Thirsty Salesman Problem

Solving TSP using R in context of {sf} & HERE API

My Alma Mater, Prague School of Economics, is located in Žižkov. A formerly working class neighborhood, now rather gentrified, it has to this day retained some traces of its former rougher edges. One of these is an active night life.

A crawl through the bars of Žižkov is therefore a familiar activity for many VŠE students, and can serve as a gateway drug for serious optimization techniques. Such as the Travelling Salesman Problem.

The TSP is an optimization classic, with a number of well understood and highly standardized solutions available in the context of statistical programming language R.

In this blog post I would like to share a practical example of solving the TSP using Open Street Map data of bars via `{osmdata}` and HERE routing engine via `{hereR}`. The actual solution will be found by utilizing the `{TSP}` package.

```r
library(sf) # for spatial data handling
library(dplyr) # for general data frame processing
library(osmdata) # to get data in from OSM
library(leaflet) # to show data interactively
library(hereR) # interface to routing engine
library(TSP) # to solve TSP
```

The first step in our exercise is acquiring data of Žižkov bars. A search is performed over the area of *core Žižkov*, defined as a polygon, using the OSM Overpass API.

As there seems not to be a clear consensus over what constitutes a bar, restaurant or a pub in Prague I am including all three of the possible amenities.

```r
# bbox = http://bboxfinder.com - "core" Žižkov
core_zizkov <- c(14.437408,50.081131,
                 14.452686,50.087561)

# acquire bar data - https://wiki.openstreetmap.org/wiki/Map_features#Amenity
search_res <- opq(bbox = core_zizkov) %>%
  add_osm_feature(key = "amenity",
                  value = c("bar", "restaurant", "pub")) %>%
  osmdata_sf(quiet = T)

# pulls bars as points
bars <- search_res$osm_points %>%
  filter(!is.na(name)) %>%
  select(name)

# show results
leaflet(bars) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addCircleMarkers(fillColor = "red",
                   radius = 5,
                   stroke = F,
                   fillOpacity = .75,
                   label = ~ name)
```

We have located 74 bars, implying a distance matrix of 5476 elements. Not a huge one by today's standards — but big enough to think twice about trying to solve using a pen and a piece of paper.

I have found that while it is not overly difficult to solve the TSP for *all* the Žižkov bars there is educational value in running the TSP over only a small sample. I have found it advantageous to be able to actually show the distance matrix — and this page will easily accommodate only about a 5×5 matrix.

```r
# a sample of bars to make the matrix fit a web page
vzorek ← bars %>%
  slice_sample(n = 5)

# a beer tankard icon for nicer display
beer_icon ← makeAwesomeIcon(
  icon = "beer",
  iconColor = "black",
  markerColor = "blue",
  library = "fa"
)

# a quick overview of our selection
leaflet(vzorek) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addAwesomeMarkers(data = vzorek,
                    icon = beer_icon, # the awesome icon declared earlier
                    label = ~name)
```

The easiest distance matrix to calculate is plain "as the crow flies" distance. This can be calculated via a `sf::st_distance()` call.

The resulting matrix will be based on pure distance, with some differences in interpretation depending on coordinate reference system of underlying data (Euclidean in projected CRS and spherical in unprojected CRS).

```r
# distance matrix "as the crow flies"
crow_matrix ← st_distance(vzorek,
                          vzorek)

# naming the dimensions for easier orientation
rownames(crow_matrix) ← vzorek$name
colnames(crow_matrix) ← vzorek$name

# a visual check; note that the matrix has a {units} dimension
crow_matrix
```

```
## Units: [m]
##                    Vincaffe Vape House Prague     CEEL The Tavern Sushi Sushi
## Vincaffe             0.0000          338.7937 316.2430   471.2013    172.7430
## Vape House Prague  338.7937            0.0000 642.0388   141.9042    376.2636
## CEEL               316.2430          642.0388   0.0000   780.4818    300.0088
## The Tavern         471.2013          141.9042 780.4818     0.0000    518.0866
## Sushi Sushi        172.7430          376.2636 300.0088   518.0866      0.0000
```

Calculating the distance matrix using plain distance is easy, and the resulting matrix is symmetrical (distance from A to B equals distance from B to A). It is also hollow (distance from A to A itself is zero).

