `mFLSSSpar()`.
`mME`	See the same argument in `mFLSSSpar()`.
`solutionNeed`	See the same argument in `mFLSSSpar()`.
`dl`	See the same argument in `mFLSSSpar()`.
`du`	See the same argument in `mFLSSSpar()`.
`useBiSrchInFB`	See the same argument in `mFLSSSpar()`.
`approxNinstance`	Approximately how many instances should the problem be decomposed into.

Details

This function and `mFLSSSobjRun()` constitute a multi-process counterpart of `mFLSSSpar()`. It decomposes a multidimensional subset sum problem into numerous independent instances that can be submitted to any computing resource of CPU threads, on each of which `mFLSSSobjRun()` receives and solves the instance.

For example, if 1000 threads are available, either on a computing cluster or on a few hundred laptops, one could (i) decompose the problem of interest into 100000 instances using `decomposeMflsss()`, (ii) transmit each instance to any available thread and calls `mFLSSSobjRun()` on the instance, (iii) collect the results from all threads. It is strongly recommended to decompose the initial problem into much more instances than the threads, provided that an automatic queueing system exists, so there would be less chance of having idling threads during computation — if the number of instances equals the number of threads, some threads may finish earlier than others due to the heterogeneous nature of the instances, thus the computing waste.

The pair `decomposeMflsss()` and `mFLSSSobjRun()` is designed for exploiting distributed resource to solve large and hard multidimensional subset sum instances.

Value

A list of two:

`$mflsssObjects`: a list. Each element is an `mFLSSS` object that would be supplied to `mFLSSSobjRun()`.

`$solutionsFound`: a list. Solutions found during decomposition.

Examples

```
N = 30L # Superset size.
len = 6L # Subset size.
dimen = 5L # Dimension.
set.seed(8120)
v = matrix(runif(N * dimen) * 1000, nrow = N) # Superset.
sol = sample(N, len)
target = colSums(v[sol, ]) # Target sum.
ME = target * 0.03 # Error threshold.
approxNinstance = 1000


validate = function(len, v, target, ME, result)
{
  all(unlist(lapply(result, function(x)
    all(abs(colSums(v[x, ]) - target) <= ME))))
}


decompedFlsss = FLSSS::decomposeMflsss(
  len = len, mV = v, mTarget = target, mME = ME, solutionNeed = 1e6,
  approxNinstance = approxNinstance)


str(decompedFlsss$solutionsFound) # See if the agent already found
# some solutions and validate them.
if(length(decompedFlsss$solutionsFound) > 0)
  print(validate(len, v, target, ME, decompedFlsss$solutionsFound))


length(decompedFlsss$mflsssObjects) # Number of independent small jobs.
someOtherSolutions = FLSSS::mFLSSSobjRun(
  decompedFlsss$mflsssObjects[[620]], tlimit = 3, solutionNeed = 1e6)


if(length(someOtherSolutions) > 0) # Validate solutions.
{
  print(someOtherSolutions)
  print(validate(len, v, target, ME, someOtherSolutions))
}
```

FLSSS *One-dimensional Subset Sum given error threshold*

Description

Given subset size `len`, sorted superset `v`, subset sum `target` and error `ME`, find at least `solutionNeed` index (integer) vector(s) `x`, such that `target - ME <= sum(v[x]) <= target + ME`. To mine subsets that sum in a given range, set `target` to the midpoint and `ME` to half of the range width.

Usage

```
FLSSS(
  len,
  v,
  target,
  ME,
  solutionNeed = 1L,
  LB = 1L : len,
  UB = (length(v) - len + 1L) : length(v),
  viaConjugate = FALSE,
  tlimit = 60,
  useBiSrchInFB = FALSE,
  NfractionDigits = Inf
  )
```

Arguments

`len`	An integer as the subset size: `0 <= len < length(v)`. If `len == 0`, `FLSSS()` mines subets without size restriction. `len <- 0` would be most likely slower than looping `len` over `1 : (length(v) - 1)`. See Details.
`v`	A sorted numeric vector, the superset. `v` can be negative and nonunique.
`target`	A numeric value, the subset sum target.
`ME`	A positive numeric value, the error threshold.
`solutionNeed`	An integer, the least number of solutions wanted. If the function returns fewer solutions, either `tlimit` is up or less than `solutionNeed` solutions exist. The function may also return more than `solutionNeed` solutions.
`LB`	An integer vector of size `len` as the lower bounds of the solution space: for any solution `x`, `LB[i] <= x[i]`. Custom `LB` should be no less than `1L : len` element-wisely. Every element in `v` should be within the range enclosed by `LB` and `UB`.
`UB`	An integer vector of size `len` as the upper bounds of the solution space: for any solution `x`, `x[i] <= UB[i]`. Custom `UB` should be no greater than `(length(v) - len + 1L ) : length(v)` element-wisely. Every element in `v` should be within the range enclosed by `LB` and `UB`.

`viaConjugate` A boolean value. If `TRUE`, `FLSSS()` mines susbets of size `length(v) - len` that sum to `sum(v) - target` with the same `ME`. Let `x` be the integer vector indexing a qualified subset. `FLSSS()` returns `(1L : length(v))[-x]`. Simulations show that `FLSSS()` often finds the first qualified conjugate subset faster if `len` is much less than `length(v) / 2`.

`tlimit` A numeric value. Enforce function to return in `tlimit` seconds.

`useBiSrchInFB` A boolean value. If `TRUE`, the function performs binary search for index bounds in the auxiliary triangle matrix of continuous sequence sums. This argument is mainly for research. Simulations show binary search has no major advantage over linear search due to caching mechanisms. The advantage may be pronounced if `length(v)` is substantial ($>$ 10000) while `len` is small ($<$ 5).

`NfractionDigits`
An integer, the maximum number of fractional digits of all elements in `v`. Internally, `v`, `target` and `ME` are multiplied by `10 ^ NfractionDigits`, and then converted as integer values before mining. The default `Inf` prevents such conversion. The goal is eliminate

Details

If `len == 0`, `FLSSS()` would (1) reset `len` to `length(v)`, (2) pad `len` zeros at the beginning of `v` and sort `v`, (3) search for size-`len` subsets, and (4) for an index vector that represents a subset, erases elements pointing to zeros in `v`. See the package documentation for more details.

Value

A list of index vectors.

