Package: nilde (via r-universe)

August 24, 2024

Encoding UTF-8

Version 1.1-7

Author Natalya Pya Arnqvist[aut, cre], Vassilly Voinov [aut], Rashid Makarov [aut], Yevgeniy Voinov [aut]

Maintainer Natalya Pya Arnqvist <nat.pya@gmail.com>

Title Nonnegative Integer Solutions of Linear Diophantine Equations with Applications

Date 2022-08-16

Description Routines for enumerating all existing nonnegative integer solutions of a linear Diophantine equation. The package provides routines for solving 0-1, bounded and unbounded knapsack problems; 0-1, bounded and unbounded subset sum problems; additive partitioning of natural numbers; and one-dimensional bin-packing problem.

Depends R (>= 2.15.0)

Imports methods, stats

Suggests parallel, lpSolve, TSP

License GPL (>= 2)

LazyLoad yes

NeedsCompilation no

Repository CRAN

Date/Publication 2022-08-16 10:10:02 UTC

Contents

`nilde-package` *Nonnegative Integer Solutions of Linear Diophantine Equations with Applications*

Description

`nilde` provides functions for enumerating all existing nonnegative integer solutions of a linear Diophantine equation. The package also includes functions for solving 0-1, bounded and unbounded knapsack problems; 0-1, bounded and unbounded subset sum problems; and a problem of additive partitioning of natural numbers. The algorithm is based on a generating function of Hardy and Littlewood used by Voinov and Nikulin (1997).

Author(s)

Natalya Pya Arnqvist[aut, cre], Vassilly Voinov [aut], Rashid Makarov [aut], Yevgeniy Voinov [aut]

Maintainer: Natalya Pya Arnqvist <nat.pya@gmail.com>

References

Voinov, V. and Nikulin, M. (1995) Generating functions, problems of additive number theory, and some statistical applications. Revue Roumaine de Mathématiques Pures et Appliquées, 40(2), 107-147

Voinov, V. and Nikulin, M. (1997) On a subset sum algorithm and its probabilistic and other applications. In: Advances in combinatorial methods and applications to probability and statistics, Ed. N. Balakrishnan, Birkhäuser, Boston, 153-163

Voinov, V. and Pya, N. (2006) A Remark on the Non-Uniqueness of a Non-Negative Integer Solution of a System of Linear Diophantine Equations with Applications to Integer Programming, Genetics, Reliability. Central Asian Journal of Management, Economics and Social Research (ISSN 1815-3356) 5(1-2), 42-47.

Voinov, V. and Pya, N. (2017) R-software for additive partitioning of positive integers. Mathematical Journal (ISSN 1682-0525), 17(1), 69-76

`bin.packing` *Enumeration of all existing solutions for one-dimensional bin-packing problem*

Description

The algorithm used for this function is a permutational modification of First Find (FF) algorithm described in Martello and Toth (1990). However, there are significant differences. First, the algorithm suggested by Martello and Toth does not set an objective to find all possible optimal solutions while algorithm suggested here finds them all. Apparently, these changes result in a significant increase of required computing time because we need to analyse and process all possible item permutations. Noteworthy, the time of the optimized algorithm is still polynomial. Second, Martello and Toth used an iterative embodiment of the algorithm while a recursive function is used here in order to reduce computing time by advantageous employment of "lazy evaluation" feature of R program and in order to optimize the code of the script.

According to its name, our algorithm is build around "generation of a bin". The objective is achieved by finding the next set of items that fit into existing bin (FF choice). The combination of added items is selected from the solutions provided by `nilde` function. This method allows to optimize computing time since we process more than one item at a time when we call the function. Further optimization is achieved by canceling any recursive calls when the addition of a new bin results in the number of bins exceeding the currently found local optimum (minimal number of bins achieved so far).

