TSP - Infrastructure for the Traveling Salesperson Problem

Hahsler, Michael; Hornik, Kurt

Published in:
Journal of Statistical Software

Published: 01/11/2007

Document Version
Publisher's PDF, also known as Version of record

Journal of Statistical Software

December 2007, Volume 23, Issue 2. *http://www.jstatsoft.org/*

TSP – Infrastructure for the Traveling Salesperson Problem

Michael Hahsler
Wirtschaftsuniversität Wien

Kurt Hornik
Wirtschaftsuniversität Wien

Abstract

The traveling salesperson (or, salesman) problem (TSP) is a well known and important combinatorial optimization problem. The goal is to find the shortest tour that visits each city in a given list exactly once and then returns to the starting city. Despite this simple problem statement, solving the TSP is difficult since it belongs to the class of NP-complete problems. The importance of the TSP arises besides from its theoretical appeal from the variety of its applications. Typical applications in operations research include vehicle routing, computer wiring, cutting wallpaper and job sequencing. The main application in statistics is combinatorial data analysis, e.g., reordering rows and columns of data matrices or identifying clusters. In this paper, we introduce the R package **TSP** which provides a basic infrastructure for handling and solving the traveling salesperson problem. The package features S3 classes for specifying a TSP and its (possibly optimal) solution as well as several heuristics to find good solutions. In addition, it provides an interface to **Concorde**, one of the best exact TSP solvers currently available.

Keywords: combinatorial optimization, traveling salesman problem, R.

1. Introduction

The traveling salesperson problem (TSP; Lawler, Lenstra, Rinnooy Kan, and Shmoys 1985; Gutin and Punnen 2002) is a well known and important combinatorial optimization problem. The goal is to find the shortest tour that visits each city in a given list exactly once and then returns to the starting city. Formally, the TSP can be stated as follows. The distances between n cities are stored in a distance matrix $\mathbf{D}$ with elements d_{ij} where $i, j = 1 \dots n$ and the diagonal elements d_{ii} are zero. A *tour* can be represented by a cyclic permutation π of $\{1, 2, \dots, n\}$ where $\pi(i)$ represents the city that follows city i on the tour. The traveling salesperson problem is then the optimization problem to find a permutation π that minimizes

the *length of the tour* denoted by

$$\sum_{i=1}^{n} d_{i\pi(i)}. \tag{1}$$

For this minimization task, the tour length of $(n-1)!$ permutation vectors have to be compared. This results in a problem which is very hard to solve and in fact known to be NP-complete (Johnson and Papadimitriou 1985a). However, solving TSPs is an important part of applications in many areas including vehicle routing, computer wiring, machine sequencing and scheduling, frequency assignment in communication networks (Lenstra and Kan 1975; Punnen 2002). Applications in statistical data analysis include ordering and clustering objects. For example, data analysis applications in psychology ranging from profile smoothing to finding an order in developmental data are presented by Hubert and Baker (1978). Clustering and ordering using TSP solvers is currently becoming popular in biostatistics. For example, Ray, Bandyopadhyay, and Pal (2007) describe an application for ordering genes and Johnson and Liu (2006) use a TSP solver for clustering proteins.

In this paper, we give a very brief overview of the TSP and introduce the package **TSP** written in the R system for statistical computing (R Development Core Team 2007). It is available from the Comprehensive R Archive Network at `http://CRAN.R-project.org/` and provides infrastructure for handling and solving TSPs. The paper is organized as follows: In Section 2, we briefly present important aspects of the TSP including different problem formulations and approaches to solve TSPs. In Section 3, we give an overview of the infrastructure implemented in **TSP** and the basic usage. In Section 4, several examples are used to illustrate the package's capabilities. Section 5 concludes the paper.

2. Theory

In this section, we briefly summarize some aspects of the TSP which are important for the implementation of the **TSP** package described in this paper. For a complete treatment of all aspects of the TSP, we refer the interested reader to the classic book edited by Lawler *et al.* (1985) and the more modern book edited by Gutin and Punnen (2002).

It has to be noted that in this paper, following the origin of the TSP, the term *distance* is used. Distance is used here interchangeably with dissimilarity or cost and, unless explicitly stated, no restrictions to measures which obey the triangle inequality are made. An important distinction can be made between the symmetric TSP and the more general asymmetric TSP. For the symmetric case (normally referred to as just *TSP*), for all distances in $\mathbf{D}$ the equality $d_{ij} = d_{ji}$ holds, i.e., it does not matter if we travel from i to j or the other way round, the distance is the same. In the asymmetric case (called *ATSP*), the distances are not equal for all pairs of cities. Problems of this kind arise when we do not deal with spatial distances between cities but, e.g., with the cost or necessary time associated with traveling between locations, where the price for the plane ticket between two cities may be different depending on which way we go.

2.1. Different formulations of the TSP

Other than the permutation problem in the introduction, the TSP can also be formulated

as a graph theoretic problem. Here the TSP is formulated by means of a complete graph $G = (V, E)$, where the cities correspond to the node set $V = \{1, 2, \ldots, n\}$ and each edge $e_i \in E$ has an associated weight w_i representing the distance between the nodes it connects. If the graph is not complete, the missing edges can be replaced by edges with very large distances. The goal is to find a *Hamiltonian cycle*, i.e., a cycle which visits each node in the graph exactly once, with the least weight in the graph (Hoffman and Wolfe 1985). This formulation naturally leads to procedures involving minimum spanning trees for tour construction or edge exchanges to improve existing tours.

TSPs can also be represented as integer and linear programming problems (see, e.g., Punnen 2002). The *integer programming (IP) formulation* is based on the assignment problem with additional constraint of no sub-tours:

$$\begin{aligned} \text{Minimize} \quad & \textstyle\sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij} \\ \text{Subject to} \quad & \textstyle\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, \ldots, n, \\ & \textstyle\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \ldots, n, \\ & x_{ij} = 0 \text{ or } 1 \\ & \text{no sub-tours allowed} \end{aligned}$$

The solution matrix $\mathbf{X} = (x_{ij})$ of the assignment problem represents a tour or a collection of sub-tour (several unconnected cycles) where only edges which corresponding to elements $x_{ij} = 1$ are on the tour or a sub-tour. The additional restriction that no sub-tours are allowed (called *sub-tour elimination constraints*) restrict the solution to only proper tours. Unfortunately, the number of sub-tour elimination constraints grows exponentially with the number of cities which leads to an extremely hard problem.

