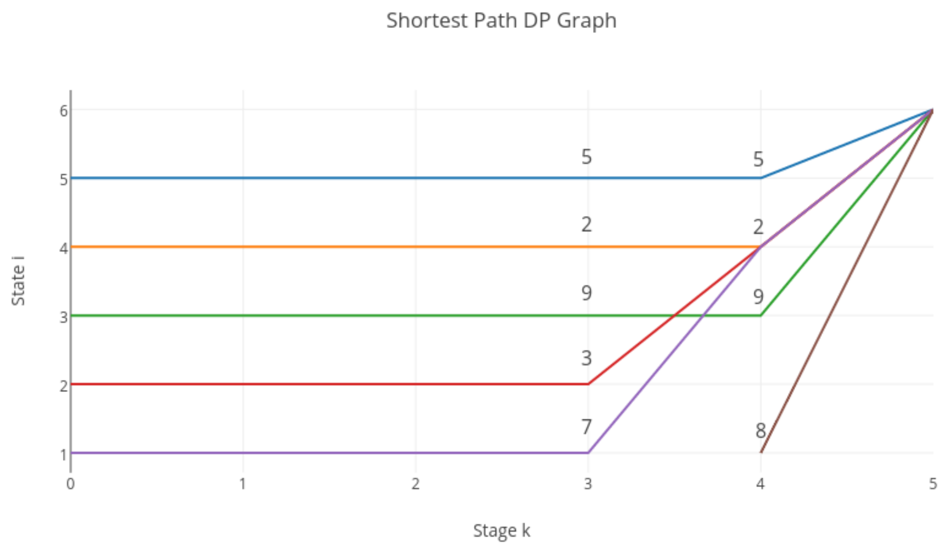
14D006 Stochastic Models and Optimization

Problemset 2

Daniel Bestard Delgado, Michael Cameron, Hans-Peter Höllwirth, Akhil Lohia

March 3, 2017

1 Shortest Path via DP



To solve this problem we used the DP set up, where
$J_{N-1}(i) = a_{it}, \quad i = 1, 2, ..., N$
$J_k(i) = min_{j=1,2,...,6}[a_{ij} + J_{k+1}(j)], \quad k = 1, 2, ..., N-2$

For example, in this graph, we have $J_4(2) = 2$, then following the algorithm $J_3(2) = min_{j=1,2,...,6}[a_{2j} + 2] = 3$. Using this DP algorithm for each node recursively, as discussed in class we find the above graph. Hence the shortest path from each node to node 6 is as follows:
From node 1: 7
From node 2: 3
From node 3: 9
From node 4: 2
From node 5: 5

2 Shortest Path via Label Correcting Methods

In order to find the shortest path of the directed graph provided in this exercise we need to apply two algorithms that come from the general framework **label correcting methods**. Before going into detail it is necessary to provide some notation from this general framework:

- i: denotes the node.
- j: denotes the child of i.
- g_i: length of the shortest path found so far.
- $d_i + a_{ij}$: length of the shortest path to i found so far followed by arc (i, j).
- $UPPER$: length of the shortest path from the source to the target found so far. It is set to $infty$ as long as the target has not been found.
- $OPEN$: contains nodes that are currently active, which means that it is possible their inclusion for the shortest path. Initially it contains the source.

The label correcting algorithm takes the following steps:

1. Remove node i from $OPEN$ and for each child j of i, execute step 2.
2. If $d_i + a_{ij} < min\{d_j, UPPER\}$, set $d_j = d_i + a_{ij}$ and set i to be the parent of j. In addition,
   (a) If $j \neq t$, place j in $OPEN$ if it is not already in $OPEN$.
   (b) If $j = t$, set $UPPER$ to be the new value $d_i + a_{ij}$ of d_t
3. If $OPEN$ is empty, terminate; else go to step 1.

One of the algorithms that is part of the label correction methods is the **Bellman-Ford algorithm**. In this algorithm the node is always removed from the top of $OPEN$ and each node entering $OPEN$ is placed at the bottom of $OPEN$. The following table displays the results of this algorithm for each iteration. Note that in the third column there is a number in parenthesis for each path, which corresponds to its length.

