Package 'FLSSS'

December 14, 2016

Type Package

Title Multi-Threaded Multidimensional Fixed Size Subset Sum Solver and Extension to General-Purpose Knapsack Problem

Version 5.2

Date 2016-12-11

Author Charlie Wusuo Liu

Maintainer Charlie Wusuo Liu <liuwusuo@gmail.com>

Description A novel algorithm for solving the subset sum problem with bounded error in multidimensional real domain and its application to the general-purpose knapsack problem.

License GPL-2

Imports Rcpp,RcppParallel

LinkingTo Rcpp,RcppParallel

NeedsCompilation yes

Repository CRAN

Suggests R.rsp

VignetteBuilder R.rsp

Date/Publication 2016-12-14 08:13:13

R topics documented:

FLSSS *Single-dimensional fixed size subset sum with bounded error solver*

Description

Given subset size "len", sorted numeric set "v", subset-sum "target", error "ME", find one or more subsets of size len whose elements sum into [target - ME, target + ME]

Usage

```
FLSSS(len, v, target, ME, sizeNeeded = 1L, tlimit=60, catch=NULL,
throw=F, LB=1L:len, UB=(length(v)-len+1L):length(v), useFloat=F)
```

Arguments

len	An integer as the subset size. If 0, activate the general-case (unfixed subset size) solver. Looping over different subset sizes to solve the general subset sum problem is preferred than setting len to 0 if v is relatively large. Example II method 2 below shows how the solver progresses for the unfixed subset size scenario.
v	Sorted numeric vector as the set
target	Subset sum
ME	Error
sizeNeeded	How many solutions at least are wanted; sizeNeeded="all" returns all the solutions
tlimit	Time Limit in seconds. E.g. tlimit=10 let FLSSS stop to return all the solutions found in 10 seconds. tlimit="none" means no time limit
catch	Catch the stack object implemented as a list
throw	Throw the stack object as a list
LB	Lower bound initializer
UB	Upper bound initializer
useFloat	Whether to use single-precision floating point in computing

Details

Please read the package vignette for more details.

Value

throw=F, returns a list of integer vectors. Each vector contains a solution's indexes.

throw=T, returns a list L, L$roots is the list of solutions, L$node is the list of the stack.

Note

1. Although the function processes fixed size subset, the general subset sum problem can be transformed to a fixed size one by padding 0s in the superset and setting the subset size to half of the superset size afterwards. Please refer to example II method 2.

2. Parameter "catch" and "throw": for example, if sizeNeeded=10 and throw=1, FLSSS will return a list L where L$roots are the solutions and L$node is a "stack" list. If 10 more solutions are needed then FLSSS with catch=L$node will start looking for the 11th solution directly.

3. Number of output solutions may be greater than sizeNeeded. A resulting empty list indicates no solutions. Setting sizeNeeded="all" and tlimit="none" could take considerable amount of time when v is long and sparse. Assigning at least one of them numeric value is strongly recommended.

4. When v is real and an error of 0 is expected, it is recommended to set ME=0.01, 0.001 or whatever less than the decimal point v's elements have reached.