The *linear programming (LP) formulation* of the TSP is given by:

$$\begin{aligned} \text{Minimize} \quad & \textstyle\sum_{i=1}^{m} w_i x_i = \mathbf{w}^T \mathbf{x} \\ \text{Subject to} \quad & \mathbf{x} \in \mathcal{S} \end{aligned}$$

where m is the number of edges e_i in G, $w_i \in \mathbf{w}$ is the weight of edge e_i and $\mathbf{x}$ is the incidence vector indicating the presence or absence of each edge in the tour. Again, the constraints given by $\mathbf{x} \in \mathcal{S}$ are problematic since they have to contain the set of incidence vectors of all possible Hamiltonian cycles in G which amounts to a direct search of all $(n-1)!$ possibilities and thus in general is infeasible. However, relaxed versions of the linear programming problem with removed integrality and sub-tour elimination constraints are extensively used by modern TSP solvers where such a partial description of constraints is used and improved iteratively in a branch-and-bound approach.

2.2. Useful manipulations of the distance matrix

Sometimes it is useful to transform the distance matrix $\mathbf{D} = (d_{ij})$ of a TSP into a different matrix $\mathbf{D}' = (d'_{ij})$ which has the same optimal solution. Such a transformation requires that for any Hamiltonian cycle H in a graph represented by its distance matrix $\mathbf{D}$ the equality

$$\sum_{i,j \in H} d_{ij} = \alpha \sum_{i,j \in H} d'_{ij} + \beta$$

holds for suitable $\alpha > 0$ and $\beta \in \mathbb{R}$. From the equality we see that additive and multiplicative constants leave the optimal solution invariant. This property is useful to rescale distances, e.g., for many solvers, distances in the interval $[0, 1]$ have to be converted into integers from 1 to a maximal value.

A different manipulation is to reformulate an asymmetric TSP as a symmetric TSP. This is possible by doubling the number of cities (Jonker and Volgenant 1983). For each city a dummy city is added. Between each city and its corresponding dummy city a very small value (e.g., $-\infty$) is used. This makes sure that each city 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 a very large value (e.g., ∞) which makes these edges infeasible. An example for equivalent formulations as an asymmetric TSP (to the left) and a symmetric TSP (to the right) for three cities is:

$$\begin{pmatrix} 0 & d_{12} & d_{13} \\ d_{21} & 0 & d_{23} \\ d_{31} & d_{32} & 0 \end{pmatrix} \Longleftrightarrow \begin{pmatrix} 0 & \infty & \infty & -\infty & d_{21} & d_{31} \\ \infty & 0 & \infty & d_{12} & -\infty & d_{31} \\ \infty & \infty & 0 & d_{13} & d_{23} & -\infty \\ -\infty & d_{12} & d_{13} & 0 & \infty & \infty \\ d_{21} & -\infty & d_{23} & \infty & 0 & \infty \\ d_{31} & d_{32} & -\infty & \infty & \infty & 0 \end{pmatrix}$$

Instead of the infinity values suitably large negative and positive values can be used. The new symmetric TSP can be solved using techniques for symmetric TSPs which are currently far more advanced than techniques for ATSPs. Removing the dummy cities from the resulting tour gives the solution for the original ATSP.

2.3. Finding exact solutions for the TSP

Finding the exact solution to a TSP with n cities requires to check $(n-1)!$ possible tours. To evaluate all possible tours is infeasible for even small TSP instances. To find the optimal tour Held and Karp (1962) presented the following *dynamic programming* formulation: Given a subset of city indices (excluding the first city) $S \subset \{2, 3, \ldots, n\}$ and $l \in S$, let $d^*(S, l)$ denote the length of the shortest path from city 1 to city l, visiting all cities in S in-between. For $S = \{l\}$, $d^*(S, l)$ is defined as d_{1l}. The shortest path for larger sets with $|S| > 1$ is

$$d^*(S, l) = \min_{m \in S \setminus \{l\}} \Big(d^*(S \setminus \{l\}, m) + d_{ml} \Big). \tag{2}$$

Finally, the minimal tour length for a complete tour which includes returning to city 1 is

$$d^{**} = \min_{l \in \{2,3,\ldots,n\}} \Big(d^*(\{2, 3, \ldots, n\}, l) + d_{l1} \Big). \tag{3}$$

Using the last two equations, the quantities $d^*(S, l)$ can be computed recursively and the minimal tour length d^{**} can be found. In a second step, the optimal permutation $\pi = \{1, i_2, i_3, \ldots, i_n\}$ of city indices 1 through n can be computed in reverse order, starting with

i_n and working successively back to i_2. The procedure exploits the fact that a permutation π can only be optimal, if

$$d^{**} = d^*(\{2, 3, \ldots, n\}, i_n) + d_{i_n 1} \tag{4}$$

and, for $2 \leq p \leq n-1$,

$$d^*(\{i_2, i_3, \ldots, i_p, i_{p+1}\}, i_{p+1}) = d^*(\{i_2, i_3, \ldots, i_p\}, i_p) + d_{i_p i_{p+1}}. \tag{5}$$

The space complexity of storing the values for all $d^*(S, l)$ is $(n-1)2^{n-2}$ which severely restricts the dynamic programming algorithm to TSP problems of small sizes. However, for very small TSP instances this approach is fast and efficient.

A different method, which can deal with larger instances, uses a relaxation of the linear programming problem presented in Section 2.1 and iteratively tightens the relaxation till a solution is found. This general method for solving linear programming problems with complex and large inequality systems is called *cutting plane method* and was introduced by Dantzig, Fulkerson, and Johnson (1954).

Each iteration begins with using instead of the original linear inequality description $\mathcal{S}$ the relaxation $A\mathbf{x} \leq b$, where the polyhedron P defined by the relaxation contains $\mathcal{S}$ and is bounded. The optimal solution $\mathbf{x}^*$ of the relaxed problem can be obtained using standard linear programming solvers. If the $\mathbf{x}^*$ found belongs to $\mathcal{S}$, the optimal solution of the original problem is obtained, otherwise, a linear inequality can be found which is satisfied by all points in $\mathcal{S}$ but violated by $\mathbf{x}^*$. Such an inequality is called a cutting plane or cut. A family of such cutting planes can be added to the inequality system $A\mathbf{x} \leq b$ to obtain a tighter relaxation for the next iteration.

If no further cutting planes can be found or the improvement in the objective function due to adding cuts gets very small, the problem is branched into two sub-problems which can be minimized separately. Branching is done iteratively which leads to a binary tree of sub-problems. Each sub-problem is either solved without further branching or is found to be irrelevant because its relaxed version already produces a longer path than a solution of another sub-problem. This method is called *branch-and-cut* (Padberg and Rinaldi 1990) which is a variation of the well known *branch-and-bound* (Land and Doig 1960) procedure.

