Package 'TSP'

November 14, 2026

Type Package

Title Infrastructure for the Traveling Salesperson Problem

Version 1.3.0

Date 2026-11-13

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. Hahsler and Hornik (2007) <doi:10.18637/jss.v023.i02>.

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

URL https://github.com/mhahsler/TSP, http://michael.hahsler.net/TSP/

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

Depends R (>= 3.5.0)

Imports graphics, foreach, utils, stats, grDevices

Suggests maps, doParallel, testthat (>= 3.0.0)

Config/testthat/edition 3

Encoding UTF-8

License GPL-3

Copyright All code is Copyright (C) Michael Hahsler and Kurt Hornik.

Config/roxygen2/version 8.1.1

NeedsCompilation yes

Author Michael Hahsler [aut, cre, cph] (ORCID: <https://orcid.org/0000-0003-2716-1405>), Kurt Hornik [aut, cph] (ORCID: <https://orcid.org/0000-0003-4198-9911>)

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

Repository CRAN

Date/Publication 2026-11-13 23:10:13 UTC

Contents

ATSP 2
Concorde 4
cut_tour 6
ETSP 7
insert_dummy 9
reformulate_ATSP_as_TSP 11
solve_TSP 13
TOUR 20
tour_length 21
TSP 23
TSPLIB 24
USCA 26

Index **28**

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

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

as.ATSP(x)

## S3 method for class 'matrix'
as.ATSP(x)

## S3 method for class 'dist'
as.ATSP(x)

## S3 method for class 'ATSP'
print(x, ...)

## S3 method for class 'ATSP'
n_of_cities(x)

## S3 method for class 'ATSP'
labels(object, ...)

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

```
## S3 method for class 'ATSP'
as.matrix(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.
...	further arguments are passed on.
order	order of cities as an integer vector or an object of class TOUR.
col	color scheme for image.

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

Other TSP: ETSP(), TSP(), insert_dummy(), reformulate_ATSP_as_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 TSPs and then solved.

Usage

```
concorde_path(path)

concorde_help()

linkern_help()
```

Arguments

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

Details

Note: Concorde and linkern natively solve ETSPs (Euclidean TSPs given as coordinates). The coordinates are passed on to the solvers as they are (after scaling to integers, see below), so no distance matrix needs to be calculated and the memory use grows only linearly with the number of cities. The other methods in `solve_TSP()` first convert an ETSP into a distance matrix.

Value

`concorde_path()` returns the path to the executable. The other functions return nothing.

Installation of Concorde

The Concorde TSP Solver is freely available for academic research. It is not included in the **TSP** R package and has to be obtained separately from the Concorde download page. Either download the precompiled executables and place them in a suitable directory (make sure they are executable), or you can get the source code and compile the program 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 Sys.setenv()).

Using Concorde for solve_TSP()

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 in rounding errors that are worse than what is specified in the precision control parameter.

For Euclidean problems (ETSP), the coordinates are shifted to start at zero and multiplied by 10^precision, because Concorde rounds the calculated distances to integers. To avoid integer overflows in Concorde, the scaling is reduced (with a warning) so that no scaled coordinate exceeds 10^7. write_TSPLIB() gives a warning if cities end up with identical coordinates (e.g., because rounding made distinct cities identical, i.e., their distances became smaller than 1); increase precision or rescale the data in this case.

Concorde and linkern have their own random number generators, which set.seed() cannot reach. Their seed can be set with the optional control parameter seed, e.g., control = list(seed = 1234). It is passed on to the programs (option -s) and has to be a whole number between 0 and .Machine$integer.max. If no seed is specified, one random number is drawn from R's random number generator and used as the seed, so calling set.seed() before solve_TSP() also makes these methods reproducible (this advances R's global random-number stream by one draw).

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.

If Concorde takes too long, then you can interrupt solve_TSP() using Esc/CTRL-C. On most operating systems, this will also terminate the Concorde executable. If Concorde keeps running, then you can kill the 'concorde' process via your operating system.

Author(s)

Michael Hahsler

References

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

David Applegate, 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

Other solver: solve_TSP()

Examples

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


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

concorde_help()

data("USCA312")

## run Concorde in verbose mode (-v) with fast cuts only (-V)
## Note: use the control parameter verbose = FALSE to suppress Concorde's output
set.seed(1234)
solve_TSP(USCA312, method = "concorde", control = list(clo = "-v -V"))

## Concorde and linkern can directly solve Euclidean TSPs (ETSP). The
## coordinates are used directly, so no large distance matrix is calculated.
etsp <- ETSP(data.frame(x = runif(20), y = runif(20)))
tour <- solve_TSP(etsp, method = "concorde", control = list(verbose = FALSE))
tour
plot(etsp, tour)

## 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)

## S3 method for class 'TOUR'
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

Returns 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

Other TOUR: TOUR(), tour_length()

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

