Introduction to **TSP** – Infrastructure for the Traveling Salesperson Problem

Michael Hahsler and Kurt Hornik

December 21, 2006

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. In addition to vehicle routing, many other applications, e.g., computer wiring, cutting wallpaper, job sequencing or several data visualization techniques, require the solution of a TSP.

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.

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, 1985b). 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 and structuring of matrices (Lenstra and Kan, 1975; Punnen, 2002).

In this paper we give a very brief overview of the TSP and introduce the R package **TSP** which provides a infrastructure for handling and solving TSPs in R. 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 packages 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 exchangeably 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 needed time associated with traveling between locations. Here 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 regarded as 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 every 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 constraints described as follows:

$$\begin{array}{ll}
\text{Minimize} & \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij} \\
\text{Subject to} & \\
& \sum_{i=1}^{n} x_{ij} = 1, j = 1, \cdots, n, \\
& \sum_{j=1}^{n} x_{ij} = 1, i = 1, \cdots, n, \\
& x_{ij} = 0 \text{ or } 1 \\
& \mathbf{X} \text{ contains no subtours}
\end{array}$$

The solution matrix $\mathbf{X} = (x_{ij})$ of the assignment problem represents a tour or a collection of subtour (several unconnected cycles) where only edges which corresponding to elements $x_{ij} = 1$ are on the tour or a subtour. The additional restriction that solution contains no subtours are called *subtour elimination constraints*). Unfortunately, the number of subtour 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:

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

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 subtour 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 a 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

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

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

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

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 the 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 found using standard linear programming solvers. If the found $\mathbf{x}^*$ belongs to $\mathcal{S}$, the optimal solution of the original problem is found, otherwise, a linear inequality can be found which satisfies all points in $\mathcal{S}$ but violates $\mathbf{x}^*$. 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 get 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 subproblems which can be minimized separately. Branching is done iteratively which leads to a binary tree of subproblems. Each subproblem 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 subproblem. 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 subtours (*subtour elimination inequalities*) and to ensure that $\mathbf{x}^*$ is an integer vector (*Gomory cuts;* Gomory, 1963) where developed over time. The currently most efficient implementation of this method is 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 medium sized TSP instances to rely on heuristics if a good but not necessarily optimal solution suffices. 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).

2.4.1 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 of the found tours. 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 stops 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 to 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 fist 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, 1985a).

2.4.2 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 tour 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 be iteratively 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, if 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).

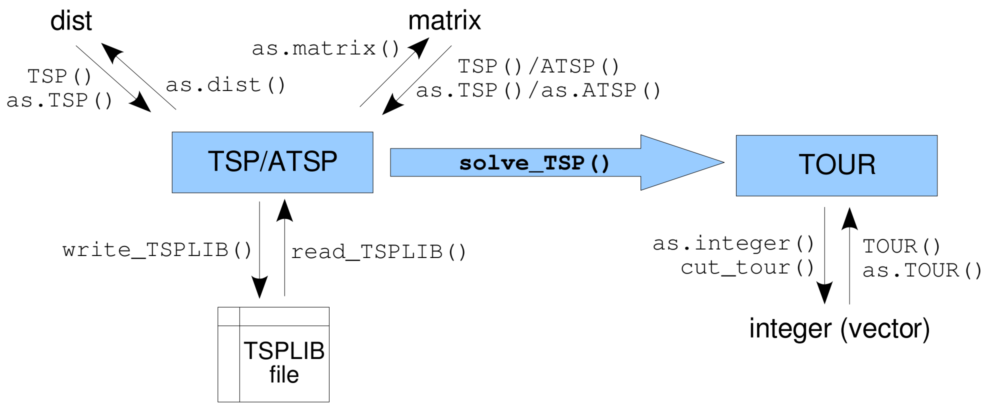

Figure 1: An overview of the classes in **TSP**.

3 Computational infrastructure: the TSP package

In the 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 a 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 used distance measure).
- `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`.

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. The 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.

The class TOUR represents a solution to a TSP in form of 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** use the simple common interface:

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

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

```
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 and the Chained Lin-Kernighan heuristic only a simple interface (using `write_TSPLIB()`, calling the executable and reads back the resulting tour) is included in **TSP**. The code 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, Chvatal, 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 nodes and took in total almost 100 CPU years (assuming a single CPU Xeon 2.8 GHz processor).

4 Examples

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.

