TSP – A R-Package for the Traveling Salesperson Problem

Michael Hahsler and Kurt Hornik

Vienna University of Economics and Business Administration

Research Meeting, Vienna, December 1, 2006

Agenda

1. Introduction
2. Definition and types of TSPs
3. Basic infrastructure in R
4. Algorithms and Heuristics
5. Some examples

Introduction

The traveling salesperson problem

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.

The TSP has many applications including (Lenstra and Kan, 1975)

- computer wiring
- vehicle routing
- clustering of data arrays
- machine scheduling

Definition

Formally, the TSP can be stated as: The distances between n cities are stored in a distance matrix D with elements $d(i, j)$ where $i, j = 1 \dots n$ and the diagonal elements $d(i, i) = 0$.

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 to find a permutation π that minimizes

$$\sum_{i=1}^{n} d(i, \pi(i)), \tag{1}$$

which is called the ***length of the tour.***

In terms of graph theory, cities can be regarded as vertices in a complete, weighted graph. The edge weights represent the distances between the cities. The goal is to find a ***Hamiltonian cycle*** with the least weight (Hoffman and Wolfe, 1985).

Alternative representations

- **Linear programming representation** – complete linear inequality structure unknown
- **Integer programming formulation** – assignment problem + subtour elimination constraints $\Rightarrow$ use LP as a relaxation
- **Binary quadratic programming formulation** – TSP is equivalent to a 0-1 quadratic programming problem

Some types of TSPs

Symmetric TSP with a symmetric distance matrix: $d(i, j) = d(j, i)$

Asymmetric TSP – the distances are not equal for all pairs of cities. Arises when instead of spatial distances cost or time is used to form D.

Euclidean TSP with euclidean distances.

TSP with triangle inequality where $d(i, j) + d(j, k) \geq d(i, k)$.

Some algorithms and heuristics

Algorithms for exact solution

- Dynamic programming for small instances (Held and Karp, 1962).
- Branch-and bound, branch-and-cut,. . .

Heuristics

1. Tour construction

   (a) Choose initial tour (e.g., random city, convex hull for Euclidean TSP)

   (b) Selection method (nearest, farthest, arbitrary,. . .)

   (c) Insertion method (minimum cost, greatest angle)

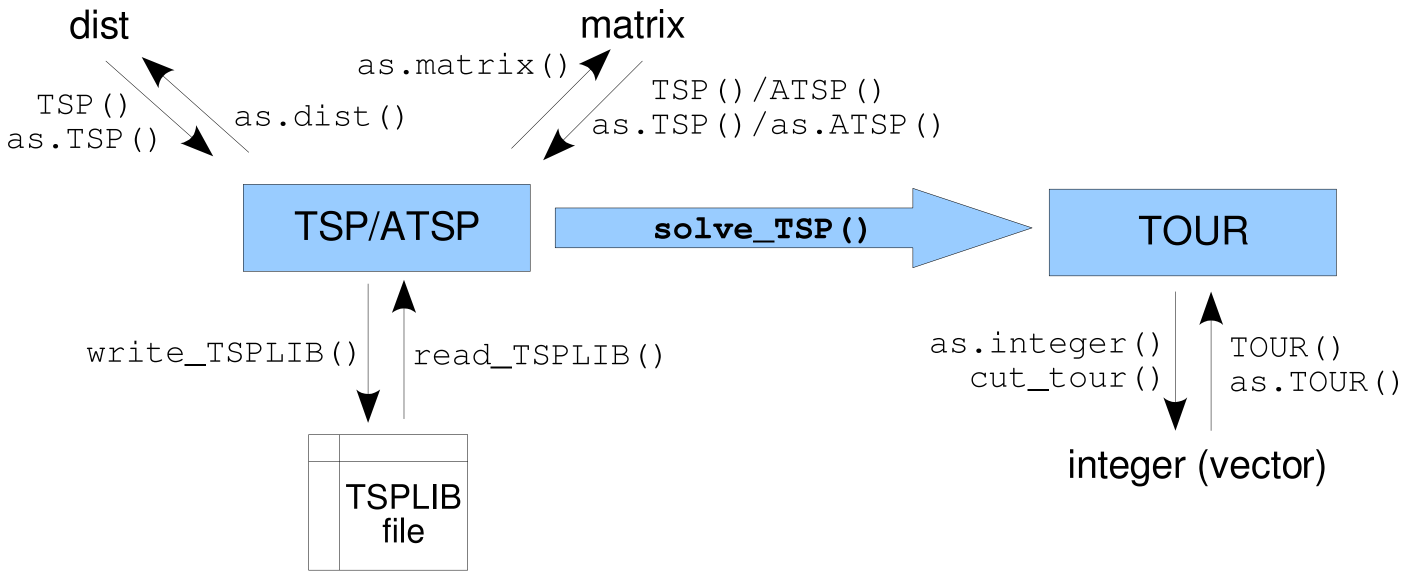
   or solve assignment problem w/linear programming + patching