The algorithm segregates two types of calls from the parent function. Namely, scenarios when the current bin is complete and incomplete are treated separately. If the remaining unused capacity of the bin is zero, i.e. the bin is complete, then we check if creation of a new bin pushes the number of bins above the current optimum. If the number is still optimal, then we start a new bin and generate the list of item clusters that can fit and move on.

If the remaining unused capacity if not zero (bin is incomplete), then, first, we try to complete the existing bin by adding items that can fit and, then, if it is not possible, we close existing bin (even if its unused capacity is more than zero) and start processing item clusters that would not fit the closed bin anyway. By doing so, solutions with smaller number of bins will be generated earlier. Thus, we will be able to find globally optimal number of bins a.s.a.p.

Obviously, the algorithm stops recursive calls only in two cases: either we have distributed all the items or we exceeded the optimal number of bins by adding next new bin. In both cases, we proceed with processing the next possible combination of items, i.e. process the next leaf on our decision-making tree.

As for robustness checks, we have tested several versions of our recursive algorithm. For example, Martello and Toth demonstrate that on sample of any complexity First Find Decreasing (FFD) algorithm leads to a significant decrease in computing time as compared to FF. Therefore, we tested FFD modification of the algorithm. However, in our case, since we are looking for all optimal solutions, implementation of FFD algorithm has yielded no results. Furthermore, we experimented with the format and list of variables transferred recursively. Specifically, a version of the algorithm that transfers only logical vector of scenarios to be processed resulted in increase of computing time.

The function demonstrates the best computing time for all the sampled scenarios of item weights and bin capacity. However, there are some limitations to be addressed. For example, if the initial set includes multiple items with the same weight but different IDs, then the output of `GenVagonE` will need to be filtered from seemingly different solutions. Yet, the filtering is not computationally demanding and definitely polynomial in terms of time.

Note: majority of input variables are pre-computed in advance, separately, see example.

Usage

```
bin.packing(input.a, input.n, bin.globals)
```

Arguments

input.a	a vector of items weights.
input.n	capacity of a bin.
bin.globals	an environment for global variables.

Value

min.bins	minimum number of bins required.
solution	solutions of the bin-packing problem. Each number is a position of an item in the input string (Input string specifies item weights. Items are numbered 1, 2, 3, etc.) Items included into one bin are separated by commas. Bins are separated by space character. Different solutions are enclosed by double quotes.
bin.ineff	bin "inefficiency", i.e. unused space of each bin respectively, for every solution.
total.ineff	total "inefficiency", i.e. unused space of every solution (sum of bin "inefficiencies" per solution).

Author(s)

Rashid Makarov

References

Martello, S. and Toth, P. (1990) Knapsack Problems: Algorithms and Computer Implementations, Wiley, Chichester, 1990.

Voinov, V., Makarov, R., Voinov, Y. (2019) An exact polynomial in time solution of the one-dimensional bin-packing problem. In: Christos H Skiadas (ed.) Proceedings of the ASMDA 2019, published by ISAST (Int. Society for the Advancement of Science and Technology), December 2019, pp. 787-798.

See Also

nilde-package, get.partitions, get.subsetsum, nlde

Examples

```
library(nilde)
input.a <- c(70, 60, 50, 40, 30, 20, 10) # weights of items
input.n <- 100 # capacity of a bin
bin.globals <- new.env() # a new environment for global variables
bin.globals$OptVag <- length(input.a) # initial min # of bins
bin.globals$TrainList <- vector("list",length(input.a)) # output with solutions
g <- bin.packing(input.a, input.n,bin.globals)
g$min.bins # minimum number of bins
g$solution # solutions
```

`get.knapsack` *Enumeration of all existing nonnegative integer solutions for unbounded, bounded and 0-1 knapsack and subset sum problems*

Description

This function solves the unbounded, bounded and 0-1 knapsack problems.

The unbounded knapsack problem can be written as follows.

