Package 'TSP'

January 23, 2020

Type Package

Title Traveling Salesperson Problem (TSP)

Version 1.1-8

Date 2020-01-23

Description Basic infrastructure and some algorithms for the traveling salesperson problem (also traveling salesman problem; TSP). The package provides some simple algorithms and an interface to the Concorde TSP solver and its implementation of the Chained-Lin-Kernighan heuristic. The code for Concorde itself is not included in the package and has to be obtained separately.

Classification/ACM G.1.6, G.2.1, G.4

URL https://github.com/mhahsler/TSP

BugReports https://github.com/mhahsler/TSP/issues

Depends R (>= 2.14.0)

Imports graphics, foreach, utils, stats, grDevices

Suggests sp, maps, maptools, testthat

License GPL-3

NeedsCompilation yes

Author Michael Hahsler [aut, cre, cph], Kurt Hornik [aut, cph]

Maintainer Michael Hahsler <mhahsler@lyle.smu.edu>

Repository CRAN

Date/Publication 2020-01-23 17:30:10 UTC

R topics documented:

ATSP *Class ATSP – Asymmetric traveling salesperson problem*

Description

Constructor to create an instance of the asymmetric traveling salesperson problem (ATSP) and some auxiliary methods.

Usage

```
## constructor
ATSP(x, labels = NULL, method = NULL)

## coercion
as.ATSP(x)

## methods
## S3 method for class 'ATSP'
n_of_cities(x)
## S3 method for class 'ATSP'
image(x, order, col = gray.colors(64), ...)
## S3 method for class 'ATSP'
labels(object, ...)
## S3 method for class 'ATSP'
print(x, ...)
```

Arguments

x, object	an object (a square matrix) to be converted into an ATSP or, for the methods, an object of class ATSP.
labels	optional city labels. If not given, labels are taken from x.
method	optional name of the distance metric.
col	color scheme for image.
order	order of cities as an integer vector or an object of class TOUR.
...	further arguments are passed on.

Details

Objects of class ATSP are internally represented by a matrix (use as.matrix() to get just the matrix).

ATSPs can be transformed into (larger) symmetric TSPs using reformulate_ATSP_as_TSP().

Value

ATSP() returns x as an object of class ATSP.

n_of_cities() returns the number of cities in x.

labels() returns a vector with the names of the cities in x.

Author(s)

Michael Hahsler

See Also

TOUR, insert_dummy, tour_length, reformulate_ATSP_as_TSP, solve_TSP.

Examples

```
data <- matrix(runif(10^2), ncol = 10, dimnames = list(1:10, 1:10))

atsp <- ATSP(data)
atsp

## use some methods
n_of_cities(atsp)
labels(atsp)

## calculate a tour
tour <- solve_TSP(atsp, method = "nn")
tour

tour_length(tour)

image(atsp, tour)
```

Concorde *Using the Concorde TSP Solver*

Description

The Concorde TSP Solver package contains several solvers. Currently, interfaces to the Concorde solver (Applegate et al. 2001), one of the most advanced and fastest TSP solvers using branch-and-cut, and the Chained Lin-Kernighan (Applegate et al. 2003) implementation are provided in **TSP**. Concorde can solve TSPs and ETSPs directly. ATSPs are reformulated as larger TSP's and then solved.

The Concorde TSP Solver package is freely available for academic research and has to be obtained separately from the Concorde web site (see details).

Usage

```
## set path for executables
concorde_path(path)

## obtain a list of command line options for the solvers
concorde_help()
linkern_help()
```

Arguments

`path`	a character string with the path to the directory where the executables are installed.

Details

The code of the Concorde TSP package is not included in this package and has to be obtained separately from the Concorde web site (see references). Either download the precompiled executables and place them in a suitable directory and make them executable, or you can get the source code and compile it on your own. **TSP** needs to know where the executables are. There are two options: (1) use `concorde_path()` to set the path to the directory containing the executables for concorde and linkern, or (2) make sure that the executables are in the search path stored in the `PATH` environment variable (see `link{Sys.setenv}`).

`solve_TSP` uses `write_TSPLIB` to write the TSP for Concorde and tries to find the appropriate `precision` value (digits after the decimal point) to convert the provided distances into the needed integer value range. The `precision` value can also be specified in `control` in `solve_TSP` with method Concorde. Warning messages will alert the user if the conversion to integer values results into rounding errors that are worse then what is specified in the `precision` control parameter.