2. Tour improvement

   - Edge exchange procedures: 2-opt (Croes, 1958), 3-opt (Lin, 1965), k-opt, Or-opt
   - Variable r-opt (Lin and Kernighan, 1973)

The TSP package

Overview



Methods for TSP/ATSP: `print()`, `n_of_cities()`, `labels()`, `image()`.

Methods for TOUR: `print()`, `labels()`, `cut_tour()`, `tour_length()`.

Solving (A)TSPs

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.

Currently available algorithms

The NP-completeness of the TSP makes it already for medium sized TSP instances necessary to resort to heuristics. In **TSP**, we implemented some simple heuristics described by Rosenkrantz, Stearns, and Philip M. Lewis (1977):

- Nearest neighbor algorithm
- Some variants of the insertion algorithm

The package also provides an interface to the ***Concorde TSP solver*** (Applegate, Bixby, Chvátal, and Cook, 2000).

Nearest neighbor algorithm

The nearest neighbor algorithm (Rosenkrantz et al., 1977) follows a very simple greedy procedure: The algorithm starts with a tour containing a randomly chosen city. Then the algorithm 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. This algorithm is implemented as method `"nn"` for `solve_TSP()`.

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 algorithm is implemented as method `"repetitive_nn"`.

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

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

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:

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 (i.e., the increase in the tour's length) is minimal.

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

Methods: `"nearest_insertion"`, `"farthest_insertion"`, `"cheapest_insertion"`, `"arbitrary_insertion"`

Some properties of the insertion algorithms

The nearest and cheapest insertion algorithms are variants of to the ***minimum spanning tree algorithm*** which is known to be a good algorithm to find a Hamiltonian cycle in a connected, undirected graph with a close to minimal weight sum. For nearest and cheapest insertion, 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 upper bound for the length of the tour found by the minimum spanning tree algorithm is ***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 1.5 ***times the optimal tour length*** (Johnson and Papadimitriou, 1985).

Concorde

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.

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).

TSP provides a simple interface to Concorde which is used for method `"concorde"`. It saves the TSP using `write_TSPLIB()` and then calls the Concorde executable and reads back the resulting tour.

Examples

Comparing some heuristics

`USCA50` contains 50 cities in the USA and Canada as a TSP.