```
> library("TSP")
> data("USCA50")
> tsp <- USCA50
> tsp
```

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

We calculate tours using different heuristics and store the results in the the list `tours`. As an example, we show the first tour which displays the used method, 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.

```
> methods <- c("nearest_insertion", "farthest_insertion", "cheapest_insertion",
+     "arbitrary_insertion", "nn", "repetitive_nn", "2-opt")
```

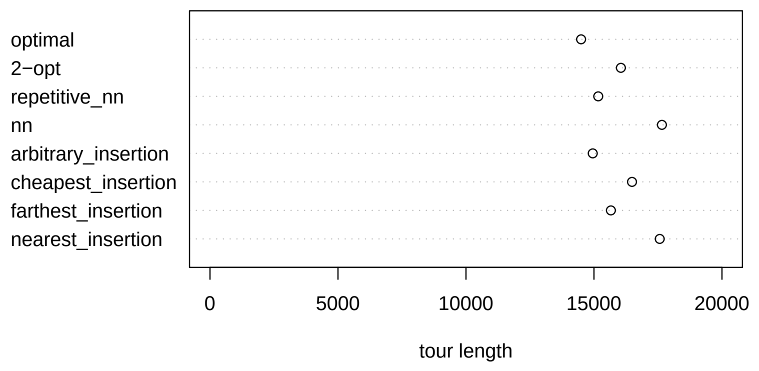

Figure 2: Comparison of the tour lengths for the USCA50 data set.

```
> tours <- lapply(methods, FUN = function(m) solve_TSP(tsp,
+     method = m))
> names(tours) <- methods
> tours[[1]]
```

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

```
> opt <- 14497
> dotchart(c(sapply(tours, FUN = attr, "tour_length"), optimal = opt),
+     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 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 implemented in **TSP** as `insert_dummy()`.

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

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

The TSP contains now an additional dummy city and we can try to solve this TSP and print the labels to see the resulting tour.

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

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

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

```
> 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**.

```
> library("maps")
> library("sp")
> library("maptools")
> data("USCA312_map")
> 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,
+         ])))))
+     plot(path_line, add = TRUE, col = "black")
+     points(USCA312_coords[c(head(path, 1), tail(path, 1)),
+         ], pch = 19, col = "black")
+ }
> 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 show in a very low level way 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 the city New York to be the starting city. We transform the data set into an ATSP and set the column corresponding to New York to zero before solving it. This means that 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 City to create a path.

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

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

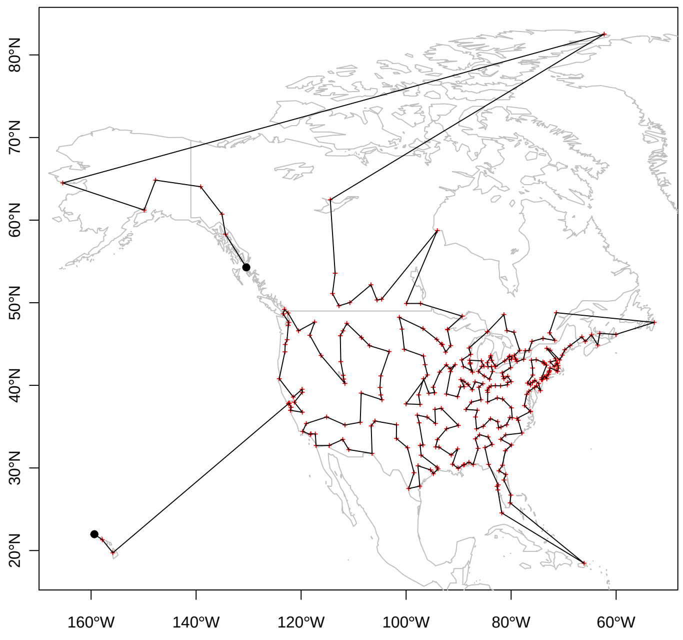

Figure 3: A "short" Hamiltonian path for the USCA312 dataset.

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

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

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

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

```
> tail(labels(path))
```

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

```
> plot_path(path)
```

The found path is presented in Figure 4. It begins with New York City 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 for each city a dummy city (see Section 2.2).