Examples

```
# ===============================================================================
# Example I: play random numbers.
# ===============================================================================
# rm(list = ls()); gc()
subsetSize = 200L
supersetSize = 1000L
superset = 10000 * sort(rnorm(supersetSize) ^ 3 + 2 * runif(supersetSize) ^ 2 +
           3 * rgamma(supersetSize, 5, 1) + 4)
subsetSum = runif(1, sum(superset[1L : subsetSize]), sum(superset[(supersetSize -
            subsetSize + 1L) : supersetSize]))
subsetSumError = 1e-3


# Mine 3 subsets
rst1 = FLSSS::FLSSS(len = subsetSize, v = superset, target = subsetSum,
                    ME = subsetSumError, solutionNeed = 3, tlimit = 4)


# Mine 3 subsets via solving the conjugate problem
rst2 = FLSSS::FLSSS(len = subsetSize, v = superset, target = subsetSum,
                    ME = subsetSumError, solutionNeed = 3, tlimit = 4,
```

```
                    viaConjugate = TRUE)


# Verify uniqueness
cat("rst1 number of solutions =",
    length(unique(lapply(rst1, function(x) sort(x)))), "\n")
cat("rst2 number of solutions =",
    length(unique(lapply(rst2, function(x) sort(x)))), "\n")


# Verify solutions
if(length(rst1) > 0)
  all(unlist(lapply(rst1, function(x)
    abs(sum(superset[x]) - subsetSum) <= subsetSumError)))
if(length(rst2) > 0)
  all(unlist(lapply(rst2, function(x)
    abs(sum(superset[x]) - subsetSum) <= subsetSumError)))


# Mine 3 subsets in bounded solution space.
# Make up the lower and upper bounds for the solution space:
tmp = sort(sample(1L : supersetSize, subsetSize))
tmp2 = sort(sample(1L : supersetSize, subsetSize))
lowerBounds = pmin(tmp, tmp2)
upperBounds = pmax(tmp, tmp2)
rm(tmp, tmp2)


# 'FLSSS()' does not work if there are elements not under the hood of
# lowerBounds + upperBounds. Exclude those elements:
remainIndex = unique(unlist(apply(cbind(lowerBounds, upperBounds), 1,
  function(x) x[1] : x[2])))
lowerBounds = match(lowerBounds, remainIndex)
upperBounds = match(upperBounds, remainIndex)
superset = superset[remainIndex]


# Plant a subset sum:
solution = integer(subsetSize)
solution[1] = sample(lowerBounds[1] : upperBounds[1], 1)
for(i in 2L : subsetSize)
{
  l = max(lowerBounds[i], solution[i - 1] + 1L)
  u = upperBounds[i]
  if(l == u) solution[i] = u
  else solution[i] = sample(l : u, 1)
}
subsetSum = sum(superset[solution])
subsetSumError = abs(subsetSum) * 0.01 # relative error within 1%
rm(solution)


rst3 = FLSSS::FLSSS(len = subsetSize, v = superset, target = subsetSum,
```

```
                    ME = subsetSumError, solutionNeed = 2, tlimit = 4,
                    LB = lowerBounds, UB = upperBounds, viaConjugate = TRUE)


print(length(rst3))


# Verify solutions
if(length(rst3) > 0)
  cat(all(unlist(lapply(rst3, function(x)
    abs(sum(superset[x]) - subsetSum) <= subsetSumError))), "\n")




# ===========================================================================================
# Example II: mine a real-world dataset.
# ===========================================================================================
# rm(list = ls()); gc()
superset = c(
  -1119924501, -793412295, -496234747,  -213654767,   16818148,   26267601,   26557292,
     27340260,   28343800,   32036573,    32847411,   34570996,   34574989,   43633028,
     44003100,   47724096,   51905122,    52691025,   53600924,   56874435,   58207678,
     60225777,   60639161,   60888288,    60890325,   61742932,   63780621,   63786876,
     65167464,   66224357,   67198760,    69366452,   71163068,   72338751,   72960793,
     73197629,   76148392,   77779087,    78308432,   81196763,   82741805,   85315243,
     86446883,   87820032,   89819002,    90604146,   93761290,   97920291,   98315039,
    310120088, -441403864, -548143111,  -645883459, -149110919,  305170449, -248934805,
  -1108320430, -527806318, -192539936, -1005074405, -101557770, -156782742, -285384687,
   -418917176,   80346546, -273215446,  -552291568,   86824498,  -95392618, -707778486)
superset = sort(superset)
subsetSum = 139254953
subsetSumError = 0.1


# Find a subset of size 10.
subsetSize = 10L
rst = FLSSS::FLSSS(len = subsetSize, v = superset, target = subsetSum,
                   ME = subsetSumError, solutionNeed = 1, tlimit = 4)
# Verify:
all(unlist(lapply(rst, function(x)
  abs(sum(superset[x]) - subsetSum) <= subsetSumError)))


# Find a subset without size specification.
rst = FLSSS::FLSSS(len = 0, v = superset, target = subsetSum,
                   ME = subsetSumError, solutionNeed = 1, tlimit = 4)
# Verify:
all(unlist(lapply(rst, function(x)
  abs(sum(superset[x]) - subsetSum) <= subsetSumError)))


# Find a subset via looping subset size over 2L : (length(v)).
```

```
for(len in 2L : length(superset))
{
  rst = FLSSS::FLSSS(len = subsetSize, v = superset, target = subsetSum,
                     ME = subsetSumError, solutionNeed = 1, tlimit = 4)
  if(length(rst) > 0) break
}
# Verify:
all(unlist(lapply(rst, function(x)
  abs(sum(superset[x]) - subsetSum) <= subsetSumError)))


# Find as many qualified susbets as possible in 2 seconds
rst = FLSSS::FLSSS(len = subsetSize, v = superset, target = subsetSum,
                   ME = subsetSumError, solutionNeed = 999999L, tlimit = 2)
cat("Number of solutions =", length(rst), "\n")


# Verify:
all(unlist(lapply(rst, function(x)
  abs(sum(superset[x]) - subsetSum) <= subsetSumError)))




# ===============================================================================
# Example III: solve a special knapsack problem.
# Given the knapsack's capacity, the number of catagories, the number of items in each
# catagory, select the least number of items to fulfill at least 95% of the knapsack's
# capacity.
# ===============================================================================
# rm(list = ls()); gc()
capacity = 361
catagories = LETTERS[1L : 10L] # A, B, ..., J, 10 catagories
catagoryMasses = round(runif(length(catagories)) * 20 + 1)
catagoryItems = sample(1L : 20L, length(catagories))


itemLabel = unlist(mapply(function(x, i) rep(i, x), catagoryItems, catagories))
itemMasses = unlist(mapply(function(x, i) rep(x, i), catagoryMasses, catagoryItems))
vorder = order(itemMasses)
itemLabel = itemLabel[vorder]


superset = itemMasses[vorder]
rate = 0.95
subsetSum = (capacity * rate + capacity) / 2
subsetSumError = capacity - subsetSum
for(subsetSize in 1L : length(itemMasses))
{
  rst = FLSSS::FLSSS(len = subsetSize, v = superset, target = subsetSum,
                     ME = subsetSumError, solutionNeed = 1, tlimit = 4)
  if(length(rst) > 0) break
}
```

```
# There may exist no qualified subsets. One can lower 'rate' until a solution
# shows up.
if(length(rst) == 0L)
{
  cat("No solutions. Please lower rate and rerun.\n")
} else
{
  cat("A solution:\n")
  print(table(itemLabel[rst[[1]]]))
}


# rm(list = ls()); gc()
```

`FLSSSmultiset` *Multi-Subset Sum given error threshold*

Description

Find a subet of a given size for each of multiple supersets such that all the subsets sum in a given range.

Usage

```
FLSSSmultiset(
  len,
  buckets,
  target,
  ME,
  solutionNeed = 1L,
  tlimit = 60,
  useBiSrchInFB = FALSE,
  NfractionDigits = Inf
  )
```

Arguments

`len`	A positive integer vector as the subset sizes for the supersets.
`buckets`	A list of the supersets. `buckets[[i]]` is an unsorted numeric vector of size `len[i]`.
`target`	See `target` in `FLSSS()`.
`ME`	See `ME` in `FLSSS()`.
`solutionNeed`	See `solutionNeed` in `FLSSS()`.
`tlimit`	See `tlimit` in `FLSSS()`.
`useBiSrchInFB`	See `useBiSrchInFB` in `FLSSS()`.

`NfractionDigits`
: An integer, the maximum number of fractional digits of all elements in `v`. Internally, `v`, `target` and `ME` are multiplied by `10 ^ NfractionDigits`, and then converted as integer values before mining. The default `Inf` prevents such conversion.

Value

A list of solutions. Each solution is a list of index vectors. Assume `X` is a solution. `X[[i]]` indexes the subset of superset `buckets[[i]]`.