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

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

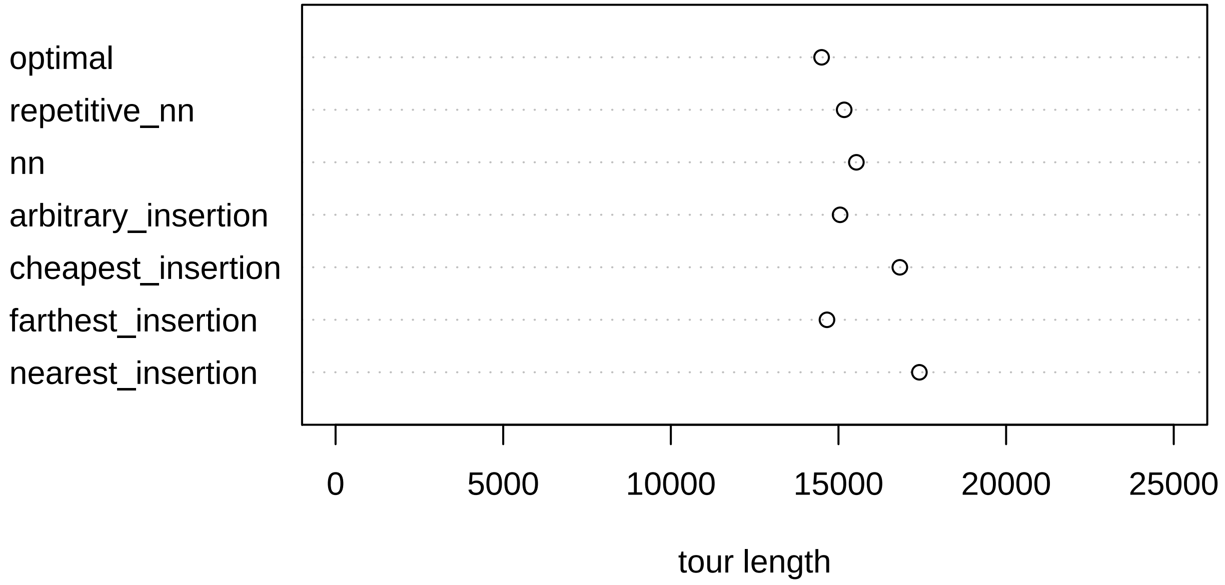
Calculate tours:

```
> methods <- c("nearest_insertion", "farthest_insertion",
+     "cheapest_insertion", "arbitrary_insertion", "nn",
+     "repetitive_nn")
> 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: 17413
```

Comparing some heuristics (cont.)

```
> opt <- 14497
> dotchart(c(sapply(tours, FUN = attr, "tour_length"),
+     optimal = opt), xlab = "tour length", xlim = c(0,
+     25000))
```



Hamiltonian paths

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.

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

object of class 'TOUR'
result of method 'nearest_insertion' for 313 cities
tour length: 38539
```

The path length is the tour length. The optimal length is 34928 (using Concorde).

Hamiltonian paths (cont.)

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