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

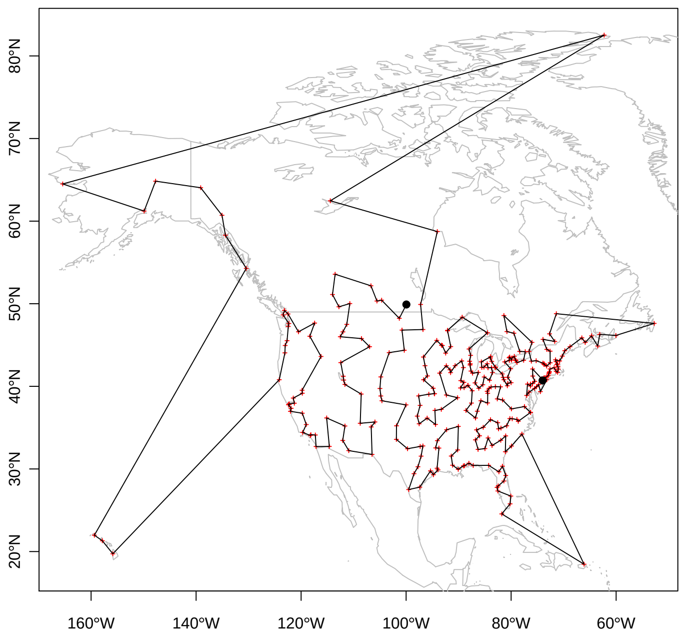

Figure 4: A Hamiltonian path for the USCA312 dataset starting in New York City.

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.

```
> tour <- solve_TSP(tsp, method = "concorde")
> 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 corresponding to this city in the distance matrix 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.

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

We use again the nearest insertion heuristic.

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

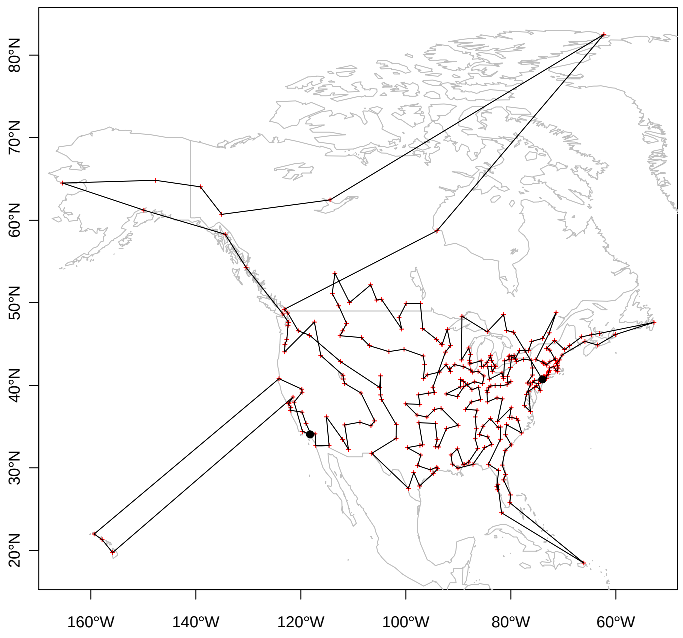

Figure 5: A Hamiltonian path for the USCA312 dataset starting in New York City and ending in Los Angles.

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

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

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

```
> tail(path_labels)
```

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