```
ETSP(x, labels = NULL)

as.ETSP(x)

## S3 method for class 'matrix'
```

```
as.ETSP(x)

## S3 method for class 'data.frame'
as.ETSP(x)

## S3 method for class 'ETSP'
as.TSP(x)

## S3 method for class 'ETSP'
as.matrix(x, ...)

## S3 method for class 'ETSP'
print(x, ...)

## S3 method for class 'ETSP'
n_of_cities(x)

## S3 method for class 'ETSP'
labels(object, ...)

## 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 = 2, labels = TRUE, ...)
```

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	logical; plot city labels.
...	further arguments are passed on.
order	order of cities for the image as an integer vector or an object of class TOUR.
col	color scheme for image.
tour, y	a tour to be visualized.
tour_lty, tour_col	line type and color for tour.

Details

Objects of class ETSP are internally represented as 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

Other TSP: ATSP(), TSP(), insert_dummy(), reformulate_ATSP_as_TSP()

Examples

```
## create a random ETSP
n <- 20
x <- data.frame(x = runif(n), y = runif(n), row.names = LETTERS[1:n])
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")

# plot with custom labels
plot(etsp, tour, tour_col = "red", labels = FALSE)
text(etsp, paste("City", rownames(etsp)), pos = 1)
```

---

insert_dummy *Insert dummy cities into a distance matrix*

Description

Inserts dummy cities into a TSP problem. 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 solver (Garfinkel 1985).

Usage

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

## S3 method for class 'TSP'
insert_dummy(x, n = 1, const = 0, inf = Inf, label = "dummy")

## S3 method for class 'ATSP'
insert_dummy(x, n = 1, const = 0, inf = Inf, label = "dummy")
```

```
## S3 method for class 'ETSP'
insert_dummy(x, n = 1, const = 0, inf = Inf, label = "dummy")
```

Arguments

x	an object with a TSP problem.
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

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

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.

Value

returns an object of the same class as x.

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

Other TSP: ATSP(), ETSP(), TSP(), reformulate_ATSP_as_TSP()

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 create 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 an ATSP as a symmetric TSP*

Description

An 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)

filter_ATSP_as_TSP_dummies(tour, atsp)
```

Arguments

x	an ATSP.
infeasible	value for infeasible connections.
cheap	value for distance between a city and its corresponding dummy city.
tour	a TOUR created for an ATSP reformulated as a TSP.
atsp	the original ATSP.

Details

To reformulate an ATSP as a TSP, a dummy city (e.g., for 'New York' a dummy city 'New York*') is added. Between each city and its corresponding dummy city a very small (or negative) distance with value cheap is used. To ensure that each city always occurs in the solution together with its dummy city, this cost has to be much smaller than the distances in the TSP. 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 prevents the solver from using these links. We use infinite values here and solve_TSP() treats them appropriately.

filter_ATSP_as_TSP_dummies() can be used to extract the solution for the original ATSP from the tour found for an ATSP reformulated as a TSP. Note that the symmetric TSP tour does not reveal the direction for the ATSP. The filter function computes the tour length for both directions and returns the shorter tour.

solve_TSP() has a parameter as_TSP that performs the reformulation and filters the dummy cities automatically.

Note on performance: Doubling the problem size reduces performance and can have a particularly negative impact on solution quality for heuristics. It should be used only together with Concorde when the optimal solution is required. Most heuristics can solve ATSPs directly with good solution quality.

Value

reformulate_ATSP_as_TSP() returns a TSP object. filter_ATSP_as_TSP_dummies() returns a TOUR 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

Other TSP: ATSP(), ETSP(), TSP(), insert_dummy()

Examples