```
[1] "Alert, NT"       "Yellowknife, NT" "Dawson, YT"
[4] "Fairbanks, AK"   "Nome, AK"        "Anchorage, AK"
```

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

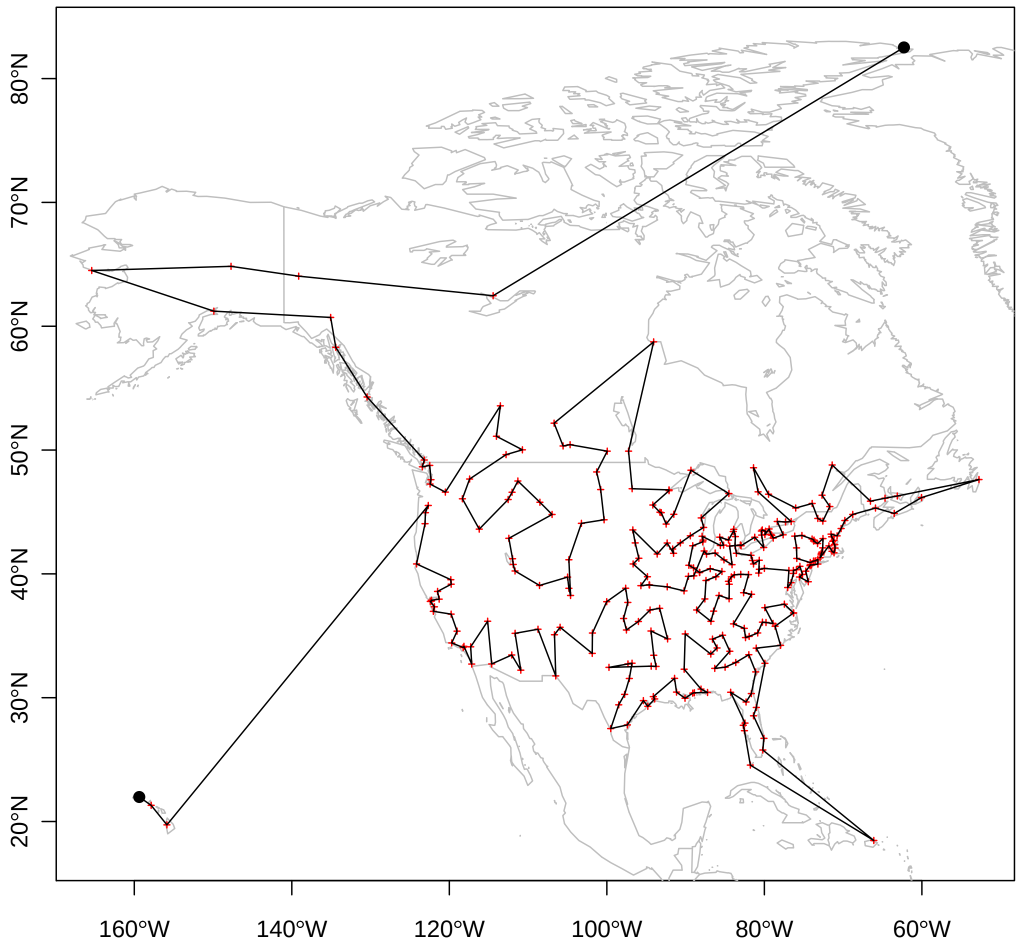
```
[1] "Eugene, OR"   "Salem, OR"    "Portland, OR" "Hilo, HI"
[5] "Honolulu, HI" "Lihue, HI"
```

Hamiltonian paths (cont.)

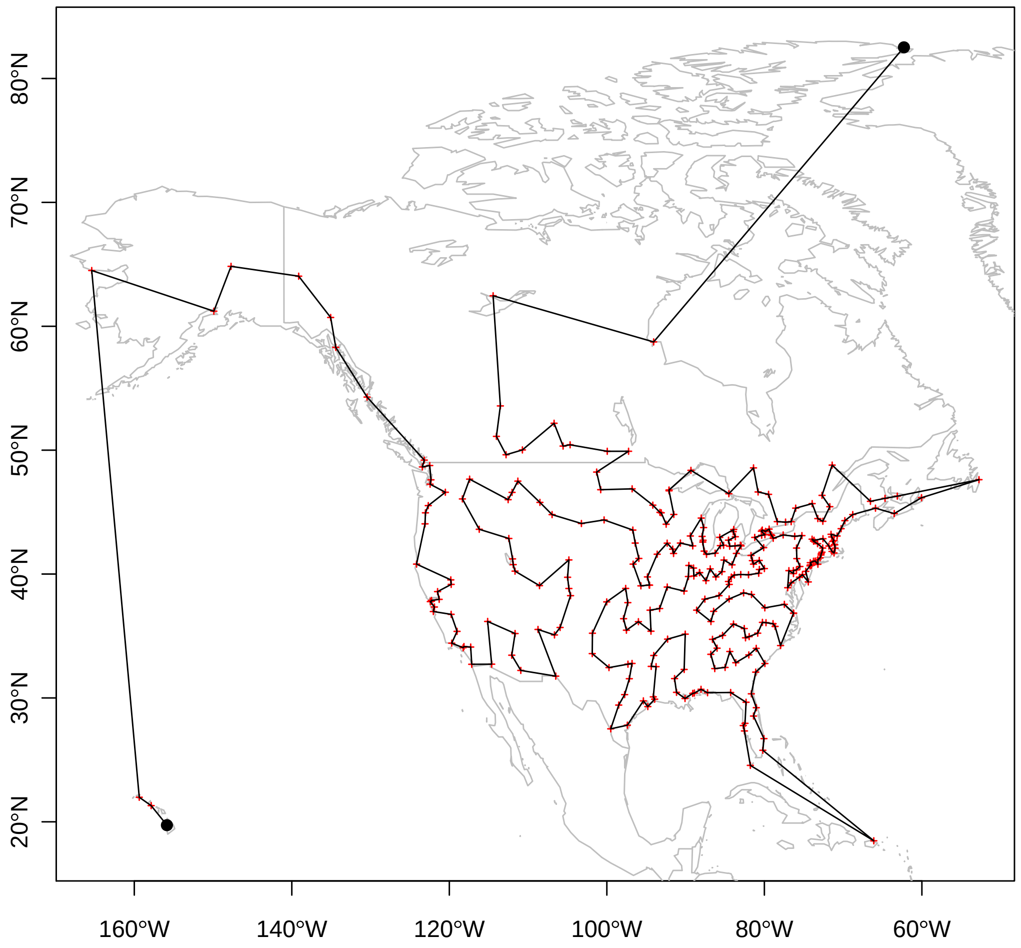
Visualizing the path using **maps** et al.

