Package ‘FLSSS’

August 28, 2018

Type Package

Title Mining Rigs for Specialized Subset Sum, Multi-Subset Sum, Multidimensional Subset Sum, Multidimensional Knapsack, Generalized Assignment Problems

Version 7.6

Author Charlie Wusuo Liu

Maintainer Charlie Wusuo Liu <liuwusuo@gmail.com>

Description Specialized solvers for combinatorial optimization problems in the Subset Sum family. These solvers differ from the mainstream in the options of (i) subset size restriction, (ii) bounds on the subset elements, (iii) mining real-value sets with predefined subset sum errors, and (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 latter is creatively scheduled in a multi-threaded environment, and the framework offers strong applications to the multidimensional Knapsack and the Generalized Assignment problems. Package 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) an option of mapping real-domain problems to the integer domain with controlled precision loss, and those integers are further zipped non-uniformly in 64-bit buffers. Arithmetic on compressed integers has a novel design with virtually zero speed lag relative to that on normal integers, and the consequent reduction in dimensionality often leads to substantial acceleration. Compilation with aggressive optimization, e.g. g++ '-Ofast', may speed up mining on some platforms. Package documentation (<arXiv:1612.04484v2>) is outdated as the time of writing.

License GPL-3

Encoding UTF-8

LazyData true

ByteCompile true

Imports Rcpp (>= 0.12.13), RcppParallel

LinkingTo Rcpp, RcppParallel

SystemRequirements GNU make
**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

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

len
An integer as the subset size: \(0 \leq \text{len} < \text{length}(v)\). If \(\text{len} = 0\), FLSSS() mines subsets without size restriction. \(\text{len} \leq 0\) would be most likely slower than looping \(\text{len}\) over \(1 : (\text{length}(v) - 1)\). 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 \(\text{len}\) as the lower bounds of the solution space: for any solution \(x\), \(\text{LB}[i] \leq x[i]\). Custom \(\text{LB}\) should be no less than \(1L : \text{len}\) element-wisely. Every element in \(v\) should be within the range enclosed by \(\text{LB}\) and \(\text{UB}\).

UB
An integer vector of size \(\text{len}\) as the upper bounds of the solution space: for any solution \(x\), \(x[i] \leq \text{UB}[i]\). Custom \(\text{UB}\) should be no greater than \((\text{length}(v) - \text{len} + 1L) : \text{length}(v)\) element-wisely. Every element in \(v\) should be within the range enclosed by \(\text{LB}\) and \(\text{UB}\).

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

tlimit
A numeric value. Enforce function to return in \(\text{tlimit}\) seconds.

useBiSrchInFB
A boolean value. If \(\text{TRUE}\), 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 \(\text{length}(v)\) is substantial (> 10000) while \(\text{len}\) is small (< 5).

useFloat
A boolean value. If \(\text{TRUE}\), convert data to single-precision floats. This argument is mainly for research. \(\text{useFloat} = \text{TRUE}\) is highly unrecommended.

Details

If \(\text{len} = 0\), FLSSS() would (1) reset \(\text{len}\) to \(\text{length}(v)\), (2) pad \(\text{len}\) zeros at the beginning of \(v\) and sort \(v\), (3) search for size-\(\text{len}\) 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 1: 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)
supersetSum = runif(1, sum(superset[1L : subsetSize]), sum(superset[(supersetSize - subsetSize + 1L) : supersetSize]))
supersetSumError = 1e-3
subsetSum = runif(1L, sum(superset[1L : subsetSize]), sum(superset[(supersetSize - subsetSize + 1L) : supersetSize]))

# 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)

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)
for(i in 2L : subetSize)
{
  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, 28334308, 32036573, 32847411, 34570996, 34574989, 43633828,
4403100, 47724996, 51905122, 52691025, 53600924, 56874435, 5820767,
60225777, 60639161, 60085288, 60890325, 61742932, 63780621, 63786876,
65167464, 66224357, 67198760, 69366452, 71163068, 72338751, 72960793,
73197629, 76148392, 77779087, 78308432, 81196763, 82741805, 85315243,
86446883, 87820632, 89819002, 90604146, 93761290, 97928291, 98315839,
310128088, -441483864, -54814311, -645883459, -14910919, 305170449, -248934805,
-1108328430, -52786318, -192539936, -1005874405, -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 = 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 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 subsets 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 categories, the number of items in each
# category, select the least number of items to fulfill at least 95% of the knapsack's
# capacity.
# ..............................................................................................................
flsssmultiset

Description

Find a subset 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, useFloat = FALSE )

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().
useFloat See useFloat in FLSSS().