```
data("USCA50")

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

## reformulate as a TSP (by doubling the number of cities with dummy cities marked with *)
tsp <- reformulate_ATSP_as_TSP(atsp)
tsp

## create tour for the TSP. You should use Concorde to find the optimal solution.
# tour_tsp <- solve_TSP(tsp, method = "concorde")
# The standard heuristic is bad for this problem. We use it here because
#   Concord may not be installed.
tour_tsp <- solve_TSP(tsp)
head(labels(tour_tsp), n = 10)
tour_tsp
# The tour length is -Inf since it includes cheap links
#  from a city to its dummy city.

## get the solution for the original ATSP by filtering out the dummy cities.
tour_atsp <- filter_ATSP_as_TSP_dummies(tour_tsp, atsp = atsp)
tour_atsp
head(labels(tour_atsp), n = 10)

## This process can also be done automatically by using as_TSP = TRUE:
# solve_TSP(atsp, method = "concorde", as_TSP = TRUE)

## The default heuristic can directly solve ATSPs with results close to the
#  optimal solution of 12715.
solve_TSP(atsp, control = list(rep = 10))
```

---

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 'ATSP'
solve_TSP(x, method = NULL, control = NULL, as_TSP = FALSE, ...)

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

Arguments

x	a TSP problem.
method	method to solve the TSP (default: "arbitrary insertion" construction followed by the "two_opt" improvement heuristic).
control	a list of arguments passed on to the TSP solver selected by method.
...	additional arguments are added to control.
as_TSP	should the ATSP reformulated as a TSP for the solver?

Value

An object of class TOUR.

TSP Methods

TSP supports tour construction methods, tour improvement heuristics and an interface to Concorde (a state-of-the-art external solver). Tour construction methods create a tour from scratch, typically using a construction heuristic. Tour improvement heuristics take an existing tour and try to improve it. If no initial tour is provided for the improvement heuristic, then an initial tour (e.g., a random tour) is automatically created.

Most heuristic methods accept the following extra control parameters in addition to the parameters specified below:

- "rep": an integer indicating how many replications (random restarts) should be performed. The best result is returned. The replications can be performed in parallel (see Section Parallel Execution Support).
- "opt": a character string selecting a local-search refinement that is performed on the constructed tour. This is just for convenience so a constructive heuristic can be followed by refinement in a single call. Available values are `"two"` (2-opt), `"or"` (Or-opt), `"three"` (3-opt), and `"two+or"` (2-opt and Or-opt alternately until neither improves the tour). The default is no refinement. See the methods `"two_opt"`, `"or_opt"` and `"three_opt"` below.
- "two_opt": **deprecated**. Use `opt = "two"` instead.
- "verbose": a logical. Most implementations provide verbose output to monitor progress. This argument can also be helpful to see what heuristics are used and what the default parameters are.

Tour Construction Methods:

These methods construct tours from scratch. The available tour construction methods are:

- **"identity", "random"** return a tour representing the order in the data (identity order) or a random order. [TSP, ATSP]

- **"nearest_insertion", "farthest_insertion", "cheapest_insertion", "arbitrary_insertion"**
  Nearest, farthest, cheapest and arbitrary insertion algorithms for symmetric and asymmetric TSPs (Rosenkrantz et al. 1977). [TSP, ATSP]
  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 algorithm 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* from 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 try to build the tour using cities that fit well into the partial tour constructed so far. The idea behind farthest insertion is to link distant cities into the tour first to establish an outline of the whole tour early.
  Additional control options:
  - "start" index of the first city (default: a random city).
- **"nn", "repetitive_nn"** Nearest neighbor and repetitive nearest neighbor algorithms for symmetric and asymmetric TSPs (Rosenkrantz et al. 1977). [TSP, ATSP]
  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: a random city).

Tour Improvement Heuristics:

Tour improvement methods take a tour as the argument `tour` and try to improve the tour. If no tour is provided, a random tour is used as the initial tour.

The following tour improvement heuristics are available:

- **"sa"** Simulated Annealing for TSPs (Kirkpatrick et al, 1983) [TSP, ATSP]
  A tour refinement method that uses simulated annealing with subtour reversal as a local move. The optimizer is `stats::optim()` with method `"SANN"`. This method is typically a lot slower than `"two_opt"` and requires parameter tuning for the cooling schedule.
  Note: For speed, a relatively small number of evaluations (`maxit`) is used by default. A much larger value will result in more competitive results.
  Additional control options:
  - "tour" an existing tour which should be improved. If no tour is given, a random tour is used.
  - "local_move" a function that creates a local move with the current tour as the first and the TSP as the second parameter. Defaults to randomized subtour reversal.

  - "temp" initial temperature. Defaults to the length of the current tour divided by the number of cities.
  - "tmax" number of evaluations per temperature step. Default is 10.
  - "maxit" number of evaluations. Default is 10000.
  - "trace" set to 1 to print progress.

  See `stats::optim()` for more details on the parameters.
- **"two_opt"** Two-edge exchange improvement procedure (Croes 1958). [TSP, ATSP]
  This is a tour refinement procedure which systematically exchanges two edges in the graph represented by the distance matrix until no improvements are possible. Exchanging two edges is equal to reversing part of the tour. The resulting tour is called *2-optimal.* This heuristic takes $O(n^2)$ time.
  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 with random restarts (default: 1).
- **"or_opt"** Or-opt segment insertion (Or 1976). [TSP, ATSP]
  This is a tour refinement procedure which moves segments of 1 to `max_segment` consecutive cities to a better position in the tour until no improvement is possible. For symmetric problems, segments may also be reinserted in reverse order. Or-opt can move a misplaced city or small group of cities in a single step, which 2-opt cannot. It is often also used as an additional improvement step after "two_opt" (see Examples). This heuristic also takes $O(n^2)$ time.
  Additional control options:
  - "tour" an existing tour which should be improved. If no tour is given, a random tour is used.
  - "max_segment" maximum number of cities in a moved segment (default: 3).
- **"three_opt"** Three-edge exchange improvement procedure (Bock 1958; Lin 1965). [TSP, ATSP]
  This is a tour refinement procedure which removes three edges and reconnects the three resulting parts of the tour in the best possible way until no improvements are possible. The resulting tour is called *3-optimal.* The search checks all triples of edges and each pass takes $O(n^3)$ time, so it is only practical for small to medium problems. For ATSPs, only the reconnection that does not reverse any part of the tour is used.
  Additional control options:
  - "tour" an existing tour which should be improved. If no tour is given, a random tour is used.

State-of-the-art Solvers Interfaces:

- **"concorde"** Concorde algorithm (Applegate et al. 2001). [TSP, ETSP]
  Concorde is an advanced exact TSP solver for *symmetric* TSPs based on branch-and-cut. ATSPs can be solved using `reformulate_ATSP_as_TSP()` done automatically with `as_TSP = TRUE`. The program is not included in this package and has to be obtained and installed separately.
  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` places to the left). The interface to Concorde uses `write_TSPLIB()`.
  - "seed" an optional whole number between 0 and `.Machine$integer.max` that is passed on to Concorde's `-s` option to make its randomized components reproducible. **If no seed is specified, Concorde is still seeded:** one random number is drawn from R's random number generator and passed on as the seed. Therefore, calling `set.seed()` before `solve_TSP()` also makes Concorde reproducible (see Section Reproducible Results). Do not combine `seed` with `-s` in `clo`.
  - "verbose" logical; `FALSE` suppresses the output printed to the terminal.

- **"linkern"** Concorde's Chained Lin-Kernighan heuristic (Applegate et al. 2003). [TSP, ETSP]
  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. This includes `seed`, which is passed on to linkern's `-s` option and is drawn from R's random number generator if it is not specified.

Treatment of `NA`s and Infinite Values in `x`