Examples

```
# Example I:
# play random numbers
v=100*sort(rnorm(1000)^3+2*rnorm(1000)^2+3*rnorm(1000)+4)
len=200
target=runif(1, sum(v[1:200]), sum(v[(1000-200+1):1000]))
ME=1e-4
rst=FLSSS(len, v, target, ME, tlimit=30)
if(length(rst)>0)
all(unlist(lapply(rst, function(x)sum(v[x])>=target-ME&sum(v[x])<=target+ME)))

# losses of 50 insurance companies
v=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
,81196763,82741805,85315243,86446883,87820032,89819002,90604146,9376129,
97920291,98315039,310120088)

# try which 10 of them sum up to target
len=10
target=139254955

# try finding all solutions:
FLSSS(len,v,target,0.1,"all")

# try finding all solutions in 2 seconds with error=10:
FLSSS(len,v,target,10,"all",tlimit=2)

# try finding at least 5 solutions with error=20:
FLSSS(len,v,target,20,5)

# experiment throw and catch functions:
result1=FLSSS(len, v, target, 20, "all", tlimit=0.5, throw=1)
result2=FLSSS(len, v, target, 20, "all", tlimit=0.5, catch=result1$node)
```

```
# ------------------------------------------------------------------------


# Example II:
# use FLSSS to solve the general subset sum problem:
# method 1: loop over all possible subset sizes
for(len in 1:length(v))
{
  lis=FLSSS(len,v,target,0.1)
  if(length(lis)!=0)break
}

# method 2: add 50 dummy 0s, set the subset size to 50.
v1=c(rep(0L,50), v)
v1index=c(rep(0,50L), 1L:50L) # first 50 0s indicate the dummies
sortOrder=order(v1)
v1=v1[sortOrder]
v1index=v1index[sortOrder]
lis2=FLSSS(len=50, v1, target, 0.1)
lis2=lapply(lis2, function(x)v1index[x][v1index[x]>0])
# This method is exactly how FLSSS(0, v, target, 0.1) would proceed

# exam solutions:
all(unlist(lapply(lis2, function(x)sum(v[x])>=target-0.1&
  sum(v[x])<=target+0.1)))


# ------------------------------------------------------------------------------


# Example III:
# use FLSSS to solve an unbounded single dimensional knapsack problem:

bagCapacity=361
# capacity of the knapsack

objectMass<-sort(sample(1:100,10))
# mass of each of the 10 kinds of objects that can
# be put in the knapsack

objectAmount<-sample(1:30,10)
# number of each kind

v<-unlist(mapply(rep,objectMass,objectAmount))
# set of choices

for(len in 1:length(v))
{
  lis=FLSSS(len,v,bagCapacity,0)
  # try using up the capacity of the knapsack with the smallest number of
  # objects
```

```
  if(length(lis)!=0)break
}
```

---

mFLSSSpar *Multithreaded multidimensional fixed size subset sum with bounded error solver*

Description

FLSSS extended to multidimensional real domain

Usage

```
mFLSSSpar(len, mV, mTarget, mME, viaConjugate = F, maxCore = 7L,
totalSolutionNeeded = 1L, tlimit = 60, singleTimeLimit = 10,
randomizeTargetOrder=sample(1L:(len*(nrow(mV)-len)+1), len*(nrow(mV)-len)+1),
LB = 1L:len, UB = (nrow(mV) - len + 1L):nrow(mV))
```

Arguments

len	An integer as the subset size. If 0, activate the general-case (unfixed subset size) solver. Looping over different subset sizes to solve the general subset sum problem is preferred than setting len to 0 if mV is relatively large. Example II in FLSSS help page shows how the solver progresses for the unfixed subset size scenario.
mV	A data frame as the multidimensional numeric vector/set. Each row is an element of the vector/set.
mTarget	A numeric vector as the subset sum for each dimension. Its length should equal the number of columns/dimensions in mV.
mME	A numeric vector as the error for each dimension. Should have the same length as mTarget.
viaConjugate	Please refer to the details section
maxCore	Number of threads to invoke. Better not be greater than the maximal concurrentable threads in system.
totalSolutionNeeded	Number of solutions needed.
tlimit	Time limit in seconds. Default to 60s
singleTimeLimit	Please refer to details section or the package vignette for more information.
randomizeTargetOrder	Discussed in the details section.
LB	Lower bound initializer.
UB	Upper bound initializer.

Details

Intuitively, arithmetic among single-dimensional reals can be extended into multidimensional space for solving a multidimensional fixed size subset sum (denoted by MFSSS) problem. However, this requires all dimensions in mV (the multidimensional vector/set) are comonotonic (perfectly positively rank-correlated).

We denote a MFSSS problem where all dimensions of mV are comonotonic by cMFSSS.

An algorithm is invented to tackle the general MFSSS problem. First, it will comonotonize the dimensions (transform mV such that its dimensions become comonotonic). Second, it will decompose the MFSSS problem into no more than L*(N–L)+1 (L being the subset size, N being the superset size) cMFSSS problems where each one of them will work on the same comonotonized mV but with a different subset sum target. The parameter "singleTimeLimit" aims to bound the time cost for solving each of the cMFSSS problems.

The cMFSSS problems out of the decomposition are independent, so they can be run in a parallel fashion. If parameter "randomizeTargetOrder" is set false, then the cMFSSS problems will be sorted by the associated subset sum targets, lining up for threads; if true, they will be randomly ordered inside the function. To give user the control of the order, "randomizeTargetOrder" also accepts any index vector of size L*(N–L)+1.