The initial polyhedron P used by Dantzig *et al.* (1954) contains all vectors $\mathbf{x}$ for which all $x_e \in \mathbf{x}$ satisfy $0 \leq x_e \leq 1$ and in the resulting tour each city is linked to exactly two other cities. Various separation algorithms for finding subsequent cuts to prevent sub-tours (*sub-tour elimination inequalities*) and to ensure an integer solution (*Gomory cuts;* Gomory 1963) were developed over time. The currently most efficient implementation of this method is **Concorde** described in Applegate, Bixby, Chvátal, and Cook (2000).

2.4. Heuristics for the TSP

The NP-completeness of the TSP already makes it more time efficient for small-to-medium size TSP instances to rely on heuristics in case a good but not necessarily optimal solution is sufficient. TSP heuristics typically fall into two groups, tour construction heuristics which create tours from scratch and tour improvement heuristics which use simple local search heuristics to improve existing tours.

In the following we will only discuss heuristics available in **TSP**, for a comprehensive overview of the multitude of TSP heuristics including an experimental comparison, we refer the reader to the book chapter by Johnson and McGeoch (2002).

Tour construction heuristics

The implemented tour construction heuristics are the nearest neighbor algorithm and the insertion algorithms.

Nearest neighbor algorithm. The nearest neighbor algorithm (Rosenkrantz, Stearns, and Philip M. Lewis 1977) follows a very simple greedy procedure: The algorithm starts with a tour containing a randomly chosen city and then always adds to the last city in the tour the nearest not yet visited city. The algorithm stops when all cities are on the tour.

An extension to this algorithm is to repeat it with each city as the starting point and then return the best tour found. This heuristic is called repetitive nearest neighbor.

Insertion algorithms. All insertion algorithms (Rosenkrantz *et al.* 1977) start with a tour consisting of an arbitrary city and then 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 the insertion cost (i.e., the increase in the tour's length)

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

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

The insertion algorithms differ in the way the city to be inserted next is chosen. The following variations are implemented:

Nearest insertion The city k is chosen in each step as the city which is nearest to a city on the tour.

Farthest insertion The city k is chosen in each step as the city which is farthest from any of the cities on the tour.

Cheapest insertion The city k is chosen in each step such that the cost of inserting the new city is minimal.

Arbitrary insertion The city k is chosen randomly from all cities not yet on the tour.

The nearest and cheapest insertion algorithms correspond to the minimum spanning tree algorithm by Prim (1957). Adding a city to a partial tour corresponds to adding an edge to a partial spanning tree. For TSPs with distances obeying the triangular inequality, the equality to minimum spanning trees provides a theoretical upper bound for the two algorithms of twice the optimal tour length.

The idea behind the farthest insertion algorithm is to link cities far outside into the tour first to establish an outline of the whole tour early. With this change, the algorithm cannot be directly related to generating a minimum spanning tree and thus the upper bound stated above cannot be guaranteed. However, it can was shown that the algorithm generates tours which approach 2/3 times the optimal tour length (Johnson and Papadimitriou 1985b).

Tour improvement heuristics

Tour improvement heuristics are simple local search heuristics which try to improve an initial tour. A comprehensive treatment of the topic can be found in the book chapter by Rego and Glover (2002).

k-Opt heuristics. The idea is to define a neighborhood structure on the set of all admissible tours. Typically, a tour t' is a neighbor of another tour t if t' can be obtained from t by deleting k edges and replacing them by a set of different feasible edges (a k-Opt move). In such a structure, the tour can iteratively be improved by always moving from one tour to its best neighbor till no further improvement is possible. The resulting tour represents a local optimum which is called k-optimal.

Typically, 2-Opt (Croes 1958) and 3-Opt (Lin 1965) heuristics are used in practice.

Lin-Kernighan heuristic. This heuristic (Lin and Kernighan 1973) does not use a fixed value for k for its k-Opt moves, but tries to find the best choice of k for each move. The heuristic uses the fact that each k-Opt move can be represented as a sequence of 2-Opt moves. It builds up a sequence of 2-Opt moves, checking after each additional move whether a stopping rule is met. Then the part of the sequence which gives the best improvement is used. This is equivalent to a choice of one k-Opt move with variable k. Such moves are used till a local optimum is reached.

By using full backtracking, the optimal solution can always be found, but the running time would be immense. Therefore, only limited backtracking is allowed in the procedure, which helps to find better local optima or even the optimal solution. Further improvements to the procedure are described by Lin and Kernighan (1973).

3. Computational infrastructure: The TSP package

In package **TSP**, a traveling salesperson problem is defined by an object of class "`TSP`" (symmetric) or "`ATSP`" (asymmetric). `solve_TSP()` is used to find a solution, which is represented by an object of class "`TOUR`". Figure 1 gives an overview of this infrastructure.

"`TSP`" objects can be created from a distance matrix (a "`dist`" object) or a symmetric matrix using the creator function `TSP()` or coercion with `as.TSP()`. Similarly, "`ATSP`" objects are created by `ATSP()` or `as.ATSP()` from square matrices representing the distances. In the creation process, labels are taken and stored as city names in the object or can be explicitly given as arguments to the creator functions. Several methods are defined for the classes:

- `print()` displays basic information about the problem (number of cities and the distance measure employed).
- `n_of_cities()` returns the number of cities.
- `labels()` returns the city names.
- `image()` produces a shaded matrix plot of the distances between cities. The order of the cities can be specified as the argument `order`.

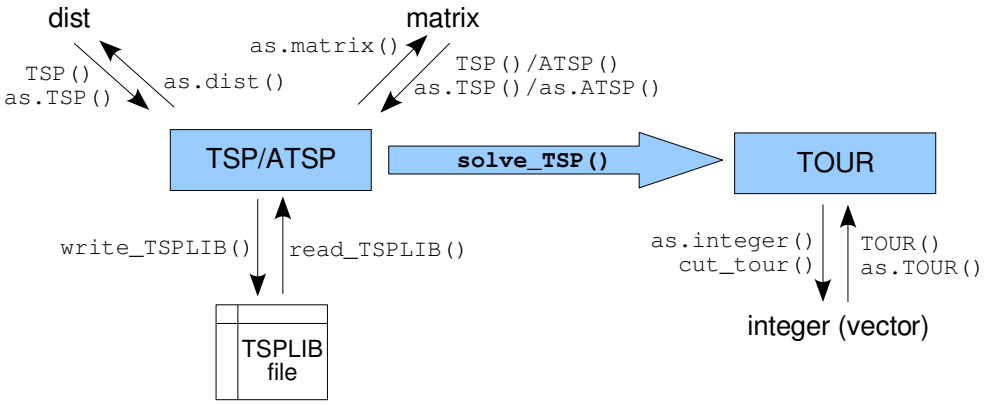

Figure 1: An overview of the classes in **TSP**.