To get a list of all available command line options which can be used via the `clo` option for `solve_TSP` use `concorde_help()` and `linkern_help()`. Several options ('`-x`', '`-o`', '`-N`', '`-Q`') are not available via `solve_TSP` since they are used by the interface.

Author(s)

Michael Hahsler

References

Concorde home page, http://www.math.uwaterloo.ca/tsp/concorde/

Concorde download page, http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm

David Appletgate, Robert Bixby, Vasek Chvatal, William Cook (2001): TSP cuts which do not conform to the template paradigm, Computational Combinatorial Optimization, M. Junger and D. Naddef (editors), Springer-Verlag.

David Applegate and William Cook and Andre Rohe (2003): Chained Lin-Kernighan for Large Traveling Salesman Problems, *INFORMS Journal on Computing*, **15**, 82–92.

See Also

solve_TSP, write_TSPLIB

Examples

```
## Not run:
## see if Concorde is correctly installed
concorde_path()


## set path to the Concorde executible if it is not in the search PATH
## Example:
## concorde_path("~/concorde/")

concorde_help()

data("USCA312")

## run concorde only with fast cuts (-V)
solve_TSP(USCA312, method = "concorde", control = list(clo = "-V"))

## End(Not run)
```

---

cut_tour *Cut a tour to form a path*

Description

Cuts a tour at a specified city to form a path.

Usage

```
cut_tour(x, cut, exclude_cut = TRUE)
```

Arguments

x	an object of class TOUR.
cut	the index or label of the city/cities to cut the tour.
exclude_cut	exclude the city where we cut? If FALSE, the city at the cut is included in the path as the first city.

Value

Retuns a named vector with city ids forming the path. If multiple cuts are used then a list with paths is returned.

Author(s)

Michael Hahsler

See Also

TOUR.

Examples

```
data("USCA50")

## find a path starting at Austin, TX
tour <- solve_TSP(USCA50)
path <- cut_tour(tour, cut = "Austin, TX", exclude_cut = FALSE)
path

## cut the tours at two cities
tour <- solve_TSP(USCA50)
path <- cut_tour(tour, cut = c("Austin, TX", "Cambridge, MA"), exclude_cut = FALSE)
path

## cut a tour at the largest gap using a dummy city
tsp <- insert_dummy(USCA50, label = "cut")
tour <- solve_TSP(tsp)

## cut tour into path at the dummy city
path <- cut_tour(tour, "cut")
path
```

ETSP *Class ETSP – Euclidean traveling salesperson problem*

Description

Constructor to create an instance of a Euclidean traveling salesperson problem (TSP) represented by city coordinates and some auxiliary methods.

Usage

```
## constructor
ETSP(x, labels = NULL)

## coercion
as.ETSP(x)

## methods
## S3 method for class 'ETSP'
n_of_cities(x)
## S3 method for class 'ETSP'
image(x, order, col = gray.colors(64), ...)
## S3 method for class 'ETSP'
plot(x, y = NULL, tour = NULL, tour_lty = 2, tour_col = 1, ...)
## S3 method for class 'ETSP'
labels(object, ...)
## S3 method for class 'ETSP'
print(x, ...)
```

Arguments

`x, object`	an object (data.frame or matrix) to be converted into a `ETSP` or, for the methods, an object of class `ETSP`.
`labels`	optional city labels. If not given, labels are taken from `x`.
`col`	color scheme for image.
`order`	order of cities for the image as an integer vector or an object of class `TOUR`.
`tour, y`	a tour to be visualized.
`tour_lty, tour_col`	line type and color for tour.
`...`	further arguments are passed on.

Details

Objects of class `ETSP` are internally represented as a `matrix` objects (use `as.matrix()` to get the `matrix` object).

Value

`ETSP()` returns `x` as an object of class `ETSP`.

`n_of_cities()` returns the number of cities in `x`.

`labels()` returns a vector with the names of the cities in `x`.

Author(s)

Michael Hahsler

See Also

TOUR, insert_dummy, tour_length, solve_TSP.