Examples

```
# # rm(list = ls()); gc()
Nsupersets = 30L
supersetSizes = sample(5L : 20L, Nsupersets, replace = TRUE)
subsetSizes = sapply(supersetSizes, function(x) sample(1L : x, 1))


# Create supersets at random:
supersets = lapply(supersetSizes, function(n)
{
  1000 * (rnorm(n) ^ 3 + 2 * runif(n) ^ 2 + 3 * rgamma(n, 5, 1) + 4)
})
str(supersets) # see the structure


# Give a subset sum
solution = mapply(function(n, l) sample(1L : n, l), supersetSizes, subsetSizes)
str(solution) # See structure
subsetsSum = sum(mapply(function(x, s) sum(x[s]), supersets, solution, SIMPLIFY = TRUE))
subsetsSumError = abs(subsetsSum) * 1e-7 # relative error within 0.00001%
rm(solution)


# Mine subsets:
rst = FLSSS::FLSSSmultiset(len = subsetSizes, buckets = supersets, target = subsetsSum,
                           ME = subsetsSumError, solutionNeed = 3, tlimit = 4)
cat("Number of solutions =", length(rst), "\n")


# Verify:
ver = all(unlist(lapply(rst, function(sol)
{
  S = sum(unlist(mapply(function(x, y) sum(x[y]), supersets, sol)))
  abs(S - subsetsSum) <= subsetsSumError
})))
cat("All subsets are qualified:", ver)
```

GAP *Generalized Assignment Problem solver*

Description

Given a number of agents and a number of tasks. An agent can finish a task with certain cost and profit. An agent also has a budget. Assign tasks to agents such that each agent costs no more than its budget while the total profit is maximized.

Usage

```
GAP(
  maxCore = 7L,
  agentsCosts,
  agentsProfits,
  agentsBudgets,
  heuristic = FALSE,
  tlimit = 60,
  threadLoad = 8L,
  verbose = TRUE
  )
```

Arguments

`maxCore`	Maximal threads to invoke. Ideally `maxCore` should not surpass the total logical processors on machine.
`agentsCosts`	A numeric matrix. `agentsCosts[i, j]` is the cost for agent `i` to finish task `j`.
`agentsProfits`	A numeric matrix. `agentsProfits[i, j]` is the profit from agent `i` finishing task `j`.
`agentsBudgets`	A numeric vector. `agentsBudgets[i]` is agent `i`'s budget.
`heuristic`	A boolean value. If `TRUE`, the function returns once it has found a solution whose sum of ranks of the profits becomes no less than that of the optimal. See `heuristic` in `mmKnapsack()`.
`tlimit`	A numeric value. Enforce function to return in `tlimit` seconds.
`threadLoad`	See `avgThreadLoad` in `mFLSSSpar()`.
`verbose`	If `TRUE`, function prints progress.

Value

A list of size nine.

`assignedAgents`	is a 2-column data frame, the mining result. The 1st column is task indexes. The 2nd column is agent indexes.
`assignmentProfit`	is the profit resulted from such assignment.

`assignmentCosts`
: is a numeric vector. `Value$assignmentCosts[i]` is the cost of agent `i`.

`agentsBudgets`
: is a numeric vector. `Value$agentsBudgets[i]` shows the budget of agent `i`.

`unconstrainedMaxProfit`
: is the would-be maximal profit if agents had infinite budgets.

`FLSSSsolution`
: is the solution from mining the corresponding multidimensional Subset Sum problem.

`FLSSSvec`
: is the multidimensional vector (a matrix) going into the multidimensional Subset Sum miner.

`MAXmat`
: is the subset sum targets' upper bounds going into the multidimensional Subset Sum miner.

`foreShadowFLSSSvec`
: is the multidimensional vector before comonotonization.

Examples

```
# ===========================================================================================
# Play random numbers
# ===========================================================================================
# rm(list = ls()); gc()
agents = 5L
tasks = 12L
costs = t(as.data.frame(lapply(1L : agents, function(x) runif(tasks) * 1000)))
budgets = apply(costs, 1, function(x) runif(1, min(x), sum(x)))
profits = t(as.data.frame(lapply(1L : agents, function(x)
  abs(rnorm(tasks) + runif(1, 0, 4)) * 10000)))


# A dirty function for examining the result's integrity. The function takes in
# the task-agent assignment, the profit or cost matrix M, and calculates the cost
# or profit generated by each agent. 'assignment' is a 2-column data
# frame, first column task, second column agent.
agentCostsOrProfits <- function(assignment, M)
{
  n = ncol(M) * nrow(M)
  M2 = matrix(numeric(n), ncol = tasks)
  for(i in 1L : nrow(assignment))
  {
    x = as.integer(assignment[i, ])
    M2[x[2], x[1]] = M[x[2], x[1]]
  }
  apply(M2, 1, function(x) sum(x))
}


dimnames(costs) = NULL
dimnames(profits) = NULL
names(budgets) = NULL
```

```
rst = FLSSS::GAP(maxCore = 7L, agentsCosts = costs, agentsProfits = profits,
                 agentsBudgets = budgets, heuristic = FALSE, tlimit = 60,
                 threadLoad = 8L, verbose = TRUE)
# Function also saves the assignment costs and profits
rst$assignedAgents
rst$assignmentProfit
rst$assignmentCosts


# Examine rst$assignmentCosts
if(sum(rst$assignedAgents) > 0) # all zeros mean the function has not found a solution.
  agentCostsOrProfits(rst$assignedAgents, costs)
# Should equal rst$assignmentCosts and not surpass budgets


# Examine rst$assignmentProfits
if(sum(rst$assignedAgents) > 0)
  sum(agentCostsOrProfits(rst$assignedAgents, profits))
# Should equal rst$assignmentProfit





# =============================================================================
# Test case P03 from
# https://people.sc.fsu.edu/~jburkardt/datasets/generalized_assignment/
# =============================================================================
agents = 3L
tasks = 8L
profits = t(matrix(c(
27, 12, 12, 16, 24, 31, 41, 13,
14,  5, 37,  9, 36, 25,  1, 34,
34, 34, 20,  9, 19, 19,  3, 34), ncol = agents))
costs = t(matrix(c(
21, 13,  9,  5,  7, 15,  5, 24,
20,  8, 18, 25,  6,  6,  9,  6,
16, 16, 18, 24, 11, 11, 16, 18), ncol = agents))
budgets = c(26, 25, 34)


rst = FLSSS::GAP(maxCore = 2L, agentsCosts = costs, agentsProfits = profits,
                 agentsBudgets = budgets, heuristic = FALSE, tlimit = 2,
                 threadLoad = 8L, verbose = TRUE)
agentCostsOrProfits(rst$assignedAgents, costs)
# Should equal rst$assignmentCosts and not surpass budgets


knownOptSolution = as.integer(c(3, 3, 1, 1, 2, 2, 1, 2))
knownOptSolution = data.frame(task = 1L : tasks, agent = knownOptSolution)


# Total profit from knownOptSolution:
```

```
sum(agentCostsOrProfits(knownOptSolution, profits))
# Total profit from FLSSS::GAP():
rst$assignmentProfit
```

`ksumHash` *Build k-sum accelerator*

Description

Compute k-sum lookup tables given a set.

Usage

```
ksumHash(
  ksumK,
  V,
  ksumTableSizeScaler = 30L,
  target = NULL,
  len = 0L,
  approxNinstance = 1000L,
  verbose = TRUE,
  maxCore = 7L
  )
```

Arguments

`ksumK`	See the same argument in `arbFLSSS()`.
`V`	See the same argument in `arbFLSSS()`.
`ksumTableSizeScaler`	See the same argument in `arbFLSSS()`.
`target`	See the same argument in `arbFLSSS()`. If `target != NULL`, the function will (i) decompose the `arbFLSSS` instance of (`len`, `target`, `V`) into about `approxNinstance` subproblems, (ii) from these subproblems infer the lower and upper index bounds for the k-subsets, and then (iii) compute & hash k-sums to build the accelerator. If `target = NULL`, no bounds will be imposed on the k-subsets and the accelerator built can be used for any subset sum instance.
`len`	See the same argument in `arbFLSSS()`. Will be ignored if `target == NULL`.
`approxNinstance`	See the same argument in `arbFLSSS()`.
`verbose`	See the same argument in `arbFLSSS()`.
`maxCore`	See the same argument in `arbFLSSS()`.