`maximize` $c_1 s_1 + c_2 s_2 + ... + c_l s_l$

`subject to` $a_1 s_1 + a_2 s_2 + ... + a_l s_l <= n,$

$s_i >= 0, integers.$

The bounded knapsack problem has additional constraints, $0 <= s_i <= b_i, i = 1, ..., l, b_i <= [n/a_i]$. The 0-1 knapsack problem arises when $s_i = 0$ or $1, i = 1, ..., l$.

The algorithm is based on a generating function of Hardy and Littlewood used by Voinov and Nikulin (1997). Subset sum problems are particular cases of knapsack problems when vectors of `weights`, $(a_1, ..., a_l)$, and `objectives`, $(c_1, ..., c_l)$, are equal.

Usage

```
get.knapsack(objective,a,n,problem="uknap",bounds=NULL)
```

Arguments

`objective`	A vector of coefficients (values of each item in the knapsack) of the objective function to be maximized when solving knapsack problem.
`a`	An `l`-vector of weights of each item in a knapsack, with `l>= 2`.
`n`	a maximal possible capacity of the knapsack.
`problem`	one of the following names of the problems to be solved: "uknap" (default) for an unbounded knapsack problem, "knap01" for a 0-1 knapsack problem, and "bknap" for a bounded knapsack problem.
`bounds`	An l-vector of positive integers, bounds of s_i, i.e. $0 <= s_i <= b_i$.

Value

`p.n`	total number of solutions obtained.
`solutions`	a matrix with each column representing a solution of `n`.

Author(s)

Vassilly Voinov, Natalya Pya Arnqvist, Yevgeniy Voinov

References

Voinov, V. and Nikulin, M. (1997) On a subset sum algorithm and its probabilistic and other applications. In: Advances in combinatorial methods and applications to probability and statistics, Ed. N. Balakrishnan, Birkhäuser, Boston, 153-163.

Hardy, G.H. and Littlewood, J.E. (1966) Collected Papers of G.H. Hardy, Including Joint Papers with J.E. Littlewood and Others. Clarendon Press, Oxford.

See Also

nilde-package, get.partitions, get.subsetsum, nlde

Examples

```
## some examples...
b1 <- get.knapsack(objective=c(200:206),a=c(100:106),n=999,problem="uknap")
b1

b2 <- get.knapsack(objective=c(41,34,21,20,8,7,7,4,3,3),a=c(41,34,21,20,8,7,7,4,3,3),
         n=50, problem="bknap", bounds=rep(2,10))
b2
colSums(b2$solutions*c(41,34,21,20,8,7,7,4,3,3))

b3 <- get.knapsack(objective=c(4,3,3),a=c(3,2,2),n=4,problem="bknap",bounds=c(2,2,2))
b3
## get maximum value of the objective function...
colSums(b3$solutions*c(4,3,3))
## checking constraint...
colSums(b3$solutions*c(3,2,2))

b4 <- get.knapsack(objective=c(4,3,3),a=c(3,2,2),n=4,problem="knap01")
b4
## get maximum value of the objective function...
colSums(b4$solutions*c(4,3,3))
## checking constraint...
colSums(b4$solutions*c(3,2,2))

## Not run:
b5 <- get.knapsack(a=c(100:106),n=2999,objective=c(200:206),problem="uknap")
b5$p.n ## total number of solutions
options(max.print=5E5)
print(b5)


## End(Not run)
```

---

get.partitions *Additive partitioning of natural numbers*

Description

This function solves the problem of additive partitioning of positive integers. The approach for additive partitioning is based on a generating function discussed in details in Voinov and Nikulin (1995). The function enumerates all partitions of a positive integer n on at most (or exactly) M parts, M <= n.

Usage

```
get.partitions(n, M, at.most=TRUE)
```

Arguments

n	A positive integer to be partitioned.
M	A positive integer, the number of parts of n, M <= n.
at.most	If TRUE then partitioning of n into at most M parts, if FALSE then partitioning on exactly M parts.