Examples

```
x <- data.frame(x = runif(20), y = runif(20), row.names = LETTERS[1:20])

## create a TSP
etsp <- ETSP(x)
etsp

## use some methods
n_of_cities(etsp)
labels(etsp)

## plot ETSP and solution
tour <- solve_TSP(etsp)
tour

plot(etsp, tour, tour_col = "red")
```

---

insert_dummy *Insert dummy cities into a distance matrix*

Description

Inserts dummy cities into objects of class TSP or ATSP. A dummy city has the same, constant distance (0) to all other cities and is infinitely far from other dummy cities. A dummy city can be used to transform a shortest Hamiltonian path problem (i.e., finding an optimal linear order) into a shortest Hamiltonian cycle problem which can be solved by a TSP solvers (Garfinkel 1985).

Several dummy cities can be used together with a TSP solvers to perform rearrangement clustering (Climer and Zhang 2006).

Usage

```
insert_dummy(x, n = 1, const = 0, inf = Inf, label =  "dummy")
```

Arguments

x	an object of class TSP or ATSP.
n	number of dummy cities.
const	distance of the dummy cities to all other cities.
inf	distance between dummy cities.
label	labels for the dummy cities. If only one label is given, it is reused for all dummy cities.

Details

The dummy cities are inserted after the other cities in `x`.

A `const` of 0 is guaranteed to work if the TSP finds the optimal solution. For heuristics returning suboptimal solutions, a higher `const` (e.g., `2 * max{x}`) might provide better results.

Author(s)

Michael Hahsler

References

Sharlee Climer, Weixiong Zhang (2006): Rearrangement Clustering: Pitfalls, Remedies, and Applications, *Journal of Machine Learning Research* **7**(Jun), pp. 919–943.

R.S. Garfinkel (1985): Motivation and modelling (chapter 2). In: E. L. Lawler, J. K. Lenstra, A.H.G. Rinnooy Kan, D. B. Shmoys (eds.) The traveling salesman problem - A guided tour of combinatorial optimization, Wiley \& Sons.

See Also

TSP, ATSP

Examples

```
## Example 1: Find a short Hamiltonian path
set.seed(1000)
x <- data.frame(x = runif(20), y = runif(20), row.names = LETTERS[1:20])

tsp <- TSP(dist(x))

## add a dummy city to cut the tour into a path
tsp <- insert_dummy(tsp, label = "cut")
tour <- solve_TSP(tsp)
tour

plot(x)
lines(x[cut_tour(tour, cut = "cut"),])


## Example 2: Rearrangement clustering of the iris dataset
set.seed(1000)
data("iris")
tsp <- TSP(dist(iris[-5]))

## insert 2 dummy cities to creates 2 clusters
tsp_dummy <- insert_dummy(tsp, n = 3, label = "boundary")

## get a solution for the TSP
tour <- solve_TSP(tsp_dummy)

## plot the reordered distance matrix with the dummy cities as lines separating
```

```
## the clusters
image(tsp_dummy, tour)
abline(h = which(labels(tour)=="boundary"), col = "red")
abline(v = which(labels(tour)=="boundary"), col = "red")

## plot the original data with paths connecting the points in each cluster
plot(iris[,c(2,3)], col = iris[,5])
paths <- cut_tour(tour, cut = "boundary")
for(p in paths) lines(iris[p, c(2,3)])

## Note: The clustering is not perfect!
```

`reformulate_ATSP_as_TSP` *Reformulate a ATSP as a symmetric TSP*

Description

A ATSP can be formulated as a symmetric TSP by doubling the number of cities (Jonker and Volgenant 1983). The solution of the TSP also represents the solution of the original ATSP.

Usage

```
reformulate_ATSP_as_TSP(x, infeasible = Inf, cheap = -Inf)
```

Arguments

`x`	an ATSP.
`infeasible`	value for infeasible connections.
`cheap`	value for distance between a city and its corresponding dummy city.

Details

To reformulate the ATSP as a TSP, for each city a dummy city (e.g, for 'New York' a dummy city 'New York*') is added. Between each city and its corresponding dummy city a negative or very small distance with value `cheap` is used. This makes sure that each cities always occurs in the solution together with its dummy city. The original distances are used between the cities and the dummy cities, where each city is responsible for the distance going to the city and the dummy city is responsible for the distance coming from the city. The distances between all cities and the distances between all dummy cities are set to `infeasible`, a very large value which makes the infeasible.

Value

a TSP object.

Author(s)

Michael Hahsler

References

Jonker, R. and Volgenant, T. (1983): Transforming asymmetric into symmetric traveling salesman problems, *Operations Research Letters, 2, 161–163.*

See Also

ATSP, TSP.