Iter.	Exiting OPEN	OPEN at the end of iter.	UPPER
0	-	1	∞
1	1	1-3(1),1-2(2)	∞
2	1-2	1-2-4(3),1-2-3(3),1-3(1)	2
3	1-3	1-3-4(4),1-3-2(2),1-2-4(3),1-2-3(3)	2
4	1-2-3	1-3-4(4),1-3-2(2),1-2-4(3)	2
5	1-2-4	1-3-4(4),1-3-2(2)	2
6	1-3-2	1-3-4(4)	2
7	1-3-4	$\emptyset$	2

Just to clarify, note that in the second iteration we reach the target through the path 1-2-5. The length of this path is 2. Given that $j = t$ we set $UPPER$ to be the new value of d_t as part b) of the algorithm specifies. The $UPPER$ variable does not get updated anymore because we do not find any other path that goes from the source to the target that has length smaller than 2. Therefore, the shortest path is 1-2-5.

The other algorithm that belongs to the label correction methods is called **Dijkstra's algorithm**. It is very similar to the previous one, but at each iteration it removes from $OPEN$ a node with the minimum value of the label instead of removing the right (top) most node of $OPEN$. The results of the algorithms are:

Iter.	Exiting OPEN	OPEN at the end of iter.	UPPER
0	-	1	∞
1	1	1-3(1),1-2(2)	∞
2	1-3	1-2(2),1-3-2(2),1-3-4(4)	∞
3	1-2	1-2-3(3),1-3-2(2),1-3-4(4)	2
4	1-3-2	1-2-3(3),1-3-4(4)	2
5	1-2-3	1-3-4(4)	2
6	1-3-4	$\emptyset$	2

Again the shortest path is 1-2-5 for the same reasoning as before. Therefore, both algorithms reach the same outcome in this case.

3 Clustering

First of all let's find the shortest path counterpart of the clustering problem. Note that we want to construct clusters that consist of consecutive objects where the number of objects is N. This can be seen as a shortest path problem by growing a binary tree in the following way. Let the first object represent the source node of the tree. At every node we can close the set (make a cluster) or keep of adding objects to the current (open) cluster. The cost of the edge of the option that keeps the cluster open is 0 because we still do not know how many objects will compose the cluster. Therefore, the cost of the edges of the tree is non-zero only when a cluster has been formed, that is, when the set is closed before and that have to be closed at this stage. At the end of the tree we add an artificial node that represents the target. The edges that reach this node will have the costs of the clusters that had not been closed. Therefore, finding the optimal cluster is just a matter is finding the shortest path of this binary tree.

Let's now set the basis of the DP algorithm that is able to deal with this shortest path problem. The first thing to note is that this is a deterministic problem because there is no uncertainty about the state of the system after taking any action. Therefore, the DP algorithm is simply composed of two variables: the state of the system at period k, x_k, and the control variable at period k, u_k. In this specific problem the control variable represent two possible action: new set and add to the set. The first action corresponds to the situation where the object of the next node is added to the a new cluster because at this stage we have just closed a cluster. On the other hand, the second action refers to the situation where the new discovered node is added to the previous open cluster (then, the cluster is not closed after adding this new object). Also note that the state of the system at the next period depends on the decision we made in the previous one. The mathematical summary of this explanation is:

$$x_{k+1} = u_k \quad where \quad u_k \in \{\text{new set, add to the set}\}$$

Finally, the objective function of the DP algorithm is

$$\begin{aligned} J_N(x_N) &= g_N(x_N) \\ J_k(x_k) &= min\ \left\{g_k(x_k, u_k^1) + J_{k+1}(x_{k+1}),\ g_k(x_k, u_k^2) + J_{k+1}(x_{k+1})\right\} \end{aligned}$$

where $g_k(x_k, u_k^1)$ corresponds to the cost of the current clusters when the decision we made at period k is to make a new set (close cluster). Likewise, $g_k(x_k, u_k^2)$ corresponds to the cost of the current clusters when the decision we made at period k is to add to the set (keeping open the current cluster).

4 Path Bottleneck Problem

- s: denotes the origin node.
- t: denotes the terminal node.