Details

k-sums are hashed using Yann Collet's xxHash that is the fastest among all non-cryptographic hash algorithms by 202204. See the benchmark <https://github.com/Cyan4973/xxHash>.

Value

Either an empty list (happens when, e.g. `ksumK < 3`), or a list of lists. The first list would be the 3-sum lookup table, and the last would be the `ksumK`-sum lookup table.

Examples

```
set.seed(42)
d = 5L # Set dimension.
N = 30L # Set size.
len = 10L # Subset size.
roundN = 2L # For rounding the numeric values before conversion to strings.


V = matrix(round(runif(N * d, -1e5, 1e5), roundN), nrow = N) # Make superset.
sol = sample(N, len) # Make a solution.
target = round(colSums(V[sol, ]), roundN) # Target subset sum.


optionSave = options()
options(scipen = 999) # Ensure numeric => string conversion does not
# produce strings like 2e-3.
Vstr = matrix(as.character(V), nrow = N) # String version of V.
targetStr = as.character(target)


system.time({
  theDecomposed = FLSSS::decomposeArbFLSSS(
    len = len, V = Vstr, target = targetStr, approxNinstance = 1000,
    maxCore = 2, ksumTable = NULL, ksumK = 4, verbose = TRUE)
})


# Run the objects sequentially.
rst = unlist(lapply(theDecomposed$arbFLSSSobjects, function(x)
{
  FLSSS::arbFLSSSobjRun(x, solutionNeed = 1e9, tlimit = 5, verbose = FALSE)
}), recursive = FALSE)
str(rst)


options(optionSave)
```

`mFLSSSobjRun` *Run an* `mFLSSS` *instance*

Description

Run a multidimensional subset sum instance decomposed from `decomposeMflsss()`.

Usage

```
mFLSSSobjRun(
  mflsssObj,
  solutionNeed = 1,
  tlimit = 60
  )
```

Arguments

mflsssObj	An mFLSSS object.
solutionNeed	See the same argument in mFLSSSpar().
tlimit	See the same argument in mFLSSSpar().

Details

See the details about decomposeMflsss().

Value

See the value of mFLSSSpar().

Examples

```
# See the example for decomposeMflsss().
```

---

mFLSSSpar *Multithreaded multidimensional Subset Sum given error thresholds*

Description

The multidimensional version of FLSSS(). See decomposeMflsss() for the multi-process version.

Usage

```
mFLSSSpar(
  maxCore = 7L,
  len,
  mV,
  mTarget,
  mME,
  solutionNeed = 1L,
  tlimit = 60,
  dl = ncol(mV),
  du = ncol(mV),
  useBiSrchInFB = FALSE,
  avgThreadLoad = 8L
  )
```

Arguments

- `maxCore` Maximal threads to invoke. Ideally `maxCore` should not surpass the total logical processors on machine.
- `len` An integer as the subset size. See `len` in `FLSSS()`.
- `mV` A data frame or a matrix as the multidimensional set, columns as dimensions.
- `mTarget` A numeric vector of size `ncol(mV)` as the subset sum.
- `mME` A numeric vector of size `ncol(mV)` as the subset sum error thresholds.
- `solutionNeed` See `solutionNeed` in `FLSSS()`.
- `tlimit` See `tlimit` in `FLSSS()`.
- `dl` An integer no greater than `ncol(mV)`. Let `sol` be the index vector of a solution. Let `dls <- 1L : dl`. The following is true:
  `colSums(mV[sol, dls]) >= mTarget[dls] - mME[dls]`.
- `du` An integer no greater than `ncol(mV)`. Let `sol` be the index vector of a solution. Let `dus <- (ncol(mV) - du + 1) : ncol(mV)`. The following is true:
  `colSums(mV[sol, dus]) <= mTarget[dus] + mME[dus]`.
- `useBiSrchInFB` See `useBiSrchInFB` in `FLSSS()`.
- `avgThreadLoad` If `mV` is comonotonic, `mFLSSSpar()` warms up with a breadth-first search and then spawns at least `B` branches for parallelization. `B` equals the first power-of-two integer no less than `avgThreadLoad * maxCore`.

Value

A list of index vectors.

Examples

```
# rm(list = ls()); gc()
subsetSize = 7L
supersetSize = 60L
dimension = 5L # dimensionality


# Create a supertset at random:
N = supersetSize * dimension
superset = matrix(1000 * (rnorm(N) ^ 3 + 2 * runif(N) ^ 2 +
  3 * rgamma(N, 5, 1) + 4), ncol = dimension)
rm(N)


# Plant a subset sum:
solution = sample(1L : supersetSize, subsetSize)
subsetSum = colSums(superset[solution, ])
subsetSumError = abs(subsetSum) * 0.01 # relative error within 1%
rm(solution)


# Mine subsets, dimensions fully bounded
```

```
rst = FLSSS::mFLSSSpar(maxCore = 2, len = subsetSize, mV = superset,
                       mTarget = subsetSum, mME = subsetSumError,
                       solutionNeed = 2, dl = ncol(superset), du = ncol(superset),
                       tlimit = 2, useBiSrchInFB = FALSE, avgThreadLoad = 8L)


# Verify:
cat("Number of solutions = ", length(rst), "\n")
if(length(rst) > 0)
{
  cat("Solutions unique: ")
  cat(length(unique(lapply(rst, function(x) sort(x)))) == length(rst), "\n")
  cat("Solutions correct: ")
  cat(all(unlist(lapply(rst, function(x)
    abs(colSums(superset[x, ]) - subsetSum) <= subsetSumError))), "\n")
} else
{
  cat("No solutions exist or timer ended too soon.\n")
}




# Mine subsets, the first 3 dimensions lower bounded,
# the last 4 dimension upper bounded
rst = FLSSS::mFLSSSpar(maxCore = 2, len = subsetSize, mV = superset,
                       mTarget = subsetSum, mME = subsetSumError,
                       solutionNeed = 2, dl = 3L, du = 4L,
                       tlimit = 2, useBiSrchInFB = FALSE, avgThreadLoad = 8L)


# Verify:
cat("Number of solutions = ", length(rst), "\n")
if(length(rst) > 0)
{
  cat("Solutions unique: ")
  cat(length(unique(lapply(rst, function(x) sort(x)))) == length(rst), "\n")
  cat("Solutions correct: ")
  cat(all(unlist(lapply(rst, function(x)
  {
    lowerBoundedDim = 1L : 3L
    lowerBounded = all(colSums(superset[x, lowerBoundedDim]) >=
      subsetSum[lowerBoundedDim] - subsetSumError[lowerBoundedDim])


    upperBoundedDim = (ncol(superset) - 3L) : ncol(superset)
    upperBounded = all(colSums(superset[x, upperBoundedDim]) <=
      subsetSum[upperBoundedDim] + subsetSumError[upperBoundedDim])


    lowerBounded & upperBounded
  }))), "\n")
} else
```

```
{
  cat("No solutions exist or timer ended too soon.\n")
}
```

`mFLSSSparImposeBounds` *Multithreaded multidimensional Subset Sum in bounded solution space given error thresholds*

Description

For comparison, function `mFLSSSpar()` puts no bounds on the solution space so it sorts `mV` internally in a special order for mining accerlation.