Examples

```
data("USCA50")

## set the distances towards Austin to zero which makes it a ATSP
austin <- which(labels(USCA50) == "Austin, TX")
atsp <- as.ATSP(USCA50)
atsp[, austin] <- 0

## reformulate as a TSP
tsp <- reformulate_ATSP_as_TSP(atsp)
labels(tsp)

## create tour (now you could use Concorde or LK)
tour_atsp <- solve_TSP(tsp, method="nn")
head(labels(tour_atsp), n = 10)
tour_atsp
## Note that the tour has a lenght of -Inf since the reformulation created
## some -Inf distances

## filter out the dummy cities (we specify tsp so the tour lenght is
## recalculated)
tour <- TOUR(tour_atsp[tour_atsp <= n_of_cities(atsp)], tsp = atsp)
tour
```

---

solve_TSP *TSP solver interface*

Description

Common interface to all TSP solvers in this package.

Usage

```
solve_TSP(x, method = NULL, control = NULL, ...)

## S3 method for class 'TSP'
solve_TSP(x, method = NULL, control = NULL, ...)
## S3 method for class 'ETSP'
solve_TSP(x, method = NULL, control = NULL, ...)
## S3 method for class 'ATSP'
solve_TSP(x, method = NULL, control = NULL, as_TSP = FALSE, ...)
```

Arguments

x	the TSP given as an object of class TSP, ATSP or ETSP.
method	method to solve the TSP (default: arbitrary insertion algorithm with two_opt refinement.
control	a list of arguments passed on to the TSP solver selected by method.
as_TSP	should the ATSP reformulated as a TSP for the solver?
...	additional arguments are added to control.

Details

Treatment of NAs and infinite values in x: TSP and ATSP contain distances and NAs are not allowed. Inf is allowed and can be used to model the missing edges in incomplete graphs (i.e., the distance between the two objects is infinite). Internally, Inf is replaced by a large value given by $max(x) + 2range(x)$. Note that the solution might still place the two objects next to each other (e.g., if x contains several unconnected subgraphs) which results in a path length of Inf.

All heuristics can be used with the control arguments repetitions (uses the best from that many repetitions with random starts) and two_opt (a logical indicating if two_opt refinement should be performed). If several repetitions are done (this includes method "repetitive_nn") then **foreach** is used so they can be performed in parallel on multiple cores/machines. To enable parallel execution an appropriate parallel backend needs to be registered (e.g., load **doParallel** and register it with registerDoParallel()).

ETSP are currently solved by first calculating a dissimilarity matrix (a TSP). Only concorde and linkern can solve the TSP directly on the ETSP.

Some solvers (including Concorde) cannot directly solve ATSP directly. ATSP can be reformulated as larger TSP and solved this way. For convenience, solve_TSP() has an extra argument as_TSP which can be set to TRUE to automatically solve the ATSP reformulated as a TSP (see reformulate_ATSP_as_TSP).

Currently the following methods are available:

"identity", "random" return a tour representing the order in the data (identity order) or a random order.

"nearest_insertion", "farthest_insertion", "cheapest_insertion", "arbitrary_insertion"
Nearest, farthest, cheapest and arbitrary insertion algorithms for a symmetric and asymmetric TSP (Rosenkrantz et al. 1977).

The distances between cities are stored in a distance matrix D with elements $d(i, j)$. All insertion algorithms start with a tour consisting of an arbitrary city and choose in each step a city k not yet on the tour. This city is inserted into the existing tour between two consecutive cities i and j, such that

$$d(i, k) + d(k, j) - d(i, j)$$

is minimized. The algorithms stops when all cities are on the tour.

The nearest insertion algorithm chooses city k in each step as the city which is *nearest* to a city on the tour.

For farthest insertion, the city k is chosen in each step as the city which is *farthest* to any city on the tour.

Cheapest insertion chooses the city k such that the cost of inserting the new city (i.e., the increase in the tour's length) is minimal.

Arbitrary insertion chooses the city k randomly from all cities not yet on the tour.

Nearest and cheapest insertion tries to build the tour using cities which fit well into the partial tour constructed so far. The idea behind behind farthest insertion is to link cities far away into the tour fist to establish an outline of the whole tour early.

Additional control options:

`start` index of the first city (default: random city).

`"nn"`, `"repetitive_nn"` Nearest neighbor and repetitive nearest neighbor algorithms for symmetric and asymmetric TSPs (Rosenkrantz et al. 1977).