- α_{ij}: length from node i to node j.
- b_i: size of bottleneck in the minimum bottleneck path from s to i.
- pi: parent of node i.
- $UPPER$: size of bottleneck in the minimum bottleneck path from s to t. It is set to $infty$ as long as the target has not been found.
- $OPEN$: contains nodes that are currently active, which means that it is possible their inclusion for the bottle neck path. Initially it contains the source.

The label correcting algorithm takes the following steps:

1. Remove node i from $OPEN$ and for each child j of i, execute step 2.
2. If $min\{b_i, \alpha_{ij}\} < min\{d_j, UPPER\}$, set $b_j = min\{b_i, \alpha_{ij}\}$, and set i to be the parent of j. In addition,
   (a) If $j \neq t$, place j in $OPEN$ if it is not already in $OPEN$.
   (b) If $j = t$, set $UPPER$ to be the new value $d_i + a_{ij}$ of d_t
3. If $OPEN$ is empty, terminate; else go to step 1.

5 TSP Computational Assignment

We want to find the shortest tour which visits all 734 cities in Uruguay as provided on the website http://www.math.uwaterloo.ca/tsp/world/countries.html. Due to the large number of cities, the DP algorithm is not applicable for this problem. We therefore rely on several heuristic algorithms which approximate the shortest possible path (79,114 km). In particular, we implemented the following algorithms:

- Nearest neighbor: Select a random city. Find the nearest unvisited city and go there. Continue as long as there are unvisited cities left. Finally, return to the first city.
- Insert heuristics: Start with random initial subtour consisting of 3 cities. Repeatedly, select a city of shortest distance to any of the cities in the subtour but which is not yet itself in the subtour. Find the shortest detour in the subtour to add this city. Repeat until no more cities remain.
- 2-Opt Improvement: Take a non-optimal (but valid) path as input. Repeatedly, remove two edges from the tour and reconnect the two paths created (only if the new tour will be shorter). Continue removing and re- connecting the tour until no 2-opt improvements can be found. The tour is now 2-optimal.

The resulting tours are as follows (the associated tours are shown in Figure 1):

- Nearest neighbor: 99,299 km (runtime < 1 sec)
- 2-opt nearest neighbor: 86,430 km (runtime < 15 sec)
- Insert heuristics: 98,752 km (runtime < 30 sec)
- 2-opt insert heuristcs: 89,168 km (runtime < 15 sec)