Solving the TSP for such a matrix is straightforward, as the hard work has been outsourced to the `{TSP}` package internals.

```
# solve the TSP via {TSP}
crow_tsp <- crow_matrix %>%
  units::drop_units() %>%  # get rid of unit dimension
  # declaring the problem as a symmetric TSP
  TSP() %>%
  solve_TSP()

# the tour (crawl) as sequence of bars
vzorek$name[as.numeric(crow_tsp)]
```

```
## [1] "Vincaffe"          "The Tavern"        "Vape House Prague"
## [4] "Sushi Sushi"       "CEEL"
```

Once we have the optimal route calculated it can be visualized using `{leaflet}`. The sequence of stops needs to be completed (by repeating the first stop after the last) and cast from points to a linestring.

```
stops <- as.numeric(crow_tsp) # sequence of "cities" as indices

# bars in sequence, with the first repeated in last place
crow_result <- vzorek[c(stops, stops[1]), ] %>%
  st_combine() %>% # combined to a single object
  st_cast("LINESTRING") # & presented as a route (a line)

# present the as-the-crow-flies based route in crimson color
leaflet(crow_result) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addPolylines(color = "crimson",
               popup = "as the crow flies...") %>%
  addAwesomeMarkers(data = vzorek,
                    icon = beer_icon, # the awesome icon declared earlier
                    label = ~name)
```

From the visual overview we can see an obvious shortcoming of the “as the crow flies” approach: it completely ignores other constraints except for distance — such as the road network.

Thus while the route shown is “optimal” in the sense that it forms the shortest path joining the five bars selected, it is not one that we could actually follow (unless we were a flying crow).

This shortcoming can be resolved by using an alternative distance matrix as input, while retaining the techniques of `{TSP}` for the actual route selection. A possible source of more applicable data are routing engines, available to R users via API interfacing packages.

There are several of them available, my personal favorite being the `{hereR}` package interfacing to the HERE routing engine. I have found HERE to be very reliable and rich in detail. It does require registration, but its free tier is more than adequate for most individual users.

```
# set the HERE API key; mine is stored in an envir variable
hereR::set_key(Sys.getenv("HERE_API_KEY"))

# a full set of all combinations - 5 x 5 = 25 rows
indices <- expand.grid(from = seq_along(vzorek$name),
```

```r
                        to = seq_along(vzorek$name))

# call routing API for all permutations & store for future use
for (i in seq_along(indices$from)) {

  active_route ← hereR::route(origin = vzorek[indices$from[i], ],
                              destination = vzorek[indices$to[i], ],
                              transport_mode = "car") %>%
    # technical columns for easier use and presentation
    mutate(idx_origin = indices$from[i],
           idx_destination = indices$to[i],
           route_name = paste(vzorek$name[indices$from[i]],
                              ">>",
                              vzorek$name[indices$to[i]])) %>%
    relocate(idx_origin, idx_destination, route_name) %>%
    st_zm() # drop z dimension, as it messes up with leaflet viz

  if (i == 1) {
    # if processing the first sample = initiate a result set
    routes ← active_route
  } else {
    # not processing the first sample = bind to the existing result set
    routes ← routes %>%
      bind_rows(active_route)
  }

}

# a quick overview of structure of the routes data frame
glimpse(routes)
```

```
## Rows: 25
## Columns: 16
## $ idx_origin      <int> 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, ~
## $ idx_destination <int> 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, ~
## $ route_name      <chr> "Vincaffe >> Vincaffe", "Vape House Prague >> Vincaffe~
## $ id              <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ rank            <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ section         <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
## $ departure       <dttm> 2022-10-03 09:11:16, 2022-10-03 09:11:17, 2022-10-03 ~
## $ arrival         <dttm> 2022-10-03 09:11:16, 2022-10-03 09:13:22, 2022-10-03 ~
## $ type            <chr> "vehicle", "vehicle", "vehicle", "vehicle", "vehicle",~
## $ mode            <chr> "car", "car", "car", "car", "car", "car", "car", "car"~
## $ distance        <int> 0, 729, 654, 997, 368, 756, 0, 1130, 245, 715, 877, 12~
## $ duration        <int> 0, 125, 97, 171, 71, 219, 0, 196, 65, 125, 154, 206, 0~
## $ duration_base   <int> 0, 125, 94, 170, 68, 109, 0, 190, 59, 124, 139, 198, 0~
## $ consumption     <dbl> 0.0000, 0.2815, 0.4970, 0.3798, 0.1598, 0.8133, 0.0000~
## $ tolls           <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
## $ geometry        <GEOMETRY [°]> POLYGON ((14.44732 50.08498 ..., LINESTRING (1~
```

The routing results give us several pieces of data:

- the routes as linestring objects in EPSG:4326 (for visualization later on)

- distance of the route (in meters)
- travel time (in seconds) both raw and adjusted for traffic
- petrol consumption

To these I have added three technical columns: indices of start & destination for easier joining of solved TSP results back and the name of the route as string for visualization purposes.

Having a variety of metrics will be helpful in construction of alternative distance matrices.