Internally, an object of class "`TSP`" is a "`dist`" object with an additional class attribute and, therefore, if needed, can be coerced to "`dist`" or to a matrix. An "`ATSP`" object is represented as a square matrix. Obviously, asymmetric TSPs are more general than symmetric TSPs, hence, symmetric TSPs can also be represented as asymmetric TSPs. To formulate an asymmetric TSP as a symmetric TSP with double the number of cities (see Section 2.2), `reformulate_ATSP_as_TSP()` is provided. This function creates the necessary dummy cities and adapts the distance matrix accordingly.

A popular format to save TSP descriptions to disk which is supported by most TSP solvers is the format used by *TSPLIB*, a library of sample instances of the TSP maintained by Reinelt (2004). The **TSP** package provides `read_TSPLIB()` and `write_TSPLIB()` to read and save symmetric and asymmetric TSPs in TSPLIB format.

Class "`TOUR`" represents a solution to a TSP by an integer permutation vector containing the ordered indices and labels of the cities to visit. In addition, it stores an attribute indicating the length of the tour. Again, suitable `print()` and `labels()` methods are provided. The raw permutation vector (i.e., the order in which cities are visited) can be obtained from a tour using `as.integer()`. With `cut_tour()`, a circular tour can be split at a specified city resulting in a path represented by a vector of city indices.

The length of a tour can always be calculated using `tour_length()` and specifying a TSP and a tour. Instead of the tour, an integer permutation vector calculated outside the **TSP** package can be used as long as it has the correct length.

All TSP solvers in **TSP** can be used with the simple common interface:

```
solve_TSP(x, method, control)
```

where `x` is the TSP to be solved, `method` is a character string indicating the method used to solve the TSP and `control` can contain a list with additional information used by the solver. The available algorithms are shown in Table 1.

All algorithms except the **Concorde** TSP solver and the Chained Lin-Kernighan heuristic (a Lin-Kernighan variation described in Applegate, Cook, and Rohe (2003)) are included in the package and distributed under the GNU Public License (GPL). For the **Concorde** TSP solver

Algorithm	Method argument	Applicable to
Nearest neighbor algorithm	`"nn"`	TSP/ATSP
Repetitive nearest neighbor algorithm	`"repetitive_nn"`	TSP/ATSP
Nearest insertion	`"nearest_insertion"`	TSP/ATSP
Farthest insertion	`"farthest_insertion"`	TSP/ATSP
Cheapest insertion	`"cheapest_insertion"`	TSP/ATSP
Arbitrary insertion	`"arbitrary_insertion"`	TSP/ATSP
Concorde TSP solver	`"concorde"`	TSP
2-Opt improvement heuristic	`"2-opt"`	TSP/ATSP
Chained Lin-Kernighan	`"linkern"`	TSP

Table 1: Available algorithms in **TSP**.

and the Chained Lin-Kernighan heuristic only a simple interface (using `write_TSPLIB()`, calling the executable and reading back the resulting tour) is included in **TSP**. The executable itself is part of the **Concorde** distribution, has to be installed separately and is governed by a different license which allows only for academic use. The interfaces are included since **Concorde** (Applegate *et al.* 2000; Applegate, Bixby, Chvátal, and Cook 2006) is currently one of the best implementations for solving symmetric TSPs based on the branch-and-cut method discussed in section 2.3. In May 2004, **Concorde** was used to find the optimal solution for the TSP of visiting all 24,978 cities in Sweden. The computation was carried out on a cluster with 96 Xeon 2.8 GHz nodes and took in total almost 100 CPU years.

4. Examples

In this section, we provide some examples for the use of package **TSP**. We start with a simple example of how to use the interface of the TSP solver to compare different heuristics. Then we show how to solve related tasks, using the Hamiltonian shortest path problem as an example. Finally, we give an example of clustering using the **TSP** package. An additional application can be found in package **seriation** (Hahsler, Buchta, and Hornik 2006) which uses the TSP solvers from **TSP** to order (seriate) objects given a proximity matrix.

4.1. Comparing some heuristics

In the following example, we use several heuristics to find a short path in the `USCA50` data set which contains the distances between the first 50 cities in the `USCA312` data set. The `USCA312` data set contains the distances between 312 cities in the USA and Canada coded as a symmetric TSP. The smaller data set is used here, since some of the heuristic solvers employed are rather slow.

```
R> library("TSP")
R> data("USCA50")
R> USCA50

object of class 'TSP'
50 cities (distance 'euclidean')
```

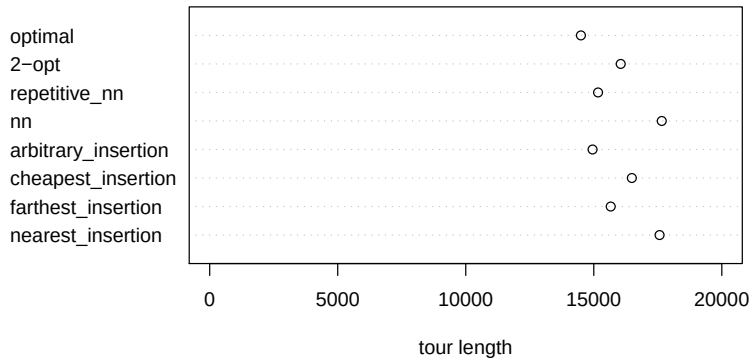

Figure 2: Comparison of the tour lengths for the USCA50 data set.

We calculate tours using different heuristics and store the results in the list `tours`. As an example, we show the first tour which displays the method employed, the number of cities involved and the tour length. All tour lengths are compared using the dot chart in Figure 2. For the chart, we add a point for the optimal solution which has a tour length of 14497. The optimal solution can be found using **Concorde** (`method = "concorde"`). It is omitted here, since **Concorde** has to be installed separately.

```
R> methods <- c("nearest_insertion", "farthest_insertion",
+      "cheapest_insertion", "arbitrary_insertion", "nn", "repetitive_nn",
+      "2-opt")
R> tours <- sapply(methods, FUN = function(m) solve_TSP(USCA50,
+      method = m), simplify = FALSE)
R> tours[[1]]

object of class 'TOUR'
result of method 'nearest_insertion' for 50 cities
tour length: 17421

R> dotchart(c(sapply(tours, FUN = attr, "tour_length"), optimal = 14497),
+      xlab = "tour length", xlim = c(0, 20000))
```

4.2. Finding the shortest Hamiltonian path

The problem of finding the shortest Hamiltonian path through a graph (i.e., a path which visits each node in the graph exactly once) can be transformed into the TSP with cities and distances representing the graphs vertices and edge weights, respectively (Garfinkel 1985).