Appendix - R Code

```r
# --------------------------------------------------------------------------
# Information
# --------------------------------------------------------------------------
#
# 14D006 Stochastic Models and Optimization
#
# (Authors) Daniel Bestard Delgado, Michael Cameron,
#           Hans-Peter Höllwirth, Akhil Lohia
# (Date)    03.2017

# --------------------------------------------------------------------------
# Loading data
# --------------------------------------------------------------------------

# house cleaning
rm( list=ls() )

# load data
data <- as.data.frame(read.table("../data/TSM_Uruguay.txt", header=FALSE, sep=" "))
N <- nrow(data)
```

```r
# --------------------------------------------------------------------------
# Compute distance matrix
# --------------------------------------------------------------------------
distance.matrix <- function(data) {
    # use high dummy value for distance to itself
    dm <- matrix(10000, N, N)
    for (i in 1:N) {
        for (j in 1:N) {
            if (i != j)
                # Euclidian distance
                dm[i,j] <- round(sqrt((data[i,2] - data[j,2])**2 +
                                      (data[i,3] - data[j,3])**2),3)
        }
    }
    return(dm)
}
```

```r
# --------------------------------------------------------------------------
# Compute path length
# --------------------------------------------------------------------------
path.length <- function(dm, path) {
    length <- 0
    N <- length(path)

    # walk through path and add up distance
    for (i in 2:N) {
        length <- length + dm[path[i-1],path[i]]
    }
    # add distance back to origin
    length <- length + dm[path[N],path[1]]
    return(length)
}
```

```r
# -----------------------------------------------------------------------
# Plot path
# -----------------------------------------------------------------------
plot.path <- function(data, path, color="red") {
    N <- length(path)

    # plot cities
    par( mar=c(0.5,0.5,0.5,0.5), mfrow=c(1,1) )
    plot(-data[,3:2], type='p', pch=16, cex=0.5, asp=1, xaxt='n', yaxt='n')

    # plot path through cities
    for (i in 2:N) {
        segments(-data[path[i-1],3], -data[path[i-1],2],
                 -data[path[i],3], -data[path[i],2], col=color)
    }
    segments(-data[path[N],3], -data[path[N],2],
             -data[path[1],3], -data[path[1],2], col=color)
}
```

```r
# -----------------------------------------------------------------------
# Nearest neighbor algorithm
# -----------------------------------------------------------------------
nn.path <- function(dm) {
    visited <- rep(0,N)
    path <- rep(0,N)

    # starting point
    city <- 1
    visited[city] <- TRUE
    path[1] <- 1

    # repeatedly visit nearest not-yet visited neighbor
    for (i in 2:N) {
        leg <- min(dm[city, !visited])

        # find index of nearest city and add it to path
        potentials <- which(dm[city,] == leg)
        for (j in 1:length(potentials)) {
            if (!visited[potentials[j]]) {
                city <- potentials[j]
                visited[city] <- TRUE
                path[i] <- city
                break
            }
        }
    }
    return(path)
}
```

```r
# ----------------------------------------------------------------------
# Insertion Heuristics
# ----------------------------------------------------------------------
insertion.path <- function(dm) {
    visited <- rep(0,N)

    # start with initial subtour
    visited[1] <- visited[2] <- visited[3] <- TRUE
    path <- c(1,2,3,1)

    while (0 %in% visited){
        # find closest unvisited city to subtour
        leg <- min(dm[!!visited, !visited])

        # find index of closest unvisited city
        potentials <- which(dm[,] == leg, arr.ind=TRUE)
        for (k in 1:nrow(potentials)) {
            if (!!visited[potentials[k,1]] & !visited[potentials[k,2]]) {
                city <- as.numeric(potentials[k,2])
                visited[city] <- TRUE
                break
            }
        }

        # find shortest detour to new city
        detour <- list(len=1000000, from=0, to=0)
        for (i in 1:(length(path)-1)) {
            pot.detour.len <- dm[path[i],city] + dm[path[i+1],city] - dm[path[i],path[i+1]]
            if (pot.detour.len < detour$len) {
                detour$len <- pot.detour.len
                detour$from <- i
                detour$to <- i+1
            }
        }
        # add detour to new path
        path <- c(path[1:detour$from], city, path[detour$to:length(path)])
    }
    return(path[1:N])
}
```

```r
# ----------------------------------------------------------------------
# 2-opt path improvement
# ----------------------------------------------------------------------
two.opt.path <- function(dm, path) {
    # (temp) add return to start
    path <- c(path, path[1])
    N <- length(path)
    changes <- TRUE

    # keep updating path until it is 2-opt path
    while (changes) {
        changes <- FALSE
        for (i in 1:(N-3)) {
            for (j in (i+2):(N-1)) {
                # if alternative path segment is shorter than current segment
                curr.len <- dm[path[i],path[i+1]] + dm[path[j],  path[j+1]]
                new.len  <- dm[path[i],path[j]]   + dm[path[i+1],path[j+1]]

                if (new.len < curr.len) {
                    # swap positions and reverse in-between segment
                    path[(i+1):j] <- path[j:(i+1)]
                    changes <- TRUE
                }
            }
        }
    }
    # return final path (without return to start)
    return(path[1:(N-1)])
}
```

```
# ----------------------------------------------------------------------
# Run path computations and plot solutions
# ----------------------------------------------------------------------

# compute distance matrix
dm <- distance.matrix(data)

# compute nearest neighbor path
path.nn <- nn.path(dm)
path.length(dm, path.nn)
plot.path(data, path.nn, color="red")

# compute insertion heuristics path
path.insert <- insertion.path(dm)
path.length(dm, path.insert)
plot.path(data, path.insert, color="blue")

# compute 2-opt improved nearest neighbor path
path.nn.2opt <- two.opt.path(dm, path.nn)
path.length(dm, path.nn.2opt)
plot.path(data, path.nn.2opt, color="red")

# compute 2-opt improved insertion heuristics path
path.insert.2opt <- two.opt.path(dm, path.insert)
path.length(dm, path.insert.2opt)
plot.path(data, path.insert.2opt, color="blue")
```