Experiments show randomly ordered cMFSSS increases the chance of finding a solution in shorter time. Different order of the cMFSSS problems usually results in different time costs for finding the required number of solutions; therefore, kicking off several instances of the function with different "randomizeTargetOrder" could be beneficial. However, if computing resource allows, spawning as many as possible threads is the best way to find the required number of solutions in the shortest time.

Any single-dimensional fixed size subset sum (denote it by SFSSS) problem has a conjugate: finding a subset of size L that sums to value S from a superset of size N where all the elements sum to value T is equivalent to finding a subset of size N–L that sums to T–S. The same conclusion holds for any MFSSS problem. Parameter "viaConjugate" controls whether the function should solve via the original MFSSS problem (0) or via its conjugate (1). The two different solving paths are sometimes observed having considerable efficiency gap. It is recommended to try both if the computing resource allows.

For more details, please refer to the package vignette.

Value

A list of integer vectors. Each vector is a solution's indexes.

Note

Unlike SFSSS which "rarely contains no solution" in the entire combinatorial space, the multidimensional fixed size Subset Sum "rarely contains a solution" when the number of dimensions is not trivial.

This fact could make a seemingly trivial task in FLSSS pretty heavy in mFLSSSpar. It is recommended to run mFLSSSpar with small subset size or large error a few times to learn its performance for non-trivial task.

Examples

```
L=5L # subset size

d=5L # number of dimensions

N=50L # superset size

# generate a multidimensional vector/set
v=as.data.frame(matrix(rnorm(d*N)*1000,ncol=d))

# make an underlying solution
solution=sample(1L:N, L)

# the subset sum target of the underlying solution
target=as.numeric(colSums(v[solution,]))

# bound the error as 5% of each dimension's target magnitude
ME=abs(target)*0.05

randomOrder=sample(1L:((nrow(v)-L)*L+1L), (nrow(v)-L)*L+1L)
# randomizeTargetOrder should be TRUE or FALSE or
# an index vector of size (N-L)*L+1

system.time({tmp=mFLSSSpar(L, v, target, ME, maxCore=7,
randomizeTargetOrder = randomOrder, tlimit = 5)})
# try finding at least one solution within total time 5 seconds
# with 7 CPUs. Change tlimit for heavier tasks. The 5s limit is
# only for passing R package check

if(length(tmp)!=0)abs(as.numeric(colSums(v[tmp[[1]],]))/target-1)
# exam the solution

system.time({tmp=mFLSSSpar(L,v,target,ME,viaConjugate=1,maxCore=7,
randomizeTargetOrder = TRUE, tlimit = 5)})
# try finding at least one solution within 5 seconds with 7 CPUs

# exam the solution
if(length(tmp)!=0)abs(as.numeric(colSums(v[tmp[[1]],]))/target-1)


# ------------------------------------------------------------------------

# try the unfixed size subset sum solver

d=5L # number of dimensions

N=20L # superset size

# generate a multidimensional vector/set
v=as.data.frame(matrix(rnorm(d*N)*1000,ncol=d))

# make an underlying solution
```

```
solution=sample(1L:N, sample(1L:N, 1L))

# the subset sum target of the underlying solution
target=as.numeric(colSums(v[solution,]))

# bound the error as 5% of each dimension's target magnitude
ME=abs(target)*0.05

system.time({tmp=mFLSSSpar(0, v, target, ME, maxCore=7,
  randomizeTargetOrder = TRUE, tlimit = 5)})
tmp
solution
```

`mmFLknapsack` *General-purpose knapsack problem solver*

Description

An application of the multidimensional fixed size subset sum solver on multi-objective multidimensional bounded/unbounded knapsack problem.

Usage

```
mmFLknapsack(len, mV, lbound, ubound, viaConjugate = F, maxCore = 7L,
totalSolutionNeeded = 1L, tlimit = 60, singleTimeLimit = 10,
randomizeTargetOrder = sample(1L:(len * (nrow(mV) - len) + 1L),
len * (nrow(mV) - len) + 1L), LB = 1L:len,
UB = (nrow(mV) - len + 1L):nrow(mV))
```

Arguments

`len`	An integer as the subset size. If 0, activate the general-case (unfixed subset size) solver. Looping over different subset sizes to solve the general subset sum problem is preferred than setting len to 0 if mV is relatively large. Example below and example II method 2 in FLSSS help page both showed how the solver progresses for the unfixed subset size scenario with len=0.
`mV`	A data frame as the multidimensional vector. Each row is a multidimensional element
`lbound`	A numeric vector as the required lower bound of the subset sum
`ubound`	A numeric vector as the required upper bound of the subset sum
`viaConjugate`	Please refer to the details section for function "mFLSSSpar"
`maxCore`	Number of threads to invoke. Better not be greater than the maximal concurrentable threads in system
`totalSolutionNeeded`	Number of solutions needed.
`tlimit`	Time limit in seconds. Default to 60s

`singleTimeLimit`
: Please refer to details section or the package vignette for more information.