Usage

```
mFLSSSparImposeBounds(
  maxCore = 7L,
  len,
  mV,
  mTarget,
  mME,
  LB = 1L : len,
  UB = (nrow(mV) - len + 1L) : nrow(mV),
  solutionNeed = 1L,
  tlimit = 60,
  dl = ncol(mV),
  du = ncol(mV),
  targetsOrder = NULL,
  useBiSrchInFB = FALSE,
  avgThreadLoad = 8L
  )
```

Arguments

`maxCore`	See `maxCore` in `mFLSSSpar()`.
`len`	See `len` in `mFLSSSpar()`.
`mV`	See `mV` in `mFLSSSpar()`.
`mTarget`	See `mTarget` in `mFLSSSpar()`.
`mME`	See `mME` in `mFLSSSpar()`.
`LB`	See `LB` in `FLSSS()`.
`UB`	See `UB` in `FLSSS()`.
`solutionNeed`	See `solutionNeed` in `mFLSSSpar()`.
`tlimit`	See `tlimit` in `mFLSSSpar()`.
`dl`	See `dl` in `mFLSSSpar()`.

du
: See du in `mFLSSSpar()`.

targetsOrder
: This argument is mainly for research and unrecommended for use. Depending on the structure of `mV`, `mFLSSSpar()` or `mFLSSSparImposeBounds()` would break the mining task into a collection of no more than `len * (nrow(mV) - len) + 1` independent subtasks. Threads work on these subtasks, sequentially coordinated by an atomic counter [1]. Different subtasks have different probabilities of yielding a qualified subset, thus the order of subtasks matters to the mining speed. `targetsOrder` is an index vector of size `len * (nrow(mV) - len) + 1` for ordering the subtasks. `targetsOrder <- NULL` makes a special order, and is implicitly the choice in `mFLSSSpar()`. This order is empirically optimal based on simulations. See the package documentation for details.

useBiSrchInFB
: See `useBiSrchInFB` in `mFLSSSpar()`.

avgThreadLoad
: See `avgThreadLoad` in `mFLSSSpar()`.

Value

A list of index vectors.

References

[1] Atomic template class in Intel TBB. An atomic counter is used to coordinate heterogeneous subtasks to avoid idle threads. The atomic operation overhead is negligible compared to the time cost of the lightest subtask.

Examples

```
# rm(list = ls()); gc()
subsetSize = 7L
supersetSize = 60L
dimension = 5L # dimensionality


# Create a supertset at random:
N = supersetSize * dimension
superset = matrix(1000 * (rnorm(N) ^ 3 + 2 * runif(N) ^ 2 +
                  3 * rgamma(N, 5, 1) + 4), ncol = dimension)
rm(N)


# Make up the lower and upper bounds for the solution space:
tmp = sort(sample(1L : supersetSize, subsetSize))
tmp2 = sort(sample(1L : supersetSize, subsetSize))
lowerBounds = pmin(tmp, tmp2)
upperBounds = pmax(tmp, tmp2)
rm(tmp, tmp2)


# Exclude elements not covered by 'lowerBounds' and 'upperBounds':
remainIndex = unique(unlist(apply(cbind(lowerBounds, upperBounds), 1,
```

```
                                  function(x) x[1] : x[2])))
lowerBounds = match(lowerBounds, remainIndex)
upperBounds = match(upperBounds, remainIndex)
superset = superset[remainIndex, ]


# Plant a subset sum:
solution = apply(rbind(lowerBounds, upperBounds), 2, function(x)
  sample(x[1] : x[2], 1))
subsetSum = colSums(superset[solution, ])
subsetSumError = abs(subsetSum) * 0.01 # relative error within 1%
rm(solution)


rst = FLSSS::mFLSSSparImposeBounds(
  maxCore = 2L, len = subsetSize, mV = superset, mTarget = subsetSum,
  mME = subsetSumError, LB = lowerBounds, UB = upperBounds,
  solutionNeed = 1, tlimit = 2, dl = ncol(superset), du = ncol(superset),
  targetsOrder = NULL, useBiSrchInFB = FALSE, avgThreadLoad = 8L)


# Verify:
cat("Number of solutions = ", length(rst), "\n")
if(length(rst) > 0)
{
  cat("Solutions unique: ")
  cat(length(unique(lapply(rst, function(x) sort(x)))) == length(rst), "\n")
  cat("Solution in bounded space: ")
  cat(all(unlist(lapply(rst, function(x)
    sort(x) <= upperBounds & sort(x) >= lowerBounds))), "\n")
  cat("Solutions correct: ")
  cat(all(unlist(lapply(rst, function(x)
    abs(colSums(superset[x, ]) - subsetSum) <= subsetSumError))), "\n")
} else
{
  cat("No solutions exist or timer ended too soon.\n")
}
```

`mFLSSSparImposeBoundsIntegerized`
An advanced version of `mFLSSSparImposeBounds()`

Description

See the description of `mFLSSSparIntegerized()`.

Usage

```
mFLSSSparImposeBoundsIntegerized(
```

```
  maxCore = 7L,
  len,
  mV,
  mTarget,
  mME,
  LB = 1L:len,
  UB = (nrow(mV) - len + 1L) : nrow(mV),
  solutionNeed = 1L,
  precisionLevel = integer(ncol(mV)),
  returnBeforeMining = FALSE,
  tlimit = 60,
  dl = ncol(mV),
  du = ncol(mV),
  targetsOrder = NULL,
  useBiSrchInFB = FALSE,
  avgThreadLoad = 8L,
  verbose = TRUE)
```

Arguments

maxCore	See maxCore in mFLSSSpar().
len	See len in mFLSSSpar().
mV	See mV in mFLSSSpar().
mTarget	See mTarget in mFLSSSpar().
mME	See mME in mFLSSSpar().
LB	See LB in FLSSS().
UB	See UB in FLSSS().
solutionNeed	See solutionNeed in mFLSSSpar().
precisionLevel	See precisionLevel in mFLSSSparIntegerized().
returnBeforeMining	See returnBeforeMining in mFLSSSparIntegerized().
tlimit	See tlimit in mFLSSSpar().
dl	See dl in mFLSSSpar().
du	See dl in mFLSSSpar().
targetsOrder	See targetsOrder in mFLSSSparImposeBounds().
useBiSrchInFB	See useBiSrchInFB in mFLSSSpar().
avgThreadLoad	See avgThreadLoad in mFLSSSpar().
verbose	If TRUE, prints mining progress.

Value

See Value in mFLSSSparIntegerized().

Note

32-bit architecture unsupported.