```
> 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)
```



```
> plot_path(cut_tour(solve_TSP(tsp, method = "concorde"),
+      "cut"))
```



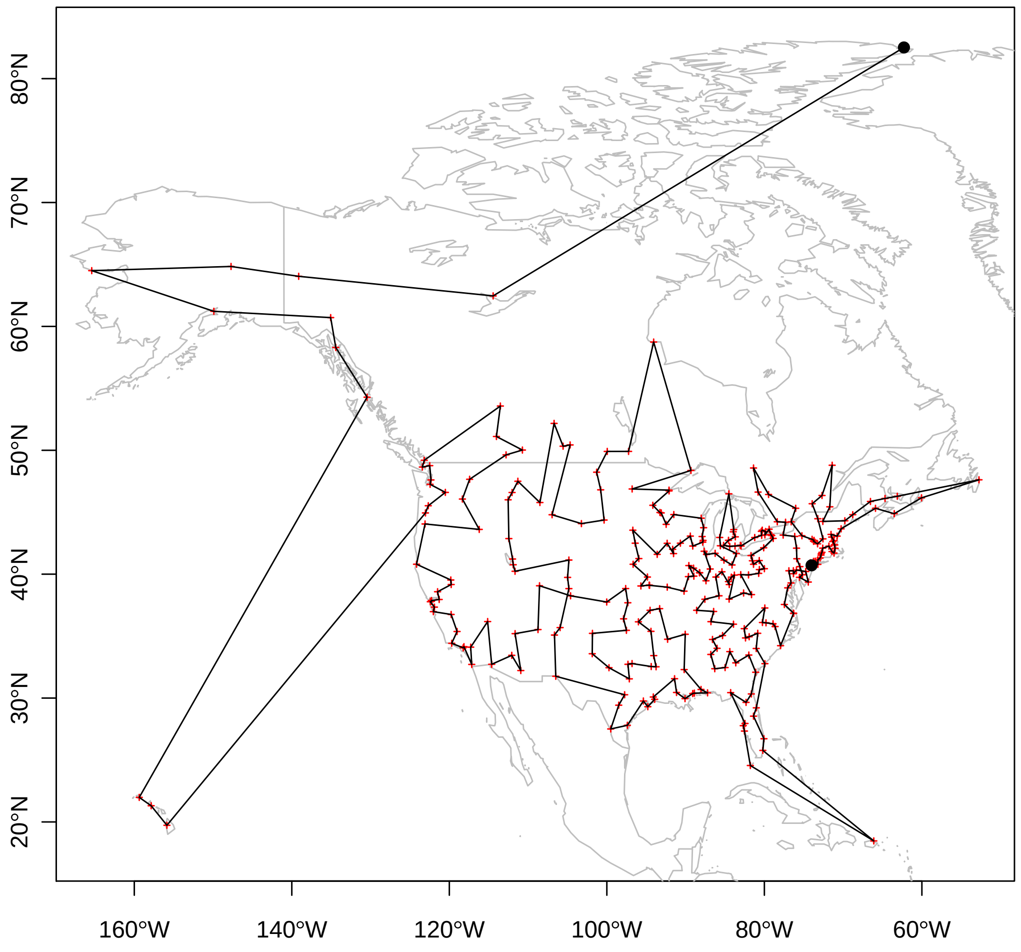
Hamiltonian paths (cont.)

Related Problem: Hamiltonian path starting with a given city (e.g., New York).

Solution: All distances to the selected city are set to zero $\Rightarrow$ asymmetric TSP

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

object of class 'TOUR'
result of method 'nearest_insertion' for 312 cities
tour length: 41654

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

[1] "New York, NY"     "Jersey City, NJ"  "Newark, NJ"
[4] "Elizabeth, NJ"    "Paterson, NJ"     "White Plains, NY"

> tail(labels(path))

[1] "Anchorage, AK"   "Nome, AK"          "Fairbanks, AK"
[4] "Dawson, YT"      "Yellowknife, NT" "Alert, NT"
```

```
> plot_path(path)
```