TSP and ATSP need to contain valid distances. `NA`s 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) or infeasible connections. 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`. `-Inf` is replaced by $min(x) - 2range(x)$ and can be used to encourage the solver to place two objects next to each other.

Solving ATSP and ETSP

Some solvers (including Concorde) cannot solve ATSP directly. An `ATSP` can be reformulated as a 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()`).

Only methods "concorde" and "linkern" currently solve ETSPs directly. For all other methods, ETSPs are converted into TSPs by creating a distance matrix and then solved. Note: distance matrices can become very large, leading to high memory usage and long computation times.

Parallel Execution Support

If several repetitions are performed (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., `doParallel::registerDoParallel()`).

Reproducible Results

All randomized methods implemented in this package (e.g., `"random"`, `"nn"`, `"arbitrary_insertion"`, `"sa"`, and repetitions with `rep`) use R's global random number generator. Call `set.seed()` before `solve_TSP()` to make the results reproducible, and note that every call advances the global random-number stream. If a parallel backend is used for repetitions (`rep > 1` and `"repetitive_nn"`), one random number per repetition is drawn from the global random number generator of the main R process and used as the seed of the iteration.

Concorde and linkern are external programs with their own random number generators. The programs are seeded with with one random number is drawn from R's global random number generator. This means that `set.seed()` also makes their results reproducible. In the unlikely evenet, that a specific seed value for these programs is needed, then it can be set via the control option `seed` (a whole number between 0 and `.Machine$integer.max`, e.g., `control = list(seed = 1234)`), which is passed on to the program via its `-s` command line option.

Author(s)

Michael Hahsler

References

David Applegate, 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.doi:10.1007/3540455868_7

D. Applegate, W. Cook and A. Rohe (2003): Chained Lin-Kernighan for Large Traveling Salesman Problems. *INFORMS Journal on Computing*, 15(1):82–92. doi:10.1287/ijoc.15.1.82.15157

F. Bock (1958): An algorithm for solving traveling-salesman and related network optimization problems. Unpublished manuscript presented at the 14th ORSA meeting, Saint Louis.

G.A. Croes (1958): A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812. doi:10.1287/opre.6.6.791

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi (1983): Optimization by Simulated Annealing. *Science* 220 (4598): 671–80 doi:10.1126/science.220.4598.671

S. Lin and B. Kernighan (1973): An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2): 498–516. doi:10.1287/opre.21.2.498

S. Lin (1965): Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269. doi:10.1002/j.15387305.1965.tb04146.x

I. Or (1976): Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking. PhD thesis, Northwestern University, Evanston, IL.

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. doi:10.1007/978-1402096884_3

See Also

Other solver: `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)

## manually run a construction followed by an improvement heuristic by
## calling solve_TSP() twice.
tour_constr <- solve_TSP(etsp, method = "nn")
tour_constr

tour_improved <- solve_TSP(etsp, method = "sa", tour = tour_constr)
tour_improved

## chain improvement heuristics: 2-opt, then Or-opt, then 3-opt
tour_2 <- solve_TSP(etsp, method = "two_opt", tour = tour_constr)
tour_or <- solve_TSP(etsp, method = "or_opt", tour = tour_2)
tour_3 <- solve_TSP(etsp, method = "three_opt", tour = tour_or)
c(nn = tour_length(tour_constr), two_opt = tour_length(tour_2),
  or_opt = tour_length(tour_or), three_opt = tour_length(tour_3))

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

## 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$'nn+two_opt' <- solve_TSP(USCA50, method = "nn", opt = "two")
tours$'nn+two_opt+or_opt' <- solve_TSP(USCA50, method = "nn", opt = "two+or")
tours$'nn+rep_10' <- solve_TSP(USCA50, method = "nn", rep = 10)
```

```
tours$'nn+two_opt+rep_10' <- solve_TSP(USCA50, method = "nn", opt = "two", rep = 10)
tours$'arbitrary_insertion+two_opt' <- solve_TSP(USCA50)

## show first tour
tours[[1]]

## compare tour lengths
opt <- 14497 # obtained by Concorde
tour_lengths <- sort(sapply(tours, tour_length), decreasing = TRUE)
dotchart(tour_lengths / opt * 100 - 100, xlab = "percent excess over optimum")
abline(v = 0, col = "red")
```