Finding the shortest Hamiltonian path through all cities disregarding the endpoints can be achieved by inserting a 'dummy city' which has a distance of zero to all other cities. The position of this city in the final tour represents the cutting point for the path. In the following we use a heuristic to find a short path in the `USCA312` data set. Inserting dummy cities is performed in **TSP** by `insert_dummy()`.

```
R> library("TSP")
R> data("USCA312")
R> tsp <- insert_dummy(USCA312, label = "cut")
R> tsp

object of class 'TSP'
313 cities (distance 'euclidean')
```

The TSP now contains an additional dummy city and we can try to solve this TSP.

```
R> tour <- solve_TSP(tsp, method = "farthest_insertion")
R> tour

object of class 'TOUR'
result of method 'farthest_insertion' for 313 cities
tour length: 38184
```

Since the dummy city has distance zero to all other cities, the path length is equal to the tour length reported above. The path starts with the first city in the list after the 'dummy' city and ends with the city right before it. We use `cut_tour()` to create a path and show the first and last 6 cities on it.

```
R> path <- cut_tour(tour, "cut")
R> head(labels(path))

[1] "Lihue, HI"         "Honolulu, HI"      "Hilo, HI"
[4] "San Francisco, CA" "Berkeley, CA"      "Oakland, CA"

R> tail(labels(path))

[1] "Anchorage, AK"     "Fairbanks, AK"     "Dawson, YT"
[4] "Whitehorse, YK"    "Juneau, AK"        "Prince Rupert, BC"
```

The tour found in the example results in a path from Lihue on Hawaii to Prince Rupert in British Columbia. Such a tour can also be visualized using the packages **sp**, **maps** and **maptools** (Pebesma and Bivand 2005).

```
R> library("maps")
R> library("sp")
R> library("maptools")
R> data("USCA312_map")
R> plot_path <- function(path) {
+      plot(as(USCA312_coords, "Spatial"), axes = TRUE)
+      plot(USCA312_basemap, add = TRUE, col = "gray")
+      points(USCA312_coords, pch = 3, cex = 0.4, col = "red")
+      path_line <- SpatialLines(list(Lines(list(Line(USCA312_coords[path,
```

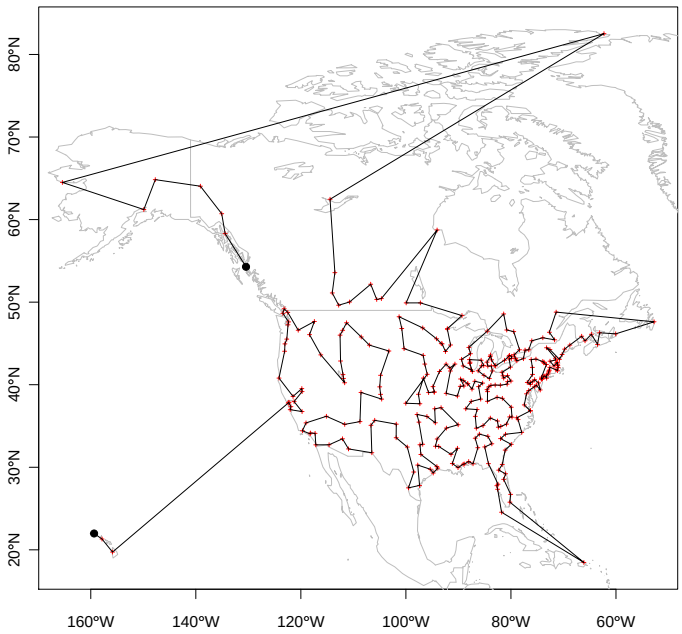

Figure 3: A "short" Hamiltonian path for the USCA312 dataset.

```
+          ])))))
+      plot(path_line, add = TRUE, col = "black")
+      points(USCA312_coords[c(head(path, 1), tail(path, 1)),
+          ], pch = 19, col = "black")
+  }
R> plot_path(path)
```

The map containing the path is presented in Figure 3. It has to be mentioned that the path found by the used heuristic is considerable longer than the optimal path found by **Concorde** with a length of 34928, illustrating the power of modern TSP algorithms.

For the following two examples, we indicate how the distance matrix between cities can be modified to solve related shortest Hamiltonian path problems. These examples serve as illustrations of how modifications can be made to transform different problems into a TSP.

The first problem is to find the shortest Hamiltonian path starting with a given city. In this case, all distances to the selected city are set to zero, forcing the evaluation of all possible paths starting with this city and disregarding the way back from the final city in the tour. By modifying the distances the symmetric TSP is changed into an asymmetric TSP (ATSP) since the distances between the starting city and all other cities are no longer symmetric.

As an example, we choose New York as the starting city. We transform the data set into an ATSP and set the column corresponding to New York to zero before solving it. Thus, the distance to return from the last city in the path to New York does not contribute to the path length. We use the nearest neighbor heuristic to calculate an initial tour which is then improved using 2-Opt moves and cut at New York to create a path.

```
R> atsp <- as.ATSP(USCA312)
R> ny <- which(labels(USCA312) == "New York, NY")
R> atsp[, ny] <- 0
R> initial_tour <- solve_TSP(atsp, method = "nn")
R> initial_tour

object of class 'TOUR'
result of method 'nn' for 312 cities
tour length: 49697

R> tour <- solve_TSP(atsp, method = "2-opt", control = list(tour = initial_tour))
R> tour

object of class 'TOUR'
result of method '2-opt' for 312 cities
tour length: 39445

R> path <- cut_tour(tour, ny, exclude_cut = FALSE)
R> head(labels(path))

[1] "New York, NY"    "Jersey City, NJ" "Elizabeth, NJ"   "Newark, NJ"
[5] "Paterson, NJ"    "Binghamtom, NY"

R> tail(labels(path))

[1] "Edmonton, AB"  "Saskatoon, SK" "Moose Jaw, SK" "Regina, SK"
[5] "Minot, ND"     "Brandon, MB"

R> plot_path(path)
```

The found path is presented in Figure 4. It begins with New York and cities in New Jersey and ends in a city in Manitoba, Canada.