`randomizeTargetOrder`
: Discussed in the details section.

`LB`
: Lower bound initializer.

`UB`
: Upper bound initializer.

Details

In example

Value

A list of integer vectors. Each vector is a solution's indexes.

Examples

```
# PROBLEM DESCRIPTION:
# N uncorrelated assets in a portfolio
N=20
# initiate prices
# price=abs(rnorm(N)+1)*1e6 # following vector generated by this command
price=c(1305622.6,1080532.2,1245737.6,1182741.4,1148068.0,680656.4,1815259.4
,148957.1,496983.9,1509825.9,1728705.0,1067839.7,377404.0,1876344.7,1689563.4,
1569611.4,453573.7,1243951.8,2351058.0,1277857.0)

# asset still in portfolio in one week probability
# prob=runif(N)^0.2 # following vectors generated by this command
prob=c(0.9261989,0.9242555,0.5369701,0.9986413,0.9476367,0.9646240,0.9992410,
0.9576450,0.8063813,0.9448719,0.9154061,0.5568553,0.8291263,0.9012308,
0.8285262,0.8677975,0.9866419,0.9172621,0.9665702,0.7906170)

# one week return rates
# returnRate=runif(N)^3 # following vector generated by this command
returnRate=c(4.249990e-01,8.496273e-01,1.698966e-04,8.774191e-01,4.906818e-03
,1.176626e-04,6.790879e-01,7.480636e-03,1.048729e-01,5.841388e-01,4.857164e-02
,9.642615e-01,1.802129e-01,5.835278e-01,4.010622e-03,3.197286e-05,1.346220e-02
,2.310800e-01,1.785672e-01,7.112244e-01)

# one week expect values
expectVal=price*(1+returnRate)*prob

# requirement: select a portfolio (subset) from the assets such that the
# probability of all initial assets still in portfolio at the end of week no
# less than 50%; portfolio bid price no greater than 3.5e7; objective:
# maximize the expect value in one week


# ------------------------------------------------------------------------
# SOLVING:
```

```
# Assets are uncorrelated, so probability of all assets being in portfolio
# equals the product of the probabilities for each asset. Transform
# multiplication to summation via logarithmic.

# formulate multidimensional vector. Since no restriction on subset
# size, insert N dummy elements, set the subset size N
mV=data.frame(c(rep(0,N),price),c(rep(0,N),log2(prob)),c(rep(0,N),expectVal))
# this padding idea can be applied to solve the unbounded knapsack problem
# (each item can be used multiple times)

# formulate lbound ubound
lbound=c(-Inf, log2(0.5), sum(expectVal)*0.75)
# try if a portfolio with total expect value no less than 75% of the maximum
# can be found. Change "0.75" manually or in loop to detect the optimal

ubound=c(2e7, Inf, Inf)
# -Inf or Inf in bounding vectors can be replaced with NA

# singleTimeLimit and tlimit shall be changed for heavier task. The 5s is
# merely for passing R package publish requirement
rst=mmFLknapsack(N, mV, lbound, ubound, totalSolutionNeeded = 1, tlimit=5)

# exclude the dummy elements which also could lead to identical solutions
if(length(rst)>0L)rst=unique(lapply(rst, function(x)x[x>N]-N))

# exam the solutions
all(unlist(lapply(rst, function(x)
{
  c(sum(price[x])<=2e7, prod(prob[x])>=0.5)
})))


# ------------------------------------------------------------------------------
# EQUIVALENTLY:

rst=mmFLknapsack(0, data.frame(price, log2(prob), expectVal), lbound, ubound,
  totalSolutionNeeded = 1, tlimit=5)
# The 5s is merely for passing R package publish requirement

all(unlist(lapply(rst, function(x)
{
  c(sum(price[x])<=2e7, prod(prob[x])>=0.5)
})))
rst
```

Index