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

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

as.TOUR(object)

## S3 method for class 'numeric'
as.TOUR(object)

## S3 method for class 'integer'
as.TOUR(object)

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

Arguments

x	an integer permutation vector or, for the methods an object of class 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 labels of the cities will be available in the tour (otherwise the labels of x are used).
object	data (an integer vector) which can be coerced to TOUR.
...	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.TSP()`, 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

Other TOUR: `cut_tour()`, `tour_length()`

Examples

```
TOUR(1:10)

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

## get tour length directly from tour
tour_length(tour)

## get permutation vector
as.integer(tour)

## show labels
labels(tour)
```

---

tour_length *Calculate the length of a tour*

Description

Calculate the length of a TOUR for a TSP.

Usage

```
tour_length(x, ...)

## S3 method for class 'TSP'
tour_length(x, order, ...)

## S3 method for class 'ATSP'
```

```
tour_length(x, order, ...)

## S3 method for class 'ETSP'
tour_length(x, order, ...)

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

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

Arguments

x	a TSP problem or a TOUR.
...	further arguments are currently unused.
order	an object of class TOUR
tsp	a TSP object.

Details

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

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

Other TOUR: TOUR(), cut_tour()

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

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

as.TSP(x)

## S3 method for class 'dist'
as.TSP(x)

## S3 method for class 'matrix'
as.TSP(x)

## S3 method for class 'TSP'
as.dist(m, ...)

## S3 method for class 'TSP'
print(x, ...)

n_of_cities(x)

## S3 method for class 'TSP'
n_of_cities(x)

## S3 method for class 'TSP'
labels(object, ...)

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

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.
m	a TSP object to be converted to a dist object.
...	further arguments are passed on.

order	order of cities for the image as an integer vector or an object of class TOUR.
col	color scheme for image.

Details

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

Impermissible paths can be set to a distance of +Inf. NAs are not allowed, and -Inf will allow the algorithm to find only an admissible tour, not the best one.

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

Other TSP: ATSP(), ETSP(), insert_dummy(), reformulate_ATSP_as_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. Many problems in TSPLIB format can be found in the local copy of the TSPLIB95 problem library.

Usage

```
read_TSPLIB(file, precision = 0)

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

## S3 method for class 'TSP'
write_TSPLIB(x, file, precision = 6, inf = NULL, neg_inf = NULL)

## S3 method for class 'ATSP'
write_TSPLIB(x, file, precision = 6, inf = NULL, neg_inf = NULL)

## S3 method for class 'ETSP'
write_TSPLIB(x, file, precision = 6, inf = NULL, neg_inf = NULL)
```

Arguments

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.
x	an object with a TSP problem. NAs are not allowed.
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 $10^{precision}$ and rounded to the nearest 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 places to the left).

For TSP and ATSP objects, a warning is given if distances that are not zero become zero after the conversion to integers (i.e., precision is too low).

For ETSP objects, the coordinates are written with precision digits (in scientific notation) and are not scaled. A warning is given if two or more cities have identical coordinates in the written file because their distance is zero.

Currently only the following EDGE_WEIGHT_TYPEs are implemented: EXPLICIT, EUC_2D, EUC_3D, ATT and GEO.

Value

returns an object of class TSP or ATSP.

Author(s)

Michael Hahsler

References

Reinelt, Gerhard. 1991. "TSPLIB—a Traveling Salesman Problem Library." ORSA Journal on Computing 3 (4): 376–84. doi:10.1287/ijoc.3.4.376

See Also

Other data: USCA

Examples

```
## Drilling problem from TSP
drill <- read_TSPLIB(system.file("examples/d493.tsp", package = "TSP"))
drill
tour <- solve_TSP(drill, method = "nn", opt = "two")
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.

Format

USCA312 and USCA50 are objects of class TSP. USCA312_GPS is a data.frame with city name, long and lat.

Details

The USCA312_GPS dataset contains the location (long/lat) of the 312 cities.