Concorde and many advanced TSP solvers can only solve symmetric TSPs. To use these solvers, we can formulate the ATSP as a TSP using `reformulate_ATSP_as_TSP()` which introduces a dummy city for each city (see Section 2.2).

```
R> tsp <- reformulate_ATSP_as_TSP(atsp)
R> tsp

object of class 'TSP'
624 cities (distance 'unknown')
```

After finding a tour for the TSP, the dummy cities are removed again giving the tour for the original ATSP. Note that the tour needs to be reversed if the dummy cities appear before and not after the original cities in the solution of the TSP. The following code is not executed here, since it takes several minutes to execute and **Concorde** has to be installed separately. **Concorde** finds the optimal solution with a length of 36091.

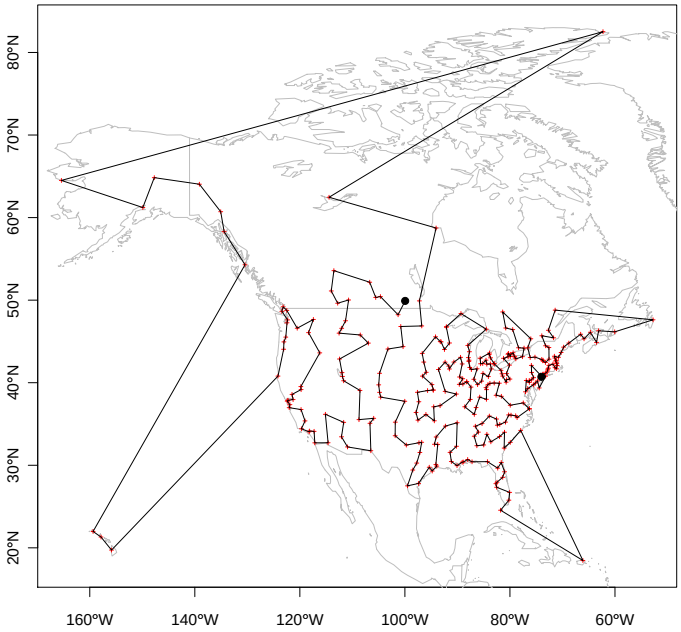

Figure 4: A Hamiltonian path for the USCA312 dataset starting in New York.

```
R> tour <- solve_TSP(tsp, method = "concorde")
R> tour <- as.TOUR(tour[tour <= n_of_cities(atsp)])
```

Finding the shortest Hamiltonian path which ends in a given city can be achieved likewise by setting the row in the distance matrix which corresponds to this city to zero.

For finding the shortest Hamiltonian path we can also restrict both end points. This problem can be transformed to a TSP by replacing the two cities by a single city which contains the distances from the start point in the columns and the distances to the end point in the rows. Obviously this is again an asymmetric TSP.

For the following example, we are only interested in paths starting in New York and ending in Los Angeles. Therefore, we remove the two cities from the distance matrix, create an asymmetric TSP and insert a dummy city called `"LA/NY"`. The distances from this dummy city are replaced by the distances from New York and the distances towards are replaced by the distances towards Los Angeles.

```
R> m <- as.matrix(USCA312)
R> ny <- which(labels(USCA312) == "New York, NY")
R> la <- which(labels(USCA312) == "Los Angeles, CA")
R> atsp <- ATSP(m[-c(ny, la), -c(ny, la)])
R> atsp <- insert_dummy(atsp, label = "LA/NY")
R> la_ny <- which(labels(atsp) == "LA/NY")
R> atsp[la_ny, ] <- c(m[-c(ny, la), ny], 0)
R> atsp[, la_ny] <- c(m[la, -c(ny, la)], 0)
```

We use the nearest insertion heuristic.

```
R> tour <- solve_TSP(atsp, method = "nearest_insertion")
R> tour

object of class 'TOUR'
result of method 'nearest_insertion' for 311 cities
tour length: 45029

R> path_labels <- c("New York, NY", labels(cut_tour(tour, la_ny)),
+      "Los Angeles, CA")
R> path_ids <- match(path_labels, labels(USCA312))
R> head(path_labels)

[1] "New York, NY"        "North Bay, ON"       "Sudbury, ON"
[4] "Timmins, ON"         "Sault Ste Marie, ON" "Thunder Bay, ON"

R> tail(path_labels)

[1] "Eureka, CA"        "Reno, NV"          "Carson City, NV"
[4] "Stockton, CA"      "Santa Barbara, CA" "Los Angeles, CA"

R> plot_path(path_ids)
```

The path jumps from New York to cities in Ontario and it passes through cities in California and Nevada before ending in Los Angeles. The path displayed in Figure 5 contains multiple crossings which indicate that the solution is suboptimal. The optimal solution generated by reformulating the problem as a TSP and using **Concorde** has only a tour length of 38489.

4.3. Rearrangement clustering

Solving a TSP to obtain a clustering was suggested several times in the literature (see, e.g., Lenstra 1974; Alpert and Kahng 1997; Johnson, Krishnan, Chhugani, Kumar, and Venkatasubramanian 2004). The idea is that objects in clusters are visited in consecutive order and from one cluster to the next larger "jumps" are necessary. Climer and Zhang (2006) call this type of clustering *rearrangement clustering* and suggest to automatically find the cluster boundaries of k clusters by adding k *dummy cities* which have constant distance c to all other cities and are infinitely far from each other. In the optimal solution of the TSP, the dummy cities must separate the most distant cities and thus represent optimal boundaries for k clusters.

For the following example, we use the well known iris data set. Since we know that the dataset contains three classes denoted by the variable `Species`, we insert three dummy cities into the TSP for the iris data set and perform rearrangement clustering using the default method (nearest insertion algorithm). Note that this algorithm does not find the optimal solution and it is not guaranteed that the dummy cities will present the best cluster boundaries.

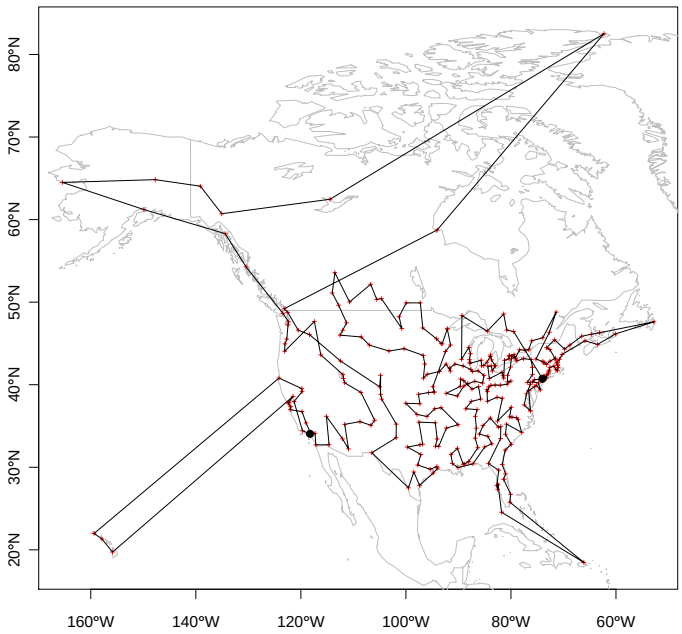

Figure 5: A Hamiltonian path for the USCA312 dataset starting in New York and ending in Los Angles.

```
R> data("iris")
R> tsp <- TSP(dist(iris[-5]), labels = iris[, "Species"])
R> tsp_dummy <- insert_dummy(tsp, n = 3, label = "boundary")
R> tour <- solve_TSP(tsp_dummy)
```