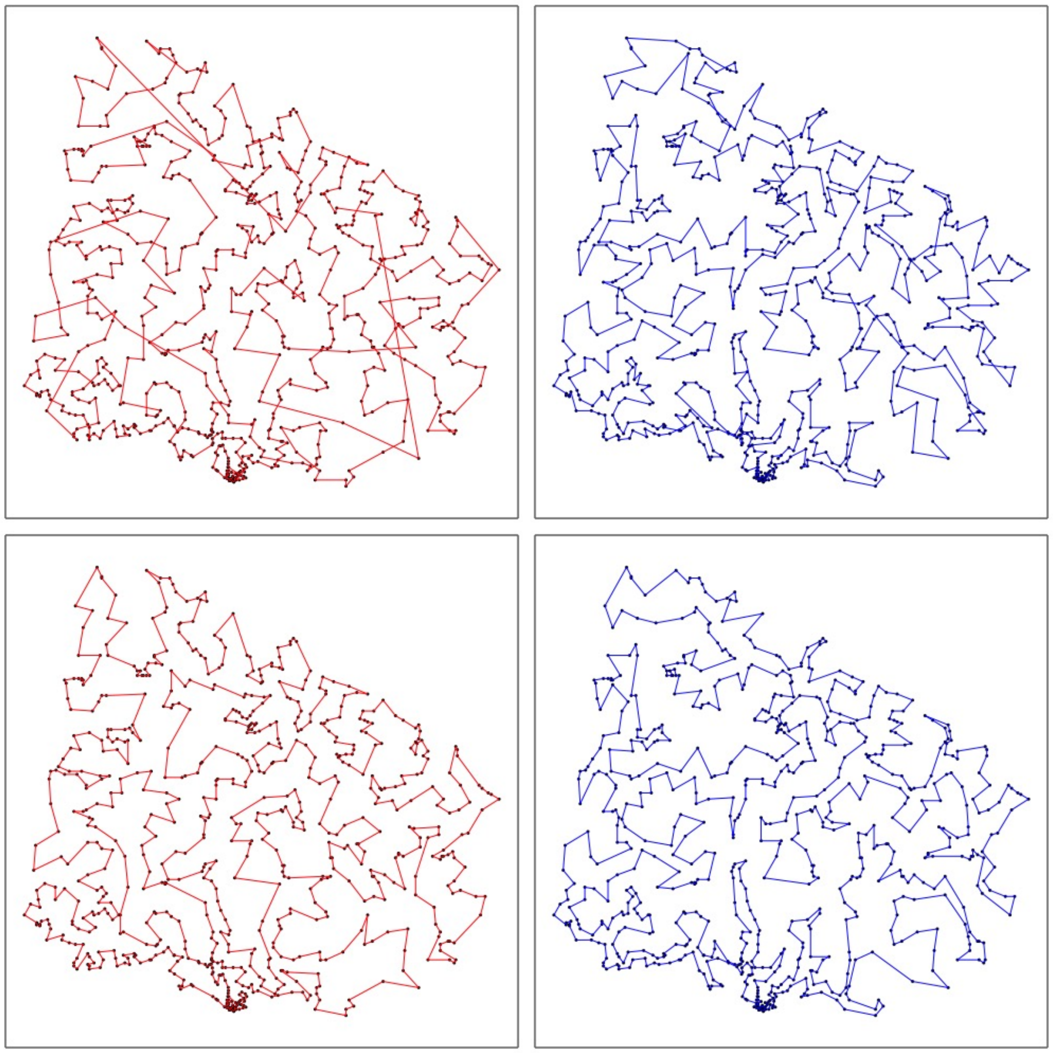


Figure 1: top-left: nearest neighbor tour, bottom-left: 2-opt nearest neighbor tour, top-right: insert heuristics tour, bottom-right: 2-opt insert heuristics tour