Author(s)

Michael Hahsler

Source

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

See Also

Other data: TSPLIB

Examples

```
data("USCA312")

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

# Visualize the tour if package maps is installed
if(require("maps")) {

  library(maps)
  data("USCA312_GPS")
  head(USCA312_GPS)

  plot((USCA312_GPS[, c("long", "lat")]), cex = .3)
  map("world", col = "gray", add = TRUE)
  polygon(USCA312_GPS[, c("long", "lat")][tour,], border = "red")
}
```

Index

∗ **TOUR**
  cut_tour, 6
  TOUR, 20
  tour_length, 21
∗ **TSP**
  ATSP, 2
  ETSP, 7
  insert_dummy, 9
  reformulate_ATSP_as_TSP, 11
  TSP, 23
∗ **classes**
  ATSP, 2
  ETSP, 7
  TOUR, 20
  TSP, 23
∗ **datasets**
  USCA, 26
∗ **data**
  TSPLIB, 24
  USCA, 26
∗ **documentation**
  Concorde, 4
∗ **file**
  TSPLIB, 24
∗ **manip**
  insert_dummy, 9
∗ **optimize**
  cut_tour, 6
  reformulate_ATSP_as_TSP, 11
  solve_TSP, 13
  tour_length, 21
∗ **solver**
  Concorde, 4
  solve_TSP, 13

as.ATSP *(*ATSP*)*, 2
as.dist(), *24*
as.dist.TSP *(*TSP*)*, 23
as.ETSP *(*ETSP*)*, 7
as.matrix.ATSP *(*ATSP*)*, 2
as.matrix.ETSP *(*ETSP*)*, 7
as.TOUR *(*TOUR*)*, 20
as.TSP *(*TSP*)*, 23
as.TSP.ETSP *(*ETSP*)*, 7
ATSP, 2, *4*, *12*, *17*, *25*
ATSP(), *9*, *10*, *12*, *24*

Concorde, 4, *16–18*
concorde *(*Concorde*)*, 4
concorde_help *(*Concorde*)*, 4
concorde_help(), *16*
concorde_path *(*Concorde*)*, 4
connection, *25*
cut_tour, 6
cut_tour(), *21*, *22*

dist, *23*
doParallel::registerDoParallel(), *17*

ETSP, *4*, *5*, 7, *17*, *25*
ETSP(), *3*, *10*, *12*, *24*

filter_ATSP_as_TSP_dummies
        *(*reformulate_ATSP_as_TSP*)*, 11

image.ATSP *(*ATSP*)*, 2
image.ETSP *(*ETSP*)*, 7
image.TSP *(*TSP*)*, 23
image.TSP(), *21*
insert_dummy, 9
insert_dummy(), *3*, *9*, *12*, *24*

labels.ATSP *(*ATSP*)*, 2
labels.ETSP *(*ETSP*)*, 7
labels.TSP *(*TSP*)*, 23
linkern_help *(*Concorde*)*, 4

n_of_cities *(*TSP*)*, 23
n_of_cities.ATSP *(*ATSP*)*, 2
n_of_cities.ETSP *(*ETSP*)*, 7

plot.ETSP *(*ETSP*)*, 7
print.ATSP *(*ATSP*)*, 2
print.ETSP *(*ETSP*)*, 7
print.TOUR *(*TOUR*)*, 20
print.TSP *(*TSP*)*, 23

read_TSPLIB *(*TSPLIB*)*, 24
reformulate_ATSP_as_TSP, 11
reformulate_ATSP_as_TSP(), *3*, *9, 10*, *16, 17*, *24*

set.seed(), *5*, *17, 18*
solve_TSP, 13
solve_TSP(), *4, 5*, *12*
stats::optim(), *15, 16*
Sys.setenv(), *5*

TOUR, *6*, *8*, *12*, *14*, *20*, 20, *21, 22*, *24*
TOUR(), *7*, *22*
tour_length, 21
tour_length(), *7*, *21*
TSP, *4*, *12*, *17*, *21*, 23, *25*
TSP(), *3*, *9, 10*, *12*
TSPLIB, 24, *27*

USCA, *26*, 26
USCA312 *(*USCA*)*, 26
USCA312_GPS *(*USCA*)*, 26
USCA50 *(*USCA*)*, 26

write_TSPLIB *(*TSPLIB*)*, 24
write_TSPLIB(), *5*, *17*