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:
\[
\text{rst} = \text{FLSSS::FLSSSmultiset(len = subsetSizes, buckets = supersets, target = subsetsSum,}
\]
\[
\quad \text{ME = subsetsSumError, solutionNeed = 3, tlimit = 4)}
\]
\[
\text{cat("Number of solutions =", length(rst), "\n")}
\]

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

---

**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, 
    useBiSrchInFB = FALSE, 
    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.

useBiSrchInFB See `useBiSrchInFB` in `FLSSS()`.

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

verbose If TRUE, function prints progress.

Value

A list of size ten.

Value$assignedAgents is an integer vector, the mining result. The i-th element denotes the agent assigned to task i.

Value$assignmentProfit is the profit resulted from such assignment.

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

Value$agentsBudgets is a numeric vector. Value$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.

FLSSStargets is the subset sum targets going into the multidimensional Subset Sum miner.

FLSSSme is the subset sum errors going into the multidimensional Subset Sum miner.

foreShadowFLSSSvec is the multidimensional vector before comonotonization.

Examples

```r
# Play random numbers
rm(list = ls()); gc()
agents = 5L
tasks = 12L
costs = t(as.data.frame(lapply(1L : agents, function(x) runif(tasks) * 10000)))
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 = 2L, agentsCosts = costs, agentsProfits = profits,
  agentsBudgets = budgets, heuristic = FALSE, tlimit = 2,
  useBisrchInFB = FALSE, 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 = matrix(c(27, 12, 12, 16, 24, 31, 41, 13,
  14, 5, 37, 9, 36, 25, 1, 34,
GAPintegerized

34, 34, 20, 9, 19, 19, 3, 34), ncol = tasks)
costs = 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 = tasks)
budgets = c(26, 25, 34)

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

knownOptSolution = as.integer(c(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
# FLSSS::GAP() generated a better solution.

GAPintegerized  An advanced version of GAP().

Description

See the description of mFLSSSparIntegerized().

Usage

GAPintegerized(
maxCore = 7L,
agentsCosts,
agentsProfits,
agentsBudgets,
heuristic = FALSE,
precisionLevel = integer(length(agentsBudgets)),
returnBeforeMining = 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.

- **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 j’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().

- **precisionLevel**
  See precisionLevel in mFLSSParsIntegerized().

- **returnBeforeMining**
  See returnBeforeMining in mFLSSParsIntegerized().

- **tlimit**
  See tlimit in FLSS().

- **useBiSrChInFB**
  See useBiSrChInFB in FLSS().

- **threadLoad**
  See avgThreadLoad in mFLSSPars().

- **verbose**
  If TRUE, function prints progress.

**Value**

See function value of GAP().

**Note**

32-bit architecture unsupported.

**Examples**

```r
if(.Machine$sizeof.pointer == 8L){
  # ====================================================================================
  # 64-bit architecture required.
  # ====================================================================================
  # Debugger.
  # ====================================================================================
  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

  # ====================================================================================
  # Debugger.
  # ====================================================================================
```
# 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 = FLSS::GAPintegerized(maxCore = 2L, agentsCosts = costs, agentsProfits = profits,
agentsBudgets = budgets, heuristic = FALSE,
precisionLevel = rep(tasks * 4L, agents), tlimit = 2,
useBiSrchInFB = FALSE, 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 = matrix(c(27, 12, 12, 16, 24, 31, 41, 13,
mFLSSspar

Multithreaded multidimensional Subset Sum given error thresholds

Description

The multidimensional version of FLSS().

Usage

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

```r
14, 5, 37, 9, 36, 25, 1, 34,
34, 34, 20, 9, 19, 19, 3, 34), ncol = tasks)
costs = 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 = tasks)
budgets = c(26, 25, 34)