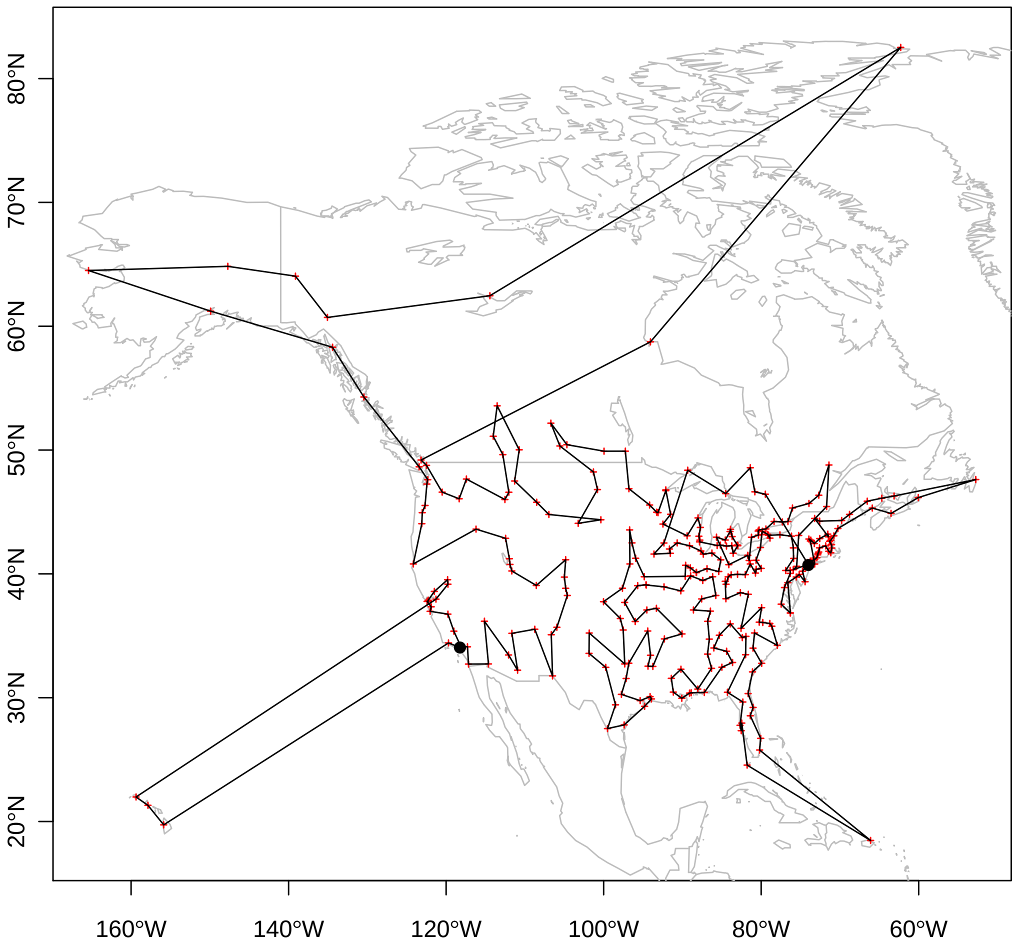
Hamiltonian paths (cont.)

Related Problem: Hamiltonian path with both end points given.
Solution: 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.

```
> 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)

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

```
object of class 'TOUR'
result of method 'nearest_insertion' for 311 cities
tour length: 45094

> 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"      "Central Islip, NY" "Albany, NY"
[4] "Schenectady, NY"   "Troy, NY"          "Pittsfield, MA"

> tail(path_labels)

[1] "Stockton, CA"      "Lihue, HI"         "Honolulu, HI"
[4] "Hilo, HI"          "Santa Barbara, CA" "Los Angeles, CA"
```