```
> plot_path(path_ids)
```

The path moves from New York on to other cities close in the State of New York and it passes through cities in California before ending in Los Angeles. The whole path is displayed in Figure 5.

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. Climer and Zhang (2006) show that in the

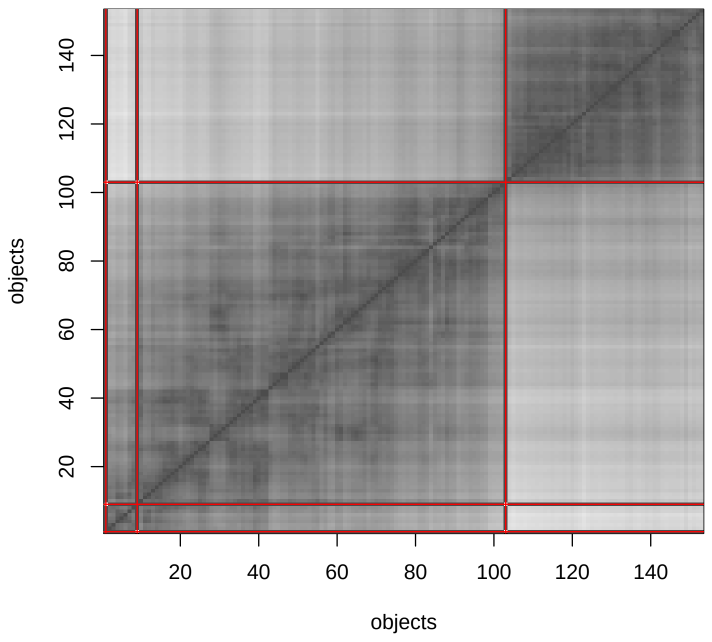

Figure 6: Result of rearrangement clustering using three dummy cities and the nearest insertion algorithm on the iris data set.

optimal solution of the TSP, the dummy cities must separate the most distant cities and thus represent optimal boundaries for k clusters.

For the example, we use the well known iris data set. Since we know that the dataset contains three classes denoted by the attribute called `"Species"`, we insert three dummy cities into the TSP for the iris data set and perform rearrangement clustering using the 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 optimal cluster boundaries.

```
> data("iris")
> tsp <- TSP(dist(iris[-5]), labels = iris[, "Species"])
> tsp_dummy <- insert_dummy(tsp, n = 3, label = "boundary")
> 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 found boundaries of the clusters.

```
> image(tsp_dummy, tour, xlab = "objects", ylab = "objects")
> abline(h = which(labels(tour) == "boundary"), col = "red")
> 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. We can look at how well the found partitioning 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.

```
> labels(tour)

  [1] "boundary"   "virginica"  "virginica"  "virginica"  "virginica"
  [6] "virginica"  "virginica"  "virginica"  "boundary"   "virginica"
```