rst = FLSS::GAPintegerized(maxCore = 2L, agentsCosts = costs, agentsProfits = profits,
  agentsBudgets = budgets, heuristic = FALSE, tlimit = 2,
  useBiSrchInFB = FALSE, 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 frim FLSS::GAP():
rst$assignmentProfit
# FLSS::GAP() generated a better solution.
# =~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# =~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#)
```
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 FLSS().
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 FLSS().
tlimit See tlimit in FLSS().
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 FLSS().
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 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)

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

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

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

# Mine subsets, the first 3 dimensions lower bounded, 
# the last 4 dimension upper bounded

\[
\text{rst} = \text{FLSSS::mFLSSSpar}(\text{maxCore} = 2, \text{len} = \text{subsetSize}, \text{mV} = \text{superset}, \text{mTarget} = \text{subsetSum}, \text{mME} = \text{subsetSumError}, \text{solutionNeed} = 2, \text{dl} = 3L, \text{du} = 4L, \text{tlimit} = 2, \text{useBiSrchInFB} = \text{FALSE}, \text{avgThreadLoad} = 8L)
\]

# Verify:
\[
\begin{align*}
\text{cat("Number of solutions = ", length(rst), "\n")} \\
\text{if(length(rst) > 0)} \\
\quad \{ \\
\qquad \text{cat("Solutions unique: ")} \\
\qquad \text{cat(length(unique(lapply(rst, function(x) sort(x)))) == length(rst), "\n")} \\
\qquad \text{cat("Solutions correct: ")} \\
\qquad \text{cat(all(unlist(lapply(rst, function(x) abs(colSums(superset[x,]) - subsetSum) <= subsetSumError)))}, "\n")
\quad \}
\end{align*}
\]

\[
\begin{align*}
\text{lowerBoundedDim} &= 1L : 3L \\
\text{lowerBounded} &= \text{all(colSums(superset[x, lowerBoundedDim]) >=} \\
& \quad \text{subsetSum[lowerBoundedDim] - subsetSumError[lowerBoundedDim]})
\end{align*}
\]

\[
\begin{align*}
\text{upperBoundedDim} &= (\text{ncol(superset) - 3L}) : \text{ncol(superset)} \\
\text{upperBounded} &= \text{all(colSums(superset[x, upperBoundedDim]) <=} \\
& \quad \text{subsetSum[upperBoundedDim] + subsetSumError[upperBoundedDim]})
\end{align*}
\]
Description

For comparison, function mFLSSpar() puts no bounds on the solution space so it can sort \( mV \) internally in a special order to accelerate computing speed. Function mFLSSPariPoseBounds() bounds the solution space and keeps the order of \( mV \). See Examples for its application to the generalized assignment problem [1].

Usage

mFLSSPariPoseBounds(
    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 mFLSSpar().
len See len in mFLSSpar().
MV See MV in mFLSSpar().
mTarget See mTarget in mFLSSpar().
mME See mME in mFLSSpar().
LB See LB in FLSSS().
UB
solutionNeed
tlimit
dl
du
targetsOrder
value
references
examples
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%
rmsolution

rst = FLSS::mFLSSparImposeBoundsIntegerized(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")
}

mFLSSparImposeBoundsIntegerized
An advanced version of mFLSSparImposeBounds()

Description

See the description of mFLSSparIntegerized().
mFLSSsparImposeBoundsIntegerized

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 mFLSSpar().
mTarget See mTarget in mFLSSpar().
mME See mME in mFLSSpar().
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 mFLSSpar().
dl See dl in mFLSSpar().
du See dl in mFLSSpar().
targetsOrder See targetsOrder in mFLSSsparImposeBounds().
useBiSrchInFB See useBiSrchInFB in mFLSSpar().
avgThreadLoad See avgThreadLoad in mFLSSpar().
verbose If TRUE, prints mining progress.

Value

See Value in mFLSSsparIntegerized().
Note

32-bit architecture unsupported.

Examples

```r
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)

  # 'mFLSSparImposeBoundsIntegerized()' 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)
  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)
```
mFLSSsparImposeBoundsIntegerized

```r
data <- NULL
# Compare the time cost of 'mFLSSsparImposeBoundsIntegerized()' and
# 'mFLSSsparImposeBounds()'. The speed advantage of 'mFLSSsparIntegerized()' 
# may not be pronounced for toy examples.

system.time(rst <- flsss::mFLSSsparimposeboundsintegerized(
  maxCore = 2L, len = subsetSize, mV = superset, mTarget = subsetSum, 
  mME = subsetSumError, LB = lowerBounds, UB = upperBounds, 
  solutionNeed = 1L, tlimit = 3L, dI = ncol(superset), du = ncol(superset), 
  targetsOrder = NULLL, useBiSrchInFB = FALSE, avgThreadLoad = 8L)

data <- NULL

# Verify:

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:
    givenRelaErr = round(abs(subsetSumError / subsetSum), 5)
    cat(givenRelaErr, "\n")

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

An advanced version of mFLSSpar()

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

mFLSSparIntegerized(
  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 mFLSSpar().
\texttt{mFLSSParIntegerized}