Value

p.n	total number of partitions obtained.
partitions	a matrix with each column presenting partitions of n.

Author(s)

Vassilly Voinov, Natalya Pya Arnqvist, Yevgeniy Voinov

References

Voinov, V. and Nikulin, M. (1995) Generating functions, problems of additive number theory, and some statistical applications. Revue Roumaine de Mathématiques Pures et Appliquées, 40(2), 107-147

Voinov, V.G. and Pya, N.E. (2017) R-software for additive partitioning of positive integers. Mathematical Journal (ISSN 1682-0525) 17(1), 69-76.

See Also

nilde-package, get.knapsack, get.subsetsum, nlde

Examples

```
## getting all partitions of n = 8 on at most 6 parts...
get.partitions(8,6,at.most=TRUE)

## getting all partitions of n = 8 on exactly 6 parts...
b <- get.partitions(8,6,at.most=FALSE)
b
colSums(b$partitions)
```

get.subsetsum *Enumeration of all existing 0-1 and bounded solutions of a subset sum problem*

Description

By default this function solves the following 0-1 subset sum problem. Given the set of positive integers (a_1, a_2, ..., a_l) and a positive integer n, find all non-empty subsets that sum to n, so that each of the integers a_i either appears in the subset or it does not, and the total number of summands should not exceed M, M <= n.

The bounded subset sum problem has restrictions on the number of times (bounds) a_i can turn up in the subset.

The algorithm is based on a generating function of Hardy and Littlewood used by Voinov and Nikulin (1997).

Usage

```
get.subsetsum(a,n,M=NULL,problem="subsetsum01",bounds=NULL)
```

Arguments

a	An l-vector of positive integers with l>= 2.
n	A positive integer.
M	A positive integer, the maximum number of summands, M <= n
problem	one of the two problems to be solved: "subsetsum01" (default) for a 0-1 subset sum problem, or "bsubsetsum" a bounded subset sum problem.
bounds	An l-vector of positive integers, bounds for s_i, i.e. $0 <= s_i <= b_i$

Value

p.n	total number of solutions obtained.
solutions	a matrix with each column presenting a solution for n.

Author(s)

Vassilly Voinov, Natalya Pya Arnqvist, Yevgeniy Voinov

References

Voinov, V. and Nikulin, M. (1997) On a subset sum algorithm and its probabilistic and other applications. In: Advances in combinatorial methods and applications to probability and statistics, Ed. N. Balakrishnan, Birkhäuser, Boston, 153-163.

Hardy, G.H. and Littlewood, J.E. (1966) Collected Papers of G.H. Hardy, Including Joint Papers with J.E. Littlewood and Others. Clarendon Press, Oxford.

See Also

nilde-package, get.partitions, get.knapsack, nlde

Examples

```
## some examples...
b1 <- get.subsetsum(a=c(41,34,21,20,8,7,7,4,3,3),M=10,n=50,problem="subsetsum01")
b1
colSums(b1$solutions*c(41,34,21,20,8,7,7,4,3,3))

b2 <- get.subsetsum(a=c(111:120),M=10,n=485,problem="subsetsum01") ## no solutions
b2

b3 <- get.subsetsum(a=c(30,29,32,31,33),M=5,n=91,problem="subsetsum01")
b3
colSums(b3$solutions*c(30,29,32,31,33))
get.subsetsum(a=c(30,29,32,31,33),M=5,n=91,problem="bsubsetsum",bounds=c(1,1,1,1,1))

b4 <- get.subsetsum(a=c(30,29,32,31,33),M=5,n=91,problem="bsubsetsum",
                bounds=c(1,2,1,3,4))
b4
colSums(b4$solutions*c(30,29,32,31,33))
```

---

nlde — *Enumeration of all existing nonnegative integer solutions of a linear Diophantine equation*

Description

This function enumerates nonnegative integer solutions of a linear Diophantine equation (NLDE):

$$a_1 s_1 + a_2 s_2 + ... + a_l s_l = n,$$

where $a_1 <= a_2 <= ... <= a_l$, $a_i > 0$, $n > 0$, $s_i >= 0$, $i = 1, 2, ..., l$, and all variables involved are integers.