Examples

```
if(.Machine$sizeof.pointer == 8L){
# =====================================================================================
# 64-bit architecture required.
# =====================================================================================
# rm(list = ls()); gc()
subsetSize = 7L
supersetSize = 60L
dimension = 5L # dimensionality


# Create a superset at random:
N = supersetSize * dimension
superset = matrix(1000 * (rnorm(N) ^ 3 + 2 * runif(N) ^ 2 +
                  3 * rgamma(N, 5, 1) + 4), ncol = dimension)
rm(N)


# Make up the lower and upper bounds for the solution space:
tmp = sort(sample(1L : supersetSize, subsetSize))
tmp2 = sort(sample(1L : supersetSize, subsetSize))
lowerBounds = pmin(tmp, tmp2)
upperBounds = pmax(tmp, tmp2)
rm(tmp, tmp2)


# 'mFLSSSparImposeBoundsIntegerized()' does not work if there are elements not
# under the hood of 'lowerBounds' + 'upperBounds'. Exclude these elements first:
remainIndex = unique(unlist(
  apply(cbind(lowerBounds, upperBounds), 1, function(x) x[1] : x[2])))
lowerBounds = match(lowerBounds, remainIndex)
upperBounds = match(upperBounds, remainIndex)
superset = superset[remainIndex, ]


# Plant a subset sum:
solution = integer(subsetSize)
solution[1] = sample(lowerBounds[1] : upperBounds[1], 1)
for(i in 2L : subsetSize)
{
  l = max(lowerBounds[i], solution[i - 1] + 1L)
  u = upperBounds[i]
  if(l == u) solution[i] = u
  else solution[i] = sample(l : u, 1)
}
subsetSum = colSums(superset[solution, ])
subsetSumError = abs(subsetSum) * 0.01 # relative error within 1%
rm(solution)
```

```
system.time({rst = FLSSS::mFLSSSparImposeBoundsIntegerized(
  maxCore = 2L, len = subsetSize, mV = superset, mTarget = subsetSum,
  mME = subsetSumError, LB = lowerBounds, UB = upperBounds,
  solutionNeed = 1, tlimit = 3, dl = ncol(superset), du = ncol(superset),
  targetsOrder = NULL, useBiSrchInFB = FALSE, avgThreadLoad = 8L)})


# Compare the time cost of 'mFLSSSparImposeBoundsIntegerized()' and
# 'mFLSSSparImposeBounds()'. The speed advantage of 'mFLSSSparIntegerized()'
# may not be pronounced for toy examples.
system.time(FLSSS::mFLSSSparImposeBounds(
  maxCore = 2L, len = subsetSize, mV = superset, mTarget = subsetSum,
  mME = subsetSumError, LB = lowerBounds, UB = upperBounds,
  solutionNeed = 1, tlimit = 2, dl = ncol(superset), du = ncol(superset),
  targetsOrder = NULL, useBiSrchInFB = FALSE, avgThreadLoad = 8L))


# Verify:
cat("Number of solutions = ", length(rst$solution), "\n")
if(length(rst$solution) > 0)
{
  cat("Solutions unique: ")
  cat(length(unique(lapply(rst$solution, function(x)
    sort(x)))) == length(rst$solution), "\n")
  cat("Solution in bounded space: ")
  cat(all(unlist(lapply(rst$solution, function(x)
    sort(x) <= upperBounds & sort(x) >= lowerBounds))), "\n")


  cat("Solutions correct regarding integerized data: ")
  cat(all(unlist(lapply(rst$solution, function(x)
    abs(colSums(rst$INT$mV[x, ]) - rst$INT$mTarget) <= rst$INT$mME))), "\n")


  cat("Solutions correct regarding original data: ")
  boolean = all(unlist(lapply(rst$solution, function(x)
    abs(colSums(superset[x, ]) - subsetSum) <= subsetSumError)))
  cat(boolean, "\n")
  if(!boolean)
  {
    cat("The given error threshold relative to subset sum:\n")
    givenRelaErr = round(abs(subsetSumError / subsetSum), 5)
    cat(givenRelaErr, "\n")


    cat("Solution subset sum relative error:\n")
    tmp = lapply(rst$solution, function(x)
    {
      err = round(abs(colSums(superset[x, ]) / subsetSum -1), 5)
      for(i in 1L : length(err))
      {
```

```
        if(givenRelaErr[i] < err[i]) message(paste0(err[i], " "), appendLF = FALSE)
        else cat(err[i], "")
      }
      cat("\n")
    })
    cat("Integerization caused the errors. Future versions of")
    cat("'mFLSSSparIntegerized()' would have a parameter of precision level.\n")
  }
} else
{
  cat("No solutions exist or timer ended too soon.\n")
}

# =======================================================================================
# =======================================================================================
}
```

`mFLSSSparIntegerized` *An advanced version of* `mFLSSSpar()`

Description

This function maps a real-value multidimensional Subset Sum problem to the integer domain with minimal precision loss. Those integers are further compressed in 64-bit buffers for dimension reduction and SWAR (SIMD within a register) that could lead to substantial acceleration.

Usage

```
mFLSSSparIntegerized(
  maxCore = 7L,
  len,
  mV,
  mTarget,
  mME,
  solutionNeed = 1L,
  precisionLevel = integer(ncol(mV)),
  returnBeforeMining = FALSE,
  tlimit = 60,
  dl = ncol(mV),
  du = ncol(mV),
  useBiSrchInFB = FALSE,
  avgThreadLoad = 8L,
  verbose = TRUE
  )
```

Arguments

`maxCore`	See `maxCore` in `mFLSSSpar()`.

`len` See `len` in `mFLSSSpar()`.

`mV` See `mV` in `mFLSSSpar()`.

`mTarget` See `mTarget` in `mFLSSSpar()`.

`mME` See `mME` in `mFLSSSpar()`.

`solutionNeed` See `solutionNeed` in `mFLSSSpar()`.

`precisionLevel` An integer vector of size equal to the dimensionality of `mV`. This argument controls the precision of real-to-integer conversion.

If `precisionLevel[i] = 0`, `mV[,i]` is shifted, scaled and rounded to the nearest integers such that the maximum becomes no less than `nrow(mV) * 8`.

If `precisionLevel[i] > 0`, e.g. `precisionLevel[i] = 1000`, `mV[,i]` is shifted, scaled and rounded to the nearest integers such that the maximum becomes no less than 1000.

If `precisionLevel[i] = -1`, `mV[,i]` is shifted, scaled and rounded to the nearest integers such that ranks of elements stay the same.

The shift operator contributes no precision loss. It only lowers the number of bits used for storing integers.

`returnBeforeMining` A boolean value. If `TRUE`, function returns the integerized `mV`, `mTarget` and `mME`.

`tlimit` See `tlimit` in `mFLSSSpar()`.

`dl` See `dl` in `mFLSSSpar()`.

`du` See `du` in `mFLSSSpar()`.

`useBiSrchInFB` See `useBiSrchInFB` in `mFLSSSpar()`.

`avgThreadLoad` See `avgThreadLoad` in `mFLSSSpar()`.

`verbose` If TRUE, prints mining progress.

Value

A list of two.

`Value$solution` is a list of solution index vectors.

`Value$INT` is a list of three.

`Value$INT$mV` is the integerized superset.

`Value$INT$mTarget` is the integerized subset sum.

`Value$INT$mME` is the integerized subset sum error threshold.

`Value$INT$compressedDim` is the dimensionality after integerization.

Note

32-bit architecture unsupported.