Next, we plot the TSP's permuted distance matrix using shading to represent distances. The result is displayed as Figure 6. Lighter areas represent larger distances. The additional red lines represent the positions of the dummy cities in the tour, which mark the cluster boundaries obtained.

```
R> image(tsp_dummy, tour, xlab = "objects", ylab = "objects")
R> abline(h = which(labels(tour) == "boundary"), col = "red")
R> abline(v = which(labels(tour) == "boundary"), col = "red")
```

One pair of red horizontal and vertical lines exactly separates the darker from lighter areas. The second pair occurs inside the larger dark block. We can look at how well the partitioning obtained fits the structure in the data given by the species field in the data set. Since we used the species as the city labels in the TSP, the labels in the tour represent the partitioning with the dummy cities named 'boundary' separating groups. The result can be summarized based on the run length encoding of the obtained tour labels:

```
R> out <- rle(labels(tour))
R> data.frame(Species = out$values, Lenghts = out$lengths, Pos = cumsum(out$lengths))
```

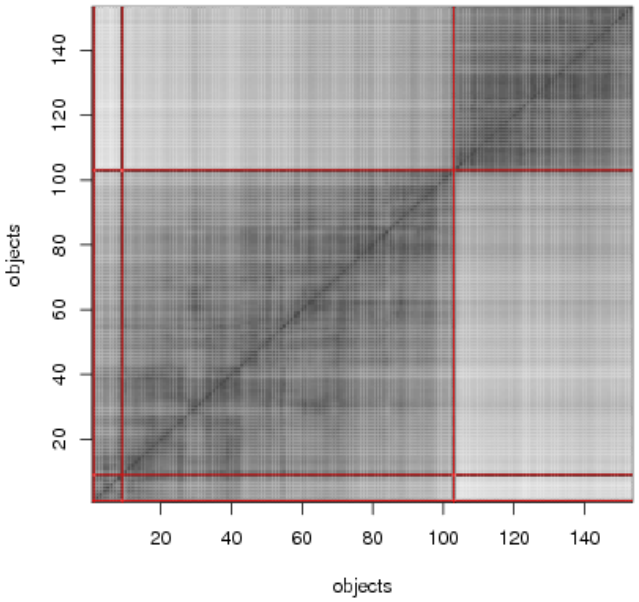

Figure 6: Result of rearrangement clustering using three dummy cities and the nearest insertion algorithm on the iris data set.

```
      Species Lenghts Pos
1    boundary       1   1
2   virginica       7   8
3    boundary       1   9
4   virginica      18  27
5  versicolor       5  32
6   virginica      20  52
7  versicolor       1  53
8   virginica       3  56
9  versicolor      13  69
10  virginica       1  70
11 versicolor      13  83
12  virginica       1  84
13 versicolor      18 102
14   boundary       1 103
15     setosa      50 153
```

One boundary perfectly splits the iris data set into a group containing only examples of species 'Setosa' and a second group containing examples for 'Virginica' and 'Versicolor'. However, the second boundary only separates several examples of species 'Virginica' from other examples of the same species. Even in the optimal tour found by **Concorde**, this problem occurs. The reason why the rearrangement clustering fails to split the data into three groups is the closeness between the groups 'Virginica' and 'Versicolor'. To inspect this problem further, we can project the data points on the first two principal components of the data set and add the

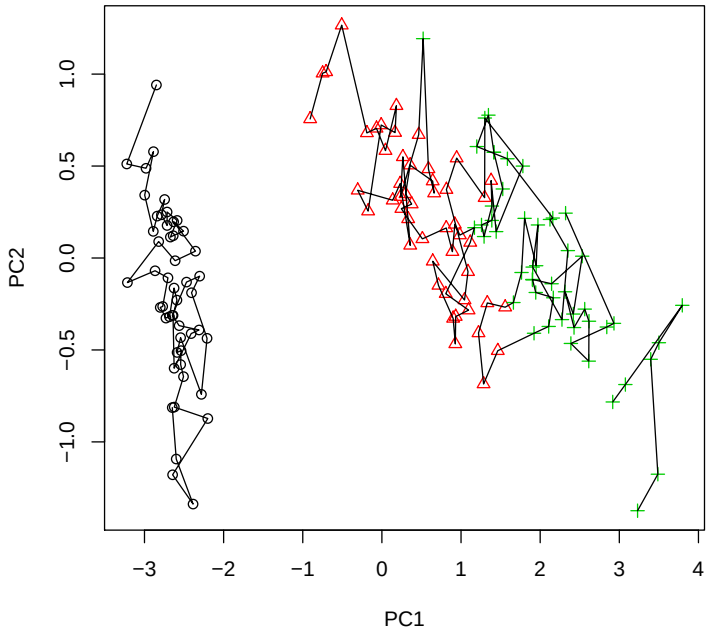

Figure 7: The 3 path segments representing a rearrangement clustering of the iris data set. The data points are projected on the set's first two principal components. The three species are represented by different markers and colors.

path segments which resulted from solving the TSP.

```
R> prc <- prcomp(iris[1:4])
R> plot(prc$x, pch = as.numeric(iris[, 5]), col = as.numeric(iris[,
+      5]))
R> indices <- c(tour, tour[1])
R> indices[indices > 150] <- NA
R> lines(prc$x[indices, ])
```

The result in shown in Figure 7. The three species are identified by different markers and all points connected by a single path represent a cluster found. Clearly, the two groups to the right side of the plot are too close to be separated correctly by using just the distances between individual points. This problem is similar to the *chaining effect* known from hierarchical clustering using the single-linkage method.

5. Conclusion

In this paper, we presented the R extension package **TSP** which implements an infrastructure to handle and solve TSPs. The package introduces classes for problem descriptions ("`TSP`" and "`ATSP`") and for the solution ("`TOUR`"). Together with a simple interface for solving TSPs, it allows for an easy and transparent usage of the package.