The algorithm starts with a tour containing a random city. Then the algorithm always adds to the last city on the tour the nearest not yet visited city. The algorithm stops when all cities are on the tour.

Repetitive nearest neighbor constructs a nearest neighbor tour for each city as the starting point and returns the shortest tour found.

Additional control options:

`start` index of the first city (default: random city).

`"two_opt"` Two edge exchange improvement procedure (Croes 1958).

This is a tour refinement procedure which systematically exchanges two edges in the graph represented by the distance matrix till no improvements are possible. Exchanging two edges is equal to reversing part of the tour. The resulting tour is called *2-optimal.*

This method can be applied to tours created by other methods or used as its own method. In this case improvement starts with a random tour.

Additional control options:

`tour` an existing tour which should be improved. If no tour is given, a random tour is used.

`two_opt_repetitions` number of times to try two_opt with a different initial random tour (default: 1).

`"concorde"` Concorde algorithm (Applegate et al. 2001).

Concorde is an advanced exact TSP solver for *only symmetric* TSPs based on branch-and-cut. The program is not included in this package and has to be obtained and installed separately (see `Concorde`).

Additional control options:

`exe` a character string containing the path to the executable (see `Concorde`).

`clo` a character string containing command line options for Concorde, e.g., `control = list(clo = "-B -v")`. See `concorde_help` on how to obtain a complete list of available command line options.

`precision` an integer which controls the number of decimal places used for the internal representation of distances in Concorde. The values given in `x` are multiplied by $10^{precision}$ before being passed on to Concorde. Note that therefore the results produced by Concorde (especially lower and upper bounds) need to be divided by $10^{precision}$ (i.e., the decimal point has to be shifted `precision` placed to the left). The interface to Concorde uses `write_TSPLIB` (see there for more information).

`"linkern"` Concorde's Chained Lin-Kernighan heuristic (Applegate et al. 2003).

The Lin-Kernighan (Lin and Kernighan 1973) heuristic uses variable k edge exchanges to improve an initial tour. The program is not included in this package and has to be obtained and installed separately (see Concorde).

Additional control options: see Concorde above.

Value

An object of class TOUR.

Author(s)

Michael Hahsler

References

David Appletgate, Robert Bixby, Vasek Chvatal, William Cook (2001): TSP cuts which do not conform to the template paradigm, Computational Combinatorial Optimization, M. Junger and D. Naddef (editors), Springer.

D. Applegate, W. Cook and A. Rohe (2003): Chained Lin-Kernighan for Large Traveling Salesman Problems. *INFORMS Journal on Computing, 15(1):82–92.*

G.A. Croes (1958): A method for solving traveling-salesman problems. *Operations Research, 6(6):791–812.*

S. Lin and B. Kernighan (1973): An effective heuristic algorithm for the traveling-salesman problem. *Operations Research, 21(2): 498–516.*

D.J. Rosenkrantz, R. E. Stearns, and Philip M. Lewis II (1977): An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing, 6(3):563–581.*

See Also

TOUR, TSP, ATSP, write_TSPLIB, Concorde.

Examples

```
## solve a simple Euclidean TSP (using the default method)
etsp <- ETSP(data.frame(x = runif(20), y = runif(20)))
tour <- solve_TSP(etsp)
tour
tour_length(tour)
plot(etsp, tour)


## compare methods
data("USCA50")
USCA50
methods <- c("identity", "random", "nearest_insertion",
  "cheapest_insertion", "farthest_insertion", "arbitrary_insertion",
  "nn", "repetitive_nn", "two_opt")

## calculate tours
tours <- lapply(methods, FUN = function(m) solve_TSP(USCA50, method = m))
```

```
names(tours) <- methods

## use the external solver which has to be installed separately
## Not run:
tours$concorde  <- solve_TSP(USCA50, method = "concorde")
tours$linkern  <- solve_TSP(USCA50, method = "linkern")

## End(Not run)

## register a parallel backend to perform repetitions in parallel
## Not run:
library(doParallel)
registerDoParallel()

## End(Not run)

## add some tours using repetition and two_opt refinements
tours$'nn+two_opt' <- solve_TSP(USCA50, method="nn", two_opt=TRUE)
tours$'nn+rep_10' <- solve_TSP(USCA50, method="nn", rep=10)
tours$'nn+two_opt+rep_10' <- solve_TSP(USCA50, method="nn", two_opt=TRUE, rep=10)
tours$'arbitrary_insertion+two_opt' <- solve_TSP(USCA50)

## show first tour
tours[[1]]

## compare tour lengths
opt <- 14497 # obtained by Concorde
tour_lengths <- c(sort(sapply(tours, tour_length), decreasing = TRUE),
  optimal = opt)
dotchart(tour_lengths/opt*100-100, xlab = "percent excess over optimum")
```

TOUR *Class TOUR – Solution to a traveling salesperson problem*

Description

Class to store the solution of a TSP. Objects of this class are returned by TSP solvers in this package. Essentially, an object of class TOUR is a permutation vector containing the order of cities to visit.