```
> plot_path(path_ids)
```



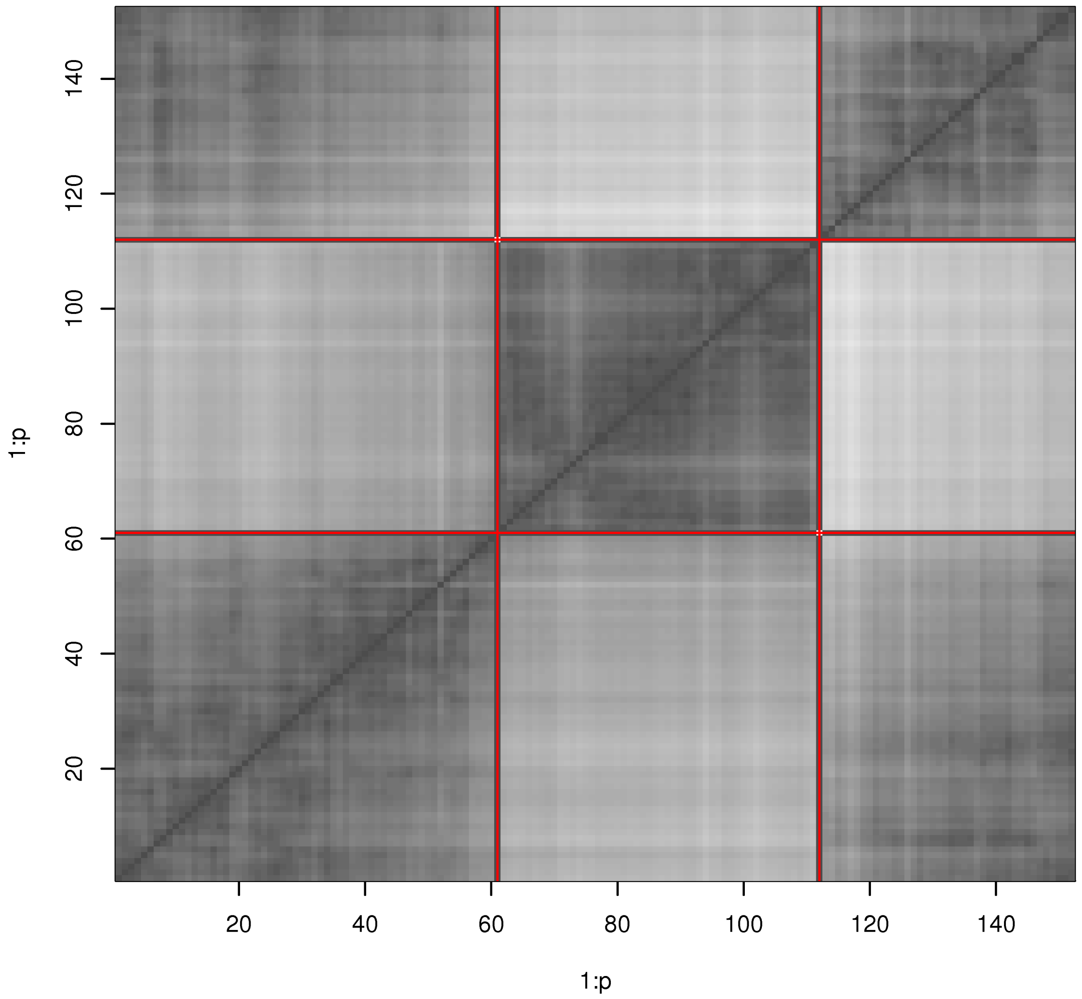
Rearrangement clustering

Climer and Zhang (2006) introduce rearrangement clustering by arranging all objects in a linear order using a TSP. The authors suggest to 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.

```
> data("iris")
> tsp <- TSP(dist(iris[-5]), labels = iris[, "Species"])
> tsp_dummy <- insert_dummy(tsp, n = 2, label = "boundary")
> tour <- solve_TSP(tsp_dummy)
```

Next, we plot the TSP's permuted distance matrix using shading to represent distances.

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


1:p
20
40
60
80
100
120
140
1:p
20
40
60
80
100
120
140

```
> labels(tour)

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

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.

References

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, 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.

R. Garfinkel. Motivation and modeling. In Lawler et al. (1985).

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).

D. Johnson and C. Papadimitriou. Performance guarantees for heuristics. In Lawler et al. (1985).

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.

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.

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.