Examples

```
if(.Machine$sizeof.pointer == 8L){
# ==================================================================================
# 64-bit architecture required.
# ==================================================================================
# rm(list = ls()); gc()
subsetSize = 7L
supersetSize = 60L
dimension = 5L # dimensionality


# Create a supertset at random:
N = supersetSize * dimension
superset = matrix(1000 * (rnorm(N) ^ 3 + 2 * runif(N) ^ 2 + 3 * rgamma(N, 5, 1) + 4),
                  ncol = dimension)
rm(N)


# Plant a subset sum:
solution = sample(1L : supersetSize, subsetSize)
subsetSum = colSums(superset[solution, ])
subsetSumError = abs(subsetSum) * 0.01 # relative error within 1%
rm(solution)


# Mine subsets, dimensions fully bounded
system.time({rst = FLSSS::mFLSSSparIntegerized(
  maxCore = 2, len = subsetSize, mV = superset, mTarget = subsetSum,
  mME = subsetSumError, solutionNeed = 2, dl = ncol(superset),
  du = ncol(superset), tlimit = 2, useBiSrchInFB = FALSE, avgThreadLoad = 8L)})


# Compare the time cost of 'mFLSSSparIntegerized()' and 'mFLSSSpar()'. The
# speed advantage of 'mFLSSSparIntegerized()' may not be pronounced for toy
# examples.
system.time(FLSSS::mFLSSSpar(
  maxCore = 2, len = subsetSize, mV = superset, mTarget = subsetSum,
  mME = subsetSumError, solutionNeed = 2, dl = ncol(superset),
  du = ncol(superset), tlimit = 2, useBiSrchInFB = FALSE, avgThreadLoad = 8L))


# Verify:
cat("Number of solutions = ", length(rst$solution), "\n")
if(length(rst$solution) > 0)
{
  cat("Solutions unique: ")
  cat(length(unique(lapply(rst$solution, function(x)
    sort(x)))) == length(rst$solution), "\n")


  cat("Solutions correct regarding integerized data: ")
  cat(all(unlist(lapply(rst$solution, function(x)
```

```
    abs(colSums(rst$INT$mV[x, ]) - rst$INT$mTarget) <= rst$INT$mME))), "\n")


  cat("Solutions correct regarding original data: ")
  boolean = all(unlist(lapply(rst$solution, function(x)
    abs(colSums(superset[x, ]) - subsetSum) <= subsetSumError)))
  cat(boolean, "\n")
  if(!boolean)
  {
    cat("The given error threshold relative to subset sum:\n")
    givenRelaErr = round(abs(subsetSumError / subsetSum), 5)
    cat(givenRelaErr, "\n")


    cat("Solution subset sum relative error:\n")
    tmp = lapply(rst$solution, function(x)
    {
      err = round(abs(colSums(superset[x, ]) / subsetSum -1), 5)
      for(i in 1L : length(err))
      {
        if(givenRelaErr[i] < err[i]) message(paste0(err[i], " "), appendLF = FALSE)
        else cat(err[i], "")
      }
      cat("\n")
    })
    cat("Integerization caused the errors. Future versions of")
    cat("'mFLSSSparIntegerized()' would have a parameter of precision level.\n")
  }
} else
{
  cat("No solutions exist or time ended too soon.\n")
}


# Mine subsets, the first 3 dimensions lower bounded,
# the last 4 dimension upper bounded
rst = FLSSS::mFLSSSparIntegerized(
  maxCore = 2, len = subsetSize, mV = superset, mTarget = subsetSum,
  mME = subsetSumError, solutionNeed = 2, dl = 3L, du = 4L, tlimit = 2,
  useBiSrchInFB = FALSE, avgThreadLoad = 8L)


# Verify:
cat("Number of solutions = ", length(rst$solution), "\n")
if(length(rst$solution) > 0)
{
  cat("Solutions unique: ")
  cat(length(unique(lapply(rst$solution, function(x)
    sort(x)))) == length(rst$solution), "\n")


  cat("Solutions correct regarding integerized data: ")
  cat(all(unlist(lapply(rst$solution, function(x)
```

```
{
  lowerBoundedDim = 1L : 3L
  lowerBounded = all(colSums(rst$INT$mV[x, lowerBoundedDim]) >=
    rst$INT$mTarget[lowerBoundedDim] - rst$INT$mME[lowerBoundedDim])


  upperBoundedDim = (ncol(rst$INT$mV) - 3L) : ncol(rst$INT$mV)
  upperBounded = all(colSums(rst$INT$mV[x, upperBoundedDim]) <=
    rst$INT$mTarget[upperBoundedDim] + rst$INT$mME[upperBoundedDim])


  lowerBounded & upperBounded
}))), "\n")
} else
{
  cat("No solutions exist or timer ended too soon.\n")
}
# =====================================================================================
# =====================================================================================
}
```

`mmKnapsack` *Multithreaded multidimensional Knapsack problem solver*

Description

Given a set of items characterized by a profit attribute and multiple cost attributes, `mmKnapsack()` seeks a subset that maximizes the total profit while the subset sum in each cost dimension is upper bounded. The function applies to the 0-1 Knapsack problem. For the bounded or unbounded Knapsack problem, one can replicate items as needed and turn the problem into 0-1 Knapsack. Profits and costs should be nonnegative. Negative values in data can be neutralized by shifting and scaling.

Usage

```
mmKnapsack(
  maxCore = 7L,
  len,
  itemsProfits,
  itemsCosts,
  capacities,
  heuristic = FALSE,
  tlimit = 60,
  useBiSrchInFB = FALSE,
  threadLoad = 8L,
  verbose = TRUE
  )
```

Arguments

`maxCore`	Maximal threads to invoke. Ideally `maxCore` should not surpass the total logical processors on machine.
`len`	An integer as the subset size. See `len` in `FLSSS()`.
`itemsProfits`	A nonnegative numeric vector of size equal to the number of items.
`itemsCosts`	A nonnegative numeric matrix. Number of rows equals number of items. Number of columns equals number of cost dimensions.
`capacities`	A numeric vector of size equal to the number of cost dimensions. `capacities[i]` upper-bounds the total cost in `itemsCosts[, i]`.
`heuristic`	A boolean value. If `TRUE`, the function returns once it has found a solution whose sum of ranks of the profits is no less than that of the optimal. See Examples.
`tlimit`	A numeric value. Enforce function to return in `tlimit` seconds.
`useBiSrchInFB`	See `useBiSrchInFB` in `FLSSS()`.
`threadLoad`	See `avgThreadLoad` in `mFLSSSpar()`.
`verbose`	If `TRUE`, function prints progress.

Value

If no solution, an empty list, otherwise a list of five:

`solution`	The optimal solution.
`selectionCosts`	Solution costs.
`budgets`	Knapsack capacities.
`selectionProfit`	Solution total profit.
`unconstrainedMaxProfit`	Maximal profit given infinite budgets.

Examples

```
# ===========================================================================
# Play random numbers
# ===========================================================================
# rm(list = ls()); gc()
subsetSize = 6
supersetSize = 60
NcostsAttr = 4


  # Make up costs for each item.
  costs = abs(6 * (rnorm(supersetSize * NcostsAttr) ^ 3 +
                   2 * runif(supersetSize * NcostsAttr) ^ 2 +
                   3 * rgamma(supersetSize * NcostsAttr, 5, 1) + 4))
  costs = matrix(costs, ncol = NcostsAttr)
```

```
  # Make up cost limits.
  budgets = apply(costs, 2, function(x)
  {
    x = sort(x)
    Min = sum(x[1L : subsetSize])
    Max = sum(x[(supersetSize - subsetSize + 1L) : supersetSize])
    runif(1, Min, Max)
  })


  # Make up item profits.
  gains = rnorm(supersetSize) ^ 2 * 10000 + 100


  rst1 = FLSSS::mmKnapsack(
    maxCore = 2L, len = subsetSize, itemsProfits = gains, itemsCosts = costs,
    capacities = budgets, heuristic = FALSE, tlimit = 60, useBiSrchInFB = FALSE,
    threadLoad = 4L, verbose = TRUE)


  # Let 'x' be the solution given 'heuristic = TRUE'. The sum of ranks of the
  # profits subsetted by 'x' would be no less than that of the optimal solution.
  rst2 = FLSSS::mmKnapsack(
    maxCore = 2L, len = subsetSize, itemsProfits = gains, itemsCosts = costs,
    capacities = budgets, heuristic = TRUE, tlimit = 60, useBiSrchInFB = FALSE,
    threadLoad = 4L, verbose = TRUE)


  # Exam difference in total profits given by the heuristic and the optimal:
  cat(length(rst1$solution)); cat(length(rst2$solution)) # See if solution exists.
  if(length(rst1$solution) > 0 & length(rst2$solution) > 0)
    sum(gains[rst2$solution]) / sum(gains[rst1$solution])




# ===========================================================================================
# Test case P08 from
# https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html
# ===========================================================================================
# rm(list = ls()); gc()
costs = matrix(c(382745, 799601, 909247, 729069, 467902,  44328,  34610, 698150,
                 823460, 903959, 853665, 551830, 610856, 670702, 488960, 951111,
                 323046, 446298, 931161,  31385, 496951, 264724, 224916, 169684),
               ncol = 1)


gains = c( 825594, 1677009, 1676628, 1523970,  943972,   97426,  69666, 1296457,
          1679693, 1902996, 1844992, 1049289, 1252836, 1319836, 953277, 2067538,
           675367,  853655, 1826027,   65731,  901489,  577243, 466257,  369261)
```

```
budgets = 6404180


# 'mmKnapsack()' is designed for the multidimensional Knapsack and may not
# be ideal for one-dimensional 0-1 Knapsack regarding computing speed.
# 'len = 0' causes substantial deceleration. Looping 'len' over possible
# values is recommended if 'len' is ungiven.
rst1 = FLSSS::mmKnapsack(
  maxCore = 2L, len = 12L, itemsProfits = gains, itemsCosts = costs,
  capacities = budgets, heuristic = FALSE, tlimit = 2, threadLoad = 4L,
  verbose = TRUE)
rst1 = sort(rst1$solution)


cat("Correct solution:\n1 2 4 5 6 10 11 13 16 22 23 24\nFLSSS solution =\n")
cat(rst1, "\n")




# ===================================================================================
# Test case P07 from
# https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html
# ===================================================================================
costs = matrix(c(70, 73, 77, 80, 82, 87, 90, 94, 98, 106, 110, 113, 115, 118, 120),
               ncol = 1)


gains = c(135, 139, 149, 150, 156, 163, 173, 184, 192, 201, 210, 214, 221, 229, 240)


budgets = 750


rst2 = FLSSS::mmKnapsack(
  maxCore = 2L, len = 8L, itemsProfits = gains, itemsCosts = costs,
  capacities = budgets, heuristic = FALSE, tlimit = 2,
  threadLoad = 4L, verbose = TRUE)
rst2 = sort(rst2$solution)


cat("Correct solution:\n1 3 5 7 8 9 14 15\nFLSSS solution =\n")
cat(rst2, "\n")
```