Usage

```
## conctructor
TOUR(x, method=NA, tsp=NULL)

## coercion
as.TOUR(object)

## methods
## S3 method for class 'TOUR'
print(x, ...)
```

Arguments

x	an integer permutation vector or, for the methods an object of class TOUR.
object	data (an integer vector) which can be coerced to TOUR..
method	character string; method used to create the tour.
tsp	TSP object the tour applies to. If available then the tour will include the tour length. Also the labes of the cities will be available in the tour (otherwise the labels of x are used).
...	further arguments are passed on.

Details

Since an object of class TOUR is an integer vector, it can be subsetted as an ordinary vector or coerced to an integer vector using `as.integer()`. It also contains the names of the objects as labels. Additionally, TOUR has the following attributes: `"method"`, `"tour_length"`.

For most functions, e.g., `tour_length` or `image`, the TSP/ATSP object used to find the tour is still needed, since the tour does not contain the distance information.

Author(s)

Michael Hahsler

See Also

TSP, ATSP, tour_length, image.

Examples

```
TOUR(1:10)

## calculate a tour
data("USCA50")
tour <- solve_TSP(USCA50)
tour

## get permutation vector
as.integer(tour)

## get tour length directly from tour
attr(tour, "tour_length")

## show labels
labels(tour)
```

tour_length *Calculate the length of a tour*

Description

Calculate the length of a tour given a TSP and an order.

Usage

```
tour_length(x, ...)
## S3 method for class 'TOUR'
tour_length(x, tsp = NULL, ...)
```

Arguments

x	an object of class TOUR.
tsp	a TSP object of class TSP, ATSP or ETSP.
...	further arguments are currently unused.

Details

If no tsp is given, then the tour length stored in x as attribute "tour_length" is returned. If tsp is given then the tour length is recalculated.

If a distance in the tour is infinite, the result is also infinite. If the tour contains positive and negative infinite distances then the method returns NA.

Author(s)

Michael Hahsler

See Also

TOUR, TSP, ATSP and ETSP.

Examples

```
data("USCA50")

## original order
tour_length(solve_TSP(USCA50, method="identity"))

## length of a manually created (random) tour
tour <- TOUR(sample(seq(n_of_cities(USCA50))))
tour
tour_length(tour)
tour_length(tour, USCA50)
```

TSP *Class TSP – Symmetric traveling salesperson problem*

Description

Constructor to create an instance of a symmetric traveling salesperson problem (TSP) and some auxiliary methods.

Usage

```
## constructor
TSP(x, labels = NULL, method = NULL)

## coercion
as.TSP(x)

## methods
## S3 method for class 'TSP'
n_of_cities(x)
## S3 method for class 'TSP'
image(x, order, col = gray.colors(64), ...)
## S3 method for class 'TSP'
labels(object, ...)
## S3 method for class 'TSP'
print(x, ...)
```

Arguments

x, object	an object (currently dist or a symmetric matrix) to be converted into a TSP or, for the methods, an object of class TSP.
labels	optional city labels. If not given, labels are taken from x.
method	optional name of the distance metric. If x is a dist object, then the method is taken from that object.
col	color scheme for image.
order	order of cities for the image as an integer vector or an object of class TOUR.
...	further arguments are passed on.

Details

Objects of class TSP are internally represented as dist objects (use as.dist() to get the dist object).

Value

TSP() returns x as an object of class TSP.

n_of_cities() returns the number of cities in x.

labels() returns a vector with the names of the cities in x.

Author(s)

Michael Hahsler

See Also

TOUR, insert_dummy, tour_length, solve_TSP.