- \texttt{len}: See \texttt{len} in \texttt{mFLSSPar()}.
- \texttt{mV}: See \texttt{mV} in \texttt{mFLSSPar()}.
- \texttt{mTarget}: See \texttt{mTarget} in \texttt{mFLSSPar()}.
- \texttt{mME}: See \texttt{mME} in \texttt{mFLSSPar()}.
- \texttt{solutionNeed}: See \texttt{solutionNeed} in \texttt{mFLSSPar()}. 
- \texttt{precisionLevel}: An integer vector of size equal to the dimensionality of \texttt{mV}. This argument controls the precision of real-to-integer conversion. If \texttt{precisionLevel[i] = 0}, \texttt{mV[,i]} is shifted, scaled and rounded to the nearest integers such that the maximum becomes no less than \texttt{nrow(mV) * 8}. If \texttt{precisionLevel[i] > 0}, e.g. \texttt{precisionLevel[i] = 1000}, \texttt{mV[,i]} is shifted, scaled and rounded to the nearest integers such that the maximum becomes no less than 1000. If \texttt{precisionLevel[i] = -1}, \texttt{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.
- \texttt{returnBeforeMining}: A boolean value. If \texttt{TRUE}, function returns the integerized \texttt{mV}, \texttt{mTarget} and \texttt{mME}.
- \texttt{tlimit}: See \texttt{tlimit} in \texttt{mFLSSPar()}.
- \texttt{dl}: See \texttt{dl} in \texttt{mFLSSPar()}.
- \texttt{du}: See \texttt{du} in \texttt{mFLSSPar()}.
- \texttt{useBiSrchInFB}: See \texttt{useBiSrchInFB} in \texttt{mFLSSPar()}.
- \texttt{avgThreadLoad}: See \texttt{avgThreadLoad} in \texttt{mFLSSPar()}.
- \texttt{verbose}: If \texttt{TRUE}, prints mining progress.

\textbf{Value}

A list of two.

\texttt{Value\$solution} is a list of solution index vectors.

\texttt{Value\$INT} is a list of three.

\texttt{Value\$INT\$mV} is the integerized superset.

\texttt{Value\$INT\$mTarget} is the integerized subset sum.

\texttt{Value\$INT\$mME} is the integerized subset sum error threshold.

\texttt{Value\$INT\$compressedDim} is the dimensionality after integerization.

\textbf{Note}

32-bit architecture unsupported.
Examples

```r
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)

  # 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)
    )))
  }
```
\( \text{abs(colSums(rst}\text{INT}\text{mMV}[x, ]) - \text{rst}\text{INT}\text{mTarget}) \leq \text{rst}\text{INT}\text{mME}) \), "\n"

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

\text{cat("Solution subset sum relative error:"
}
tmp = lapply(rst\$solution, function(x)
{
\text{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], ""
}
\text{cat("\n")}
\text{cat("Integerization caused the errors. Future versions of")
\text{cat("mFLSSparIntegerized()’ would have a parameter of precision level."
")
} else
{
\text{cat("No solutions exist or time ended too soon."
})

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

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

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

An index vector as the optimal solution.

Examples

```r
# Play random numbers
# make up costs for each item.
# make up cost limits.
```

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

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

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

# ==============================================================
# 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)
mmKnapsackIntegerized

```r
rst1 = sort(rst1)

cat("Correct solution:\n2 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)

cat("Correct solution:\n3 5 7 8 9 14 15\nFLSSS solution =\n")
cat(rst2, "\n")
```

---

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

**Description**

See the description of mFLSSPariIntegerized().

**Usage**

```r
mmKnapsackIntegerized(
  maxCore = 7L,
  len,
  itemsProfits,
  itemsCosts,
  capacities,
  heuristic = FALSE,
  precisionLevel = integer(length(capacities)),
)```
mmKnapsackIntegerized

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 mFLSSParIntegerized().
returnBeforeMining See returnBeforeMining in mFLSSParIntegerized().
tlimit See tlimit in mmKnapsack().
useBiSrchInFB See useBiSrchInFB in FLSS().
threadLoad See avgThreadLoad in mFLSSPar().
verbose If TRUE, function prints 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**

```r
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)
  # 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 = FLSS::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, 464298, 931161, 31385, 496951, 264724, 224916, 169684),
               ncol = 1)
```

gains = c(825569, 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 = FLSS::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:
1 2 4 5 6 10 11 13 16 22 23 24\nFLSS 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 = FLSS::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\n")
cat(rst, "\n")

# end of file
Index

FLSSS, 2
FLSSSmultiset, 7

GAP, 9
GAPintegerized, 12

mFLSSSpar, 15
mFLSSSparImposeBounds, 18
mFLSSSparImposeBoundsIntegerized, 20
mFLSSSparIntegerized, 24
mmKnapsack, 28
mmKnapsackIntegerized, 31