The algorithm is based on a generating function of Hardy and Littlewood used by Voinov and Nikulin (1997).

Usage

```
nlde(a, n, M=NULL, at.most=TRUE, option=0)
```

Arguments

a	An l-vector of positive integers (coefficients of the left-hand-side of NLDE) with l>= 2.
n	A positive integer which is to be partitioned.

M	A positive integer, the number of parts of n, M <= n.
at.most	If TRUE partitioning of n into at most M parts, if FALSE partitioning on exactly M parts.
option	When set to 1 (or any positive number) finds only 0-1 solutions of the linear Diophantine equation. When set to 2 (or any positive number > 1) finds 0-1 solutions of the linear Diophantine inequality.

Value

p.n	total number of partitions obtained.
solutions	a matrix with each column forming a partition of n.

Author(s)

Vassilly Voinov, Natalya Pya Arnqvist, Yevgeniy Voinov

References

Voinov, V. and Nikulin, M. (1997) On a subset sum algorithm and its probabilistic and other applications. In: Advances in combinatorial methods and applications to probability and statistics, Ed. N. Balakrishnan, Birkhäuser, Boston, 153-163.

Hardy, G.H. and Littlewood, J.E. (1966) Collected Papers of G.H. Hardy, Including Joint Papers with J.E. Littlewood and Others. Clarendon Press, Oxford.

See Also

nilde-package, get.partitions, get.subsetsum, get.knapsack

Examples

```
## some examples...
## example 1...
nlde(a=c(3,2,5,16),n=18,at.most=TRUE)
b1 <- nlde(a=c(3,2,5,16),n=18,M=6,at.most=FALSE)
b1
## checking M, the number of parts that n=18 has been partitioned into...
colSums(b1$solutions)
## checking the value of n...
colSums(b1$solutions*c(3,2,5,16))

## example 2:  solving 0-1 nlde ...
b2 <- nlde(a=c(3,2,5,16),n=18,M=6,option=1)
b2
colSums(b2$solutions*c(3,2,5,16))

## example 3...
b3 <- nlde(c(15,21),261)
b3
## checking M, the number of parts that n has been partitioned into...
colSums(b3$solutions)
```

```
## checking the value of n...
colSums(b3$solutions*c(15,21))

## example 4...
nlde(c(5,6),19) ## no solutions

## example 5: solving 0-1 inequality...
b4 <- nlde(a=c(70,60,50,33,33,33,11,7,3),n=100,at.most=TRUE,option=2)
```

`print.partitions` *Print partitions object.*

Description

The default print method for a `partitions`, `nlde` objects.

Usage

```
## S3 method for class 'partitions'
print(x, ...)
## S3 method for class 'nlde'
print(x, ...)
## S3 method for class 'knapsack'
print(x, ...)
## S3 method for class 'subsetsum'
print(x, ...)
## S3 method for class 'tsp_solver'
print(x, ...)
```

Arguments

`x, ...`	objects of class `partitions`, `nlde`, `knapsack`, `subsetsum` as produced by `get.partitions()`, `nlde()`, `get.knapsack()`, `get.subsetsum()`, `tsp_solver()` correspondingly.

Details

Prints the number of partitions/solutions obtained and all resulted partitions/solutions themselves.

Author(s)

Natalya Pya Arnqvist <nat.pya@gmail.com>

`tsp_solver` *Travelling salesperson problem solver*

Description

Interface to travelling salesperson problem solver.