The first routing distance matrix will be based on route distance; notice that while the matrix is hollow it is not symmetrical. This is not surprising, as routing is not commutative — optimal route from A to B need not be the same as from B to A, due to constraints such as one way roads. Žižkov is a veritable warren of one way streets.

```
# distance matrix based on actual distances
distance_matrix ← matrix(routes$distance,
                          nrow = nrow(vzorek),
                          ncol = nrow(vzorek))

# naming the dimensions for easier orientation
rownames(distance_matrix) ← vzorek$name
colnames(distance_matrix) ← vzorek$name

# a visual check; the units are meters (distance)
distance_matrix
```

```
##                   Vincaffe Vape House Prague CEEL The Tavern Sushi Sushi
## Vincaffe                 0               756  877       1148         209
## Vape House Prague      729                 0 1212        548         504
## CEEL                   654              1130    0       1522         513
## The Tavern             997               245 1480          0         876
## Sushi Sushi            368               715  851       1107           0
```

```
# solve the TSP via {TSP}
distance_tsp ← distance_matrix %>%
  # declaring the problem as asymmetric TSP
  ATSP() %>%
  solve_TSP()

# the tour (crawl) as sequence of bars
vzorek$name[as.numeric(distance_tsp)]
```

```
## [1] "CEEL"              "Vincaffe"          "The Tavern"
## [4] "Vape House Prague" "Sushi Sushi"
```

Once we have solved the TSP and figured the sequence of "cities" to visit it is time to report our results.

For this purpose it is advantageous to prepare a data frame of indices of start and destination, and join it back with the original dataset from HERE API (which contains routes as linestrings).

```
stops ← as.numeric(distance_tsp) # sequence of "cities" as indices

# a route as a set of origin & destination pairs, as indexes,
# destination is offset by one from start (last destination = first start)
distance_route ← data.frame(idx_origin = stops,
                            idx_destination = c(stops[2:(nrow(vzorek))],
                                                stops[1]))

# amend the origin & destination indexes by actual routes
distance_result ←  distance_route %>%
  inner_join(routes,
             by = c("idx_origin", "idx_destination")) %>%
  st_as_sf()

# present the distance based route in goldenrod color
leaflet(distance_result) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addPolylines(color = "GoldenRod",
               popup = ~route_name) %>%
  addAwesomeMarkers(data = vzorek,
                    icon = beer_icon, # the awesome icon declared earlier
                    label = ~name)
```

Since the HERE API is generous in terms of results provided it is not difficult to construct an alternative distance matrix, using a different metric. This could be either trip duration or petrol consumption.

In our specific situation both of these can be expected to be be highly correlated with the plain distance results. All the streets in Žižkov are of very similar type, and the average speed & consumption are unlikely to vary greatly between the routes.

The most significant difference between the distance and time based TSP will be driven by current traffic, which is a factor HERE routing engine considers.

```
# distance matrix based on travel time
duration_matrix ← matrix(routes$duration,
                         nrow = nrow(vzorek),
                         ncol = nrow(vzorek))

# names make the distance matrix easier to interpret
rownames(duration_matrix) ← vzorek$name
colnames(duration_matrix) ← vzorek$name

# a visual check; the units are seconds (time)
duration_matrix
```

```
##                    Vincaffe Vape House Prague CEEL The Tavern Sushi Sushi
## Vincaffe                  0               219  154        274          52
## Vape House Prague       125                 0  206        102          80
## CEEL                     97               196    0        251          79
## The Tavern              171                65  252          0         142
## Sushi Sushi              71               125  144        180           0
```

```
# solving using the same pattern as distance based TSP
duration_tsp ← duration_matrix %>%
  ATSP() %>%
  solve_TSP()
```

```
# the tour (crawl) as sequence of bars
vzorek$name[as.numeric(duration_tsp)]
```

```
## [1] "Vape House Prague" "CEEL"              "Vincaffe"
## [4] "Sushi Sushi"       "The Tavern"
```

Once we have solved the trip duration optimized TSP we again need to report the results; in our use case the output is very similar to the distance based one.

This will not necessarily be the case in other contexts, especially ones with greater variation of road types (city streets vs. highways).

```
# the same steps as for distance based matrix
stops ← as.numeric(duration_tsp)

duration_route ← data.frame(idx_origin = stops,
                             idx_destination = c(stops[2:(nrow(vzorek))],
                                                 stops[1]))
# again, the same as for distance based calculation
duration_result ←  duration_route %>%
  inner_join(routes,
             by = c("idx_origin", "idx_destination")) %>%
  st_as_sf()

# present the duration based route in light blue color
leaflet(duration_result) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addPolylines(color = "cornflowerblue",
               popup = ~route_name) %>%
  addAwesomeMarkers(data = vzorek,
                    icon = beer_icon,
                    label = ~ name)
```

I believe my tongue in cheek example has succeeded in showing two things:

- the ease of applying a standardized solution (the {TSP} package) to a well known and well understood problem (the Travelling Salesman Problem) within the context of R ecosystem
- construction of distance matrices from HERE API routing results, with option to optimize for multiple metrics (such as minimizing the travel distance, travel time and petrol consumption)