`mmKnapsackIntegerized` *An advanced version of* `mmKnapsack()`

Description

See the description of `mFLSSSparIntegerized()`.

Usage

```
mmKnapsackIntegerized(
  maxCore = 7L,
  len,
  itemsProfits,
  itemsCosts,
  capacities,
  heuristic = FALSE,
  precisionLevel = integer(length(capacities)),
  returnBeforeMining = FALSE,
  tlimit = 60,
  useBiSrchInFB = FALSE,
  threadLoad = 8L,
  verbose = TRUE
  )
```

Arguments

`maxCore`	See `maxCore` in `mmKnapsack()`.
`len`	See `len` in `mmKnapsack()`.
`itemsProfits`	See `itemsProfits` in `mmKnapsack()`.
`itemsCosts`	See `itemsCosts` in `mmKnapsack()`.
`capacities`	See `capacities` in `mmKnapsack()`.
`heuristic`	See `heuristic` in `mmKnapsack()`.
`precisionLevel`	See `precisionLevel` in `mFLSSSparIntegerized()`.
`returnBeforeMining`	See `returnBeforeMining` in `mFLSSSparIntegerized()`.
`tlimit`	See `tlimit` in `mmKnapsack()`.
`useBiSrchInFB`	See `useBiSrchInFB` in `FLSSS()`.
`threadLoad`	See `avgThreadLoad` in `mFLSSSpar()`.
`verbose`	If `TRUE`, function prints progress.

Value

A list of six:

`solution`	The optimal solution.
`selectionCosts`	Solution costs.
`budgets`	Knapsack capacities.
`selectionProfit`	Solution total profit.
`unconstrainedMaxProfit`	Maximal profit given infinite budgets.
`INT`	A list of four:

`INT$mV` The integerized superset.

`INT$mTarget` The integerized subset sum.

`INT$mME` The integerized subset sum error threshold.

`INT$compressedDim`
The dimensionality after integerization.

Note

32-bit architecture unsupported.

Examples

```
if(.Machine$sizeof.pointer == 8L){
# =====================================================================================
# 64-bit architecture required.
# =====================================================================================
# =====================================================================================
# Play random numbers
# =====================================================================================
# rm(list = ls()); gc()
subsetSize = 6
supersetSize = 60
NcostsAttr = 4


# Make up costs for each item.
costs = abs(6 * (rnorm(supersetSize * NcostsAttr) ^ 3 +
  2 * runif(supersetSize * NcostsAttr) ^ 2 +
  3 * rgamma(supersetSize * NcostsAttr, 5, 1) + 4))
costs = matrix(costs, ncol = NcostsAttr)


# Make up cost limits.
budgets = apply(costs, 2, function(x)
{
  x = sort(x)
  Min = sum(x[1L : subsetSize])
  Max = sum(x[(supersetSize - subsetSize + 1L) : supersetSize])
  runif(1, Min, Max)
})


# Make up item profits.
gains = rnorm(supersetSize) ^ 2 * 10000 + 100


rst1 = FLSSS::mmKnapsackIntegerized(
  maxCore = 2L, len = subsetSize, itemsProfits = gains, itemsCosts = costs,
  capacities = budgets, heuristic = FALSE, tlimit = 2, useBiSrchInFB = FALSE,
  threadLoad = 4L, verbose = TRUE)
```

```
# Examine if 'mmKnapsackIntegerized()' gives the same solution as 'mmKnapsack()'.
rst2 = FLSSS::mmKnapsack(
  maxCore = 2L, len = subsetSize, itemsProfits = gains, itemsCosts = costs,
  capacities = budgets, heuristic = FALSE, tlimit = 2, useBiSrchInFB = FALSE,
  threadLoad = 4L, verbose = TRUE)
# Possible differences in solutions are due to real-integer conversion




# Let 'x' be the solution given 'heuristic = T'. The sum of ranks of the
# profits subsetted by 'x' would be no less than that of the optimal solution.
rst3 = FLSSS::mmKnapsackIntegerized(
  maxCore = 2L, len = subsetSize, itemsProfits = gains, itemsCosts = costs,
  capacities = budgets, heuristic = TRUE, tlimit = 2, useBiSrchInFB = FALSE,
  threadLoad = 4L, verbose = TRUE)


# Exam difference in total profits given by the heuristic and the optimal:
if(length(rst3$solution) > 0 & length(rst1$solution) > 0)
  sum(gains[rst3$solution]) / sum(gains[rst1$solution])





# ===========================================================================================
# Test case P08 from
# https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html
# ===========================================================================================
costs = matrix(c(382745, 799601, 909247, 729069, 467902,  44328,  34610, 698150,
                 823460, 903959, 853665, 551830, 610856, 670702, 488960, 951111,
                 323046, 446298, 931161,  31385, 496951, 264724, 224916, 169684),
               ncol = 1)


gains = c( 825594, 1677009, 1676628, 1523970,  943972,   97426,  69666, 1296457,
          1679693, 1902996, 1844992, 1049289, 1252836, 1319836, 953277, 2067538,
           675367,  853655, 1826027,   65731,  901489,  577243, 466257,  369261)


budgets = 6404180


# 'mmKnapsackIntegerized()' is designed for the multidimensional Knapsack
# and may not be ideal for one-dimensional 0-1 Knapsack regarding computing speed.
# 'len = 0' would cause severe deceleration. Looping 'len' over possible
# values is recommended if 'len' is ungiven.
rst = FLSSS::mmKnapsackIntegerized(
  maxCore = 2L, len = 12L, itemsProfits = gains, itemsCosts = costs,
  capacities = budgets, heuristic = FALSE, tlimit = 2, threadLoad = 4L, verbose = TRUE)
rst = sort(rst$solution)
```

```
cat("Correct solution:\n1 2 4 5 6 10 11 13 16 22 23 24\nFLSSS solution =\n")
cat(rst, "\n")
# The difference is due to rounding errors in real-integer conversion. The default
# 'precisionLevel' shifts, scales and rounds 'itemCosts' such that its
# maximal element is no less than 8 times the number of items.


# Increase the precision level
rst = FLSSS::mmKnapsackIntegerized(
  maxCore = 2L, len = 12L, itemsProfits = gains, itemsCosts = costs,
  capacities = budgets, heuristic = FALSE, precisionLevel = rep(500L, 1),
  tlimit = 2, threadLoad = 4L, verbose = TRUE)
# 'precisionLevel = 500' shifts, scales and rounds 'itemCosts' such that its
# maximal element is no less than 500.
rst = sort(rst$solution)
cat("Correct solution:\n1 2 4 5 6 10 11 13 16 22 23 24\nFLSSS solution =\n")
cat(rst, "\n")
}
# ===========================================================================
# ===========================================================================
```

Index