Consider an integer linear programming (ILP) formulation of DFJ (Dantzig et al, 1954) used in this research. Let $G = (V, A)$ be a graph with a set V of n vertices and A be a set of arcs or edges. Let $C = (c_{ij})$ be a distance (or cost) matrix associated with A. Elements of the distance matrix C, c_{ij}, are positive integers, $i, j \in V, i \neq j$. The TSP focuses on finding a minimum distance circuit (a tour or Hamiltonian circuit) that passes through each vertex once and only once. The DFJ formulation is

$$\texttt{minimize } L = \sum_{j \neq i} c_{ij}\delta_{ij} \qquad (1)$$

$$\texttt{subject to } \sum_{j=1}^{n} \delta_{ij} = 1, i = 1, ..., n \qquad (2)$$

$$\sum_{i=1}^{n} \delta_{ij} = 1, j = 1, ..., n \qquad (3)$$

$$\sum_{i,j \in S} \delta_{ij} \leq |S| - 1, S \subset V, 2 \leq |S| \leq n - 2 \qquad (4)$$

$$\delta_{ij} \in 0, 1, i, j = 1, ..., n, i \neq j \qquad (5)$$

Constraints (2) and (3) are known as degree constraints indicating that every vertex should be entered and left exactly once correspondingly. Constraints (4) are subtour elimination constraints that prevent from forming subtours (several unconnected tours on subsets of less than n vertices), with $|S|$ denoting the number of vertices in S.

In the DFJ formulation there are $n(n-1)$ unknown binary variables, $2n$ degree constraints and $2^n - 2n - 2$ subtour elimination constraints. Since the number of subtour elimination constraints increases exponentially, solving this problem directly using an integer linear programming code is in general intractable. However, relaxed versions of the integer linear programming problem where some constraints are initially removed, and later restored via an iterative process, have been proposed and extensively used.

Here it is proposed to combine heuristics (to get an initial feasible solution) and a linear Diophantine equation (nilde) relaxation to develop a new exact algorithm that constructs all existing optimal solutions for the TSP in an efficient way.

Below is a brief summary of the proposed algorithm.

Step 1. (Initialization) Solve a corresponding assignment problem to obtain an initial lower bound on the value of the optimal TSP solution. Apply heuristics to obtain an initial upper bound.

Step 2. (Subproblem solution) Given the initial lower bound construct all 0-1 solutions to a linear Diophantine equation introduced by Voinov and Nikulin (1997).

Step 3. (Degree constraints check) Remove solutions that do not satisfy the degree constraints.

Step 4. (Subtour elimination) Remove solutions that contain subtours by applying a new simple subtour elimination routine. If there is a solution(s) that contains no subtours, it forms the optimal

solution(s): stop. Otherwise, increase the initial lower bound by one and go to step 2. Repeat until the upper bound is reached.

The integer programming formulation of the assignment problem solved in Step 1 of the above algorithm is obtained by relaxing constraints (4), i.e. given by (1) subject to constraints (2), (3) and (5).

For implementing Step 2, solutions of the corresponding subset sum problem should be enumerated. A subset sum problem formulation can be expressed as

$$a_1 s_1 + a_2 s_2 + \ldots + a_p s_p = L, \qquad (6)$$

where $s_i \in \{0, 1\}$, $i = 1, \ldots, p, p = n(n-1)$ is the number of unknown binary variables of the original TSP. a_i are positive integers matching the costs c_{ij} of the cost matrix C.

Voinon and Nikulin (1997) introduced an algorithm that enumerates all nonnegative integer solutions of equation (6) by using the corresponding generating function and the binomial theorem. All $0-1$ solutions to the equation in (6) can be found by means of the following generating function:

$$\Psi_L(z) = (z^{a_1} + z^{a_2} + \ldots + z^{a_p})^L = \sum_{k=L\cdot\min_i(a_i)}^{k=L\cdot\max_i(a_i)} R_k(L,p),$$

where

$$R_k(L,p) = \sum_{s_p=0}^{\min\left(1,\left[\frac{L}{a_p}\right]\right)} \sum_{s_{p-1}=0}^{\min\left(1,\left[\frac{L-a_p s_p}{a_{p-1}}\right]\right)} \cdots \sum_{s_2=0}^{\min\left(1,\left[\frac{L-a_p s_p-\ldots-a_3 s_3}{a_2}\right]\right)} \frac{L!}{(L-s_1-\ldots-s_p)!s_1!\ldots s_p!}, \qquad (7)$$