With the interface to **Concorde**, **TSP** also can use a state of the art implementation which efficiently computes exact solutions using branch-and-cut.

Acknowledgments

The authors of this paper want to thank Roger Bivand for providing the code to correctly draw tours and paths on a projected map.

References

Alpert CJ, Kahng AB (1997). "Splitting an Ordering into a Partititon to Minimize Diameter." *Journal of Classification*, **14**(1), 51–74.

Applegate D, Bixby R, Chvátal V, Cook W (2006). **Concorde** *TSP Solver*. URL http://www.tsp.gatech.edu/concorde/.

Applegate D, Bixby RE, Chvátal V, Cook W (2000). "TSP Cuts Which Do Not Conform to the Template Paradigm." In M Junger, D Naddef (eds.), "Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions," volume 2241 of *Lecture Notes In Computer Science*, pp. 261–304. Springer-Verlag, London, UK.

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

Climer S, Zhang W (2006). "Rearrangement Clustering: Pitfalls, Remedies, and Applications." *Journal of Machine Learning Research*, **7**, 919–943.

Croes GA (1958). "A Method for Solving Traveling-Salesman Problems." *Operations Research*, **6**(6), 791–812.

Dantzig GB, Fulkerson DR, Johnson SM (1954). "Solution of a Large-scale Traveling Salesman Problem." *Operations Research*, **2**, 393–410.

Garfinkel RS (1985). "Motivation and Modeling." In Lawler *et al.* (1985), chapter 2, pp. 17–36.

Gomory RE (1963). "An Algorithm for Integer Solutions to Linear Programs." In R Graves, P Wolfe (eds.), "Recent Advances in Mathematical Programming," pp. 269–302. McGraw-Hill, New York.

Gutin G, Punnen AP (eds.) (2002). *The Traveling Salesman Problem and Its Variations*, volume 12 of *Combinatorial Optimization*. Kluwer, Dordrecht.

Hahsler M, Buchta C, Hornik K (2006). **seriation***: Infrastructure for Seriation*. R package version 0.1-1, URL http://CRAN.R-project.org/.

Held M, Karp RM (1962). "A Dynamic Programming Approach to Sequencing Problems." *Journal of SIAM*, **10**, 196–210.

Hoffman AJ, Wolfe P (1985). "History." In Lawler *et al.* (1985), chapter 1, pp. 1–16.

Hubert LJ, Baker FB (1978). "Applications of Combinatorial Programming to Data Analysis: The Traveling Salesman and Related Problems." *Psychometrika*, **43**(1), 81–91.

Johnson D, Krishnan S, Chhugani J, Kumar S, Venkatasubramanian S (2004). "Compressing Large Boolean Matrices Using Reordering Techniques." In "Proceedings of the 30th VLDB Conference," pp. 13–23.

Johnson DS, McGeoch LA (2002). "Experimental Analysis of Heuristics for the STSP." In Gutin and Punnen (2002), chapter 9, pp. 369–444.

Johnson DS, Papadimitriou CH (1985a). "Computational Complexity." In Lawler *et al.* (1985), chapter 3, pp. 37–86.

Johnson DS, Papadimitriou CH (1985b). "Performance Guarantees for Heuristics." In Lawler *et al.* (1985), chapter 5, pp. 145–180.

Johnson O, Liu J (2006). "A Traveling Salesman Approach for Predicting Protein Functions." *Source Code for Biology and Medicine*, **1**(3), 1–7.

Jonker R, Volgenant T (1983). "Transforming Asymmetric into Symmetric Traveling Salesman Problems." *Operations Research Letters*, **2**, 161–163.

Land AH, Doig AG (1960). "An Automatic Method for Solving Discrete Programming Problems." *Econometrica*, **28**, 497–520.

Lawler EL, Lenstra JK, Rinnooy Kan AHG, Shmoys DB (eds.) (1985). *The Traveling Salesman Problem*. Wiley, New York.

Lenstra JK (1974). "Clustering a Data Array and the Traveling-Salesman Problem." *Operations Research*, **22**(2), 413–414.

Lenstra JK, Kan AHGR (1975). "Some Simple Applications of the Travelling Salesman Problem." *Operational Research Quarterly*, **26**(4), 717–733.

Lin S (1965). "Computer Solutions of the Traveling-Salesman Problem." *Bell System Technology Journal*, **44**, 2245–2269.

Lin S, Kernighan BW (1973). "An Effective Heuristic Algorithm for the Traveling-Salesman Problem." *Operations Research*, **21**(2), 498–516.

Padberg M, Rinaldi G (1990). "Facet Identification for the Symmetric Traveling Salesman Polytope." *Mathematical Programming*, **47**(2), 219–257. ISSN 0025-5610.

Pebesma EJ, Bivand RS (2005). "Classes and Methods for Spatial Data in R." *R News*, **5**(2), 9–13. URL http://CRAN.R-project.org/doc/Rnews/.

Prim RC (1957). "Shortest Connection Networks and Some Generalisations." *Bell System Technical Journal*, **36**, 1389–1401.

Punnen AP (2002). "The Traveling Salesman Problem: Applications, Formulations and Variations." In Gutin and Punnen (2002), chapter 1, pp. 1–28.

Ray SS, Bandyopadhyay S, Pal SK (2007). "Gene Ordering in Partitive Clustering using Microarray Expressions." *Journal of Biosciences*, **32**(5), 1019–1025.

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Rego C, Glover F (2002). "Local Search and Metaheuristics." In Gutin and Punnen (2002), chapter 8, pp. 309–368.

Reinelt G (2004). *TSPLIB*. Universität Heidelberg, Institut für Informatik, Heidelberg, Germany. URL http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

Rosenkrantz DJ, Stearns RE, Philip M Lewis I (1977). "An Analysis of Several Heuristics for the Traveling Salesman Problem." *SIAM Journal on Computing*, **6**(3), 563–581.

Affiliation:

Michael Hahsler
Department für Informationsverarbeitung und Prozessmanagement
Wirtschaftsuniversität Wien
Augasse 2–6
A-1090 Wien, Austria
E-mail: Michael.Hahsler@wu-wien.ac.at
URL: http://wwwai.wu-wien.ac.at/~hahsler/

Kurt Hornik
Department für Statistik & Mathematik
Wirtschaftsuniversität Wien
Augasse 2–6
A-1090 Wien, Austria
E-mail: Kurt.Hornik@wu-wien.ac.at
URL: http://statmath.wu-wien.ac.at/~hornik/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 23, Issue 2 *Submitted:* 2006-12-21
December 2007 *Accepted:* 2007-11-24