Examples

```
data("iris")
d <- dist(iris[-5])

## create a TSP
tsp <- TSP(d)
tsp

## use some methods
n_of_cities(tsp)
labels(tsp)
image(tsp)
```

TSPLIB *Read and write TSPLIB files*

Description

Reads and writes TSPLIB format files. TSPLIB files can be used by most TSP solvers. Sample instances for the TSP in TSPLIB format are available on the TSPLIB homepage (see references).

Usage

```
write_TSPLIB(x, file, precision = 6, inf = NULL, neg_inf = NULL)
read_TSPLIB(file, precision = 0)
```

Arguments

x	an object of class TSP, ATSP or ETSP. NAs are not allowed.
file	file name or a connection.
precision	controls the number of decimal places used to represent distances (see details). If x already is integer, this argument is ignored and x is used as is.
inf	replacement value for Inf (TSPLIB format cannot handle Inf). If inf is NULL, a large value of $max(x) + 2range(x)$ (ignoring infinite entries) is used.
neg_inf	replacement value for -Inf. If no value is specified, a small value of $min(x) - 2range(x)$ (ignoring infinite entries) is used.

Details

In the TSPLIB format distances are represented by integer values. Therefore, if `x` contains `double` values (which is normal in R) the values given in `x` are multiplied by $10^{precision}$ before coercion to `integer`. Note that therefore all results produced by programs using the TSPLIB file as input need to be divided by $10^{precision}$ (i.e., the decimal point has to be shifted `precision` placed to the left).

Currently only the following `EDGE_WEIGHT_TYPE`s are implemented: `EXPLICIT`, `EUC_2D` and `EUC_3D`.

Value

`read_TSPLIB` returns an object of class `TSP` or `ATSP`.

Author(s)

Michael Hahsler

References

TSPLIB home page, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Examples

```
## Drilling problem from TSP
drill <- read_TSPLIB(system.file("examples/d493.tsp", package = "TSP"))
drill
tour <- solve_TSP(drill, method = "nn", two_opt = TRUE)
tour
plot(drill, tour, cex=.6, col = "red", pch= 3, main = "TSPLIB: d493")


## Write and read data in TSPLIB format
x <- data.frame(x=runif(5), y=runif(5))

## create TSP, ATSP and ETSP (2D)
tsp <- TSP(dist(x))
atsp <- ATSP(dist(x))
etsp <- ETSP(x[,1:2])

write_TSPLIB(tsp, file="example.tsp")
#file.show("example.tsp")
r <- read_TSPLIB("example.tsp")
r

write_TSPLIB(atsp, file="example.tsp")
#file.show("example.tsp")
r <- read_TSPLIB("example.tsp")
r

write_TSPLIB(etsp, file="example.tsp")
#file.show("example.tsp")
r <- read_TSPLIB("example.tsp")
r
```

```
## clean up
unlink("example.tsp")
```

USCA *USCA312/USCA50 – 312/50 cities in the US and Canada*

Description

The USCA312 dataset contains the distances between 312 cities in the US and Canada as an object of class TSP. USCA50 is a subset of USCA312 containing only the first 50 cities.

The USCA312_map dataset contains spatial data of the 312 cities.

Usage

```
data("USCA312")
data("USCA312_map")
data("USCA50")
```

Format

USCA312 and USCA50 are objects of class TSP. USCA312_map contains in USCA312_coords the spatial coordinates of the 312 cities and in USCA312_basemap a part of the map of North America.

Details

For USCA312_map several packages for geographic data are needed (see Examples section).

We want to thank Roger Bivand for his help with plotting the map.

Author(s)

Michael Hahsler

Source

John Burkardt, CITIES – City Distance Datasets, Florida State University, Department of Scientific Computing

Examples

```
data("USCA312")

## calculate a tour
tour <- solve_TSP(USCA312)
tour

# Using the USCA312_map requites the suggested package sp, maps, and maptools.
# We run the example only if the packages are installed.
```

```
# You can load them with library() in your code.

if(require(sp) &&
    require(maps) &&
    require(maptools)) {

  data("USCA312_map")

  ## plot map
  plot(as(USCA312_coords, "Spatial"), axes=TRUE)
  plot(USCA312_basemap, add=TRUE, col = "gray")

  ## plot tour and add cities
  tour_line <- SpatialLines(list(Lines(list(
  Line(USCA312_coords[c(tour, tour[1]),])), ID="1")))

  plot(tour_line, add=TRUE, col = "red")
  points(USCA312_coords, pch=3, cex=0.4, col="black")
}
```

Index