$s_1 = \frac{L-a_p s_p-\ldots-a_2 s_2}{a_1}$ is necessarily either 0 or 1. Otherwise, the equation in (6) does not have any solutions. The notation $[x]$ denotes the greatest integer part of x. The right-hand side multiplier in (7) presents the total number of compositions that satisfy the above condition. If the value of that multiplier is set to 1, (7) gives the number of $0-1$ solutions for the equation (6). The solutions, if exist, are written explicitly as

$$\left\{a_1^{s_1}, a_2^{s_2}, \ldots, a_p^{s_p}\right\}, \qquad (8)$$

where $\{s_2, \ldots, s_p\}$ are sets of summation indices in (7), with s_1 as specified above. The notation (8) means that in a particular partition (a solution of the equation (6)) there are s_1 terms equal to a_1, s_2 terms of a_2 and so on.

Usage

```
tsp_solver(data, labels=NULL,cluster=0,upper_bound=NULL,
        lower_bound=NULL,method="cheapest_insertion",no_go=NULL)
```

Arguments

`data`	An `n` x `n` matrix of costs/distances of the TSP (with `0`'s or `NA`s on the main diagonal). Costs/distances of the unconnected edges must be supplied as NA.
`labels`	An `n` vector of optional city labels. If not given, labels are taken from `data`.

cluster	Degree constraints can be checked in parallel using parLapply from the parallel package. cluster is either 0 (default) for no parallel computing to be used; or 1 for one less than the number of cores; or user-supplied cluster on which to do checking. a cluster here can be some cores of a single machine.
upper_bound	A positive integer, an upper bound of the tour length (cost function), if not supplied (default: NULL) heuristic solution is obtained using TSP::solve_TSP(data,method).
lower_bound	A positive integer, a lower possible value of the tour lenght (cost function); if not supplied (default: NULL), obtained by solving a corresponding assignment problem using lpSolve::lp.assign(data)
method	Heuristic method used in TSP::solve_TSP() (default: cheapest_insertion)
no_go	A suitably large value used in the distance/cost matrix to make related edges infeasible, if NULL (default) set to max(data)*10^5. It can be set to Inf for TSP(). However, lpSolve() is very sensitive to too large values and can result in high values of the lower_bound.

Value

tour	optimal tour(s).
tour_length	an optimal (minimal) length of the obtained tour(s).
coming_solutions	a list of coming feasible tours obtained within [lower_bound, upper_bound].
coming_tour_lengths	a vector of feasible tour length gone within [lower_bound, upper_bound].
iter	a number of feasible tour length gone through
upper_bound	an upper bound of the tour length
lower_bound	a lower bound value of the tour lenght

Author(s)

Vassilly Voinov, Natalya Pya Arnqvist

References

Voinov, V. and Nikulin, M. (1997) On a subset sum algorithm and its probabilistic and other applications. In: Advances in combinatorial methods and applications to probability and statistics, Ed. N. Balakrishnan, Birkhäuser, Boston, 153-163.

Dantzig, G., Fulkerson, R. and Johnson, S. (1954) Solution of a large-scale traveling-salesman problem. Journal of the operations research society of America , 2(4):393-410.

See Also

nilde-package, get.partitions, get.knapsack, get.subsetsum

Examples

```
## Not run:
## some examples...
library(nilde)
set.seed(3)
d <- matrix(sample(1:100,25,replace=TRUE),5,5)
diag(d) <-NA # although no need to specify as the code assumes NAs by default
g <- tsp_solver(d)
g

## End(Not run)
```

Index