```
 [11] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
 [16] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
 [21] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
 [26] "virginica"  "virginica"  "versicolor" "versicolor" "versicolor"
 [31] "versicolor" "versicolor" "virginica"  "virginica"  "virginica"
 [36] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
 [41] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
 [46] "virginica"  "virginica"  "virginica"  "virginica"  "virginica"
 [51] "virginica"  "virginica"  "versicolor" "virginica"  "virginica"
 [56] "virginica"  "versicolor" "versicolor" "versicolor" "versicolor"
 [61] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
 [66] "versicolor" "versicolor" "versicolor" "versicolor" "virginica"
 [71] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
 [76] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
 [81] "versicolor" "versicolor" "versicolor" "virginica"  "versicolor"
 [86] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
 [91] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
 [96] "versicolor" "versicolor" "versicolor" "versicolor" "versicolor"
[101] "versicolor" "versicolor" "boundary"   "setosa"     "setosa"
[106] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[111] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[116] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[121] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[126] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[131] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[136] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[141] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[146] "setosa"     "setosa"     "setosa"     "setosa"     "setosa"
[151] "setosa"     "setosa"     "setosa"
```

One boundary perfectly splits the iris data set into a group containing only examples of the species 'Setosa' and a second group containing examples for 'Virginica' and 'Versicolor'. However, the second boundary only separates several examples of the 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 path segments which resulted from solving the TSP.

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

The result in shown in Figure 7. The three true groups are identified by different markers and all points connected by a single path represent a found cluster. 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 package **TSP** which implements the 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.

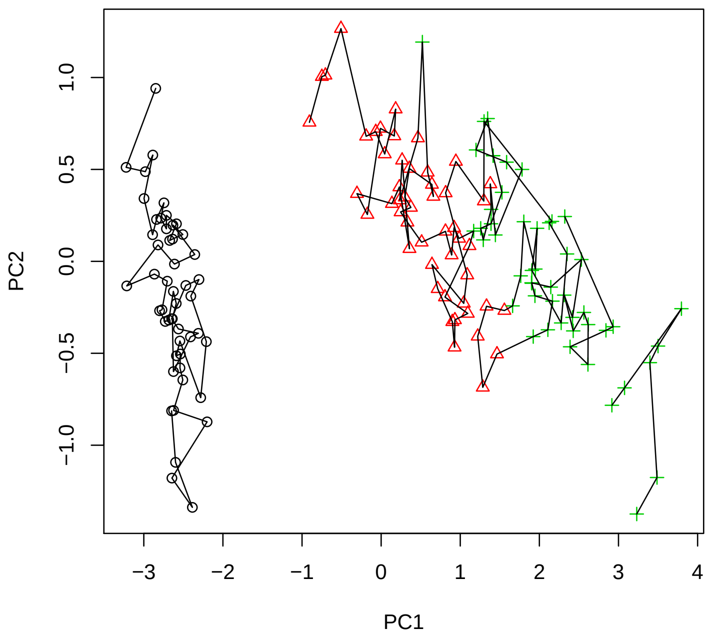

Figure 7: The 3 path segments representing a rearrangement clustering of the iris data set. The data points are projected on the set's first two principal components. The three species are represented by different markers and colors.

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

C. J. Alpert and A. B. Kahng. Splitting an ordering into a partititon to minimize diameter. *Journal of Classification*, 14(1):51–74, 1997.

D. Applegate, R. E. Bixby, V. Chvátal, and W. Cook. TSP cuts which do not conform to the template paradigm. In M. Junger and D. Naddef, editors, *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions*, volume 2241 of *Lecture Notes In Computer Science*, pages 261–304, London, UK, 2000. Springer-Verlag.

D. Applegate, W. Cook, and A. Rohe. Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92, 2003.

D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *Concorde TSP Solver*, 2006. URL `http://www.tsp.gatech.edu/concorde/`.

S. Climer and W. Zhang. Rearrangement clustering: Pitfalls, remedies, and applications. *Journal of Machine Learning Research*, 7:919–943, June 2006.

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

G. Dantzig, D. Fulkerson, and S. Johnson. Solution of a large-scale traveling salesman problem. *Operations Research*, 2:393–410, 1954.

R. Garfinkel. Motivation and modeling. In Lawler et al. (1985), chapter 2, pages 17–36.

R. Gomory. An algorithm for integer solutions to linear programs. In R. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302, New York, 1963. McGraw-Hill.

G. Gutin and A. Punnen, editors. *The Traveling Salesman Problem and Its Variations*, volume 12 of *Combinatorial Optimization*. Kluwer, Dordrecht, 2002.

M. Held and R. Karp. A dynamic programming approach to sequencing problems. *Journal of SIAM*, 10:196–210, 1962.

A. Hoffman and P. Wolfe. History. In Lawler et al. (1985), chapter 1, pages 1–16.

D. Johnson and L. McGeoch. Experimental analysis of heuristics for the STSP. In Gutin and Punnen (2002), chapter 9, pages 369–444.

D. Johnson and C. Papadimitriou. Performance guarantees for heuristics. In Lawler et al. (1985), chapter 5, pages 145–180.

D. Johnson and C. Papadimitriou. Computational complexity. In Lawler et al. (1985), chapter 3, pages 37–86.

D. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. Compressing large boolean matrices using reordering techniques. In *Proceedings of the 30th VLDB Conference*, pages 13–23, 2004.

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

A. Land and A. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem*. Wiley, New York, 1985.

J. Lenstra and A. R. Kan. Some simple applications of the travelling salesman problem. *Operational Research Quarterly*, 26(4):717–733, November 1975.

J. K. Lenstra. Clustering a data array and the traveling-salesman problem. *Operations Research*, 22(2):413–414, 1974.

S. Lin. Computer solutions of the traveling-salesman problem. *Bell System Technology Journal*, 44:2245–2269, 1965.

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

M. Padberg and G. Rinaldi. Facet identification for the symmetric traveling salesman polytope. *Mathematical Programming*, 47(2):219–257, 1990. ISSN 0025-5610.

R. Prim. Shortest connection networks and some generalisations. *Bell System Technical Journal*, 36:1389–1401, 1957.

A. Punnen. The traveling salesman problem: Applications, formulations and variations. In Gutin and Punnen (2002), chapter 1, pages 1–28.

C. Rego and F. Glover. Local search and metaheuristics. In Gutin and Punnen (2002), chapter 8, pages 309–368.

G. Reinelt. *TSPLIB*. Universität Heidelberg, Institut für Informatik, Im Neuenheimer Feld 368,D-69120 Heidelberg, Germany, 2004. URL `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`.

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