The Travelling Salesman Problem:

Introductory notes and computational analysis

Stefano Nasini
Dept. of Statistics and Operations Research
Universitat Politécnica de Catalunya

1. Introduction

The travelling salesman problem (TSP) is an NP-hard problem in which, given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each place exactly once.
The origins of the travelling salesman problem are unclear. A handbook for travelling salesmen from 1832 mentions the problem and includes example tours through Germany and Switzerland, but contains no mathematical treatment.

The travelling salesman problem was defined in the 1800s by the Irish mathematician W. R. Hamilton and by the British mathematician Thomas Kirkman. Hamilton's Icosian Game was a recreational puzzle based on finding a Hamiltonian cycle. The general form of the TSP appears to have been first studied by mathematicians during the 1930s in Vienna and at Harvard, notably by Karl Menger, who defines the problem, considers the obvious brute-force algorithm, and observes the non-optimality of the nearest neighbour heuristic.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. In many

applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In this paper we are going to consider the solution of a symmetric travelling salesman problem on a complete graph G=(V,E) with n=12 vertices.

We shall consider three different methods for solving the problem: the first one is an heuristic procedure based on a constructive part (nearest neighbour heuristic) and an improving part (local search).

A second method we shall use is a Lagrangian Relaxation applied to the Mathematical Programming model. We shall solve the lagrangian problem by means of the sub-gradient algorithm.

Finally, we shall consider solve it by Brunch & Cut, and see how generating new constrains each time the solution provided has sub-cycle.

2. Data for a TSP on a complete graph

The following data matrix constitutes the distance between points we are going to use along the overall documents.

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	750	431	402	133	489	310	566	302	214	785	762
2	750	0	736	671	85	174	870	927	683	336	882	150
3	431	736	0	117	934	65	939	305	422	291	88	507
4	402	671	117	0	982	727	911	870	380	754	367	580
5	133	85	934	982	0	956	834	118	795	633	447	446
6	489	174	65	727	956	0	322	397	356	336	27	872
7	310	870	939	911	834	322	0	127	262	821	776	81
8	566	927	305	870	118	397	127	0	257	429	320	524
9	302	683	422	380	795	356	262	257	0	555	244	290
10	214	336	291	754	633	336	821	429	555	0	508	77
11	785	882	88	367	447	27	776	320	244	508	0	293
12	762	150	507	580	446	872	81	524	290	77	293	0

3. Providing an upper bound of the optimal solution

The word heuristic has a Greek root, Εὑρίσκω, which mean "to find" or "to discover". It refers to experience-based techniques for problem solving, learning, and discovery and is used to speed up the process of finding a good enough solution, where an exhaustive search is impractical. Examples of this method include using a "rule of thumb", an educated guess, an intuitive judgment, or common sense.

In what follows we are going to apply a 2-step heuristic procedure to the solution of a TSP with 12 vertices on a complete graph with the cost matrix defined in the previous session. The first step is to construct a feasible solution and the second is to improve this solution.

The constructive part is based on the nearest neighbour algorithm, which was one of the first algorithm used to determine a solution to the TSP. In it, the salesman starts at a random city and repeatedly visits the nearest city until all have been visited. It quickly yields a short tour, but usually not the optimal one.

The pseudo code of nearest neighbor algorithm is the following.

1. stand on an arbitrary vertex as current vertex.
2. find out the lightest edge connecting current vertex and an unvisited vertex V.
3. set current vertex to V.
4. mark V as visited.
5. if all the vertices in domain are visited, then terminate.
6. Go to step 2.

The sequence of the visited vertices is the output of the algorithm, which we wish to improve by applying a second heuristic procedure: a local search.

A way of improving the current solution provided by the nearest neighbor algorithm is the pairwise exchange. The pairwise exchange or 2-opt technique involves iteratively removing two edges and replacing these with two different edges that reconnect the fragments created by edge removal into a new and shorter tour.

The following R code implements the constructive and improving heuristic method and applies it repeatedly for different initial vertices.

```
> Distance <- read.table("Distance.txt", header = FALSE)
> N <- sqrt(length(Distance)
> X <- matrix(rep(0,N^2),N,N)
> distance <- matrix(Distance,N,N)
> objective_list <- list()
>
> for(initial in 1:N){
+
+ i = initial
+ DISTANCE[,i] <- 1000
+ order <- i
+
+ for (i in 1:N){distance[i,i] = 10000}
+
+       #-------------------------------------------------------------
+       # GREEDY
+       #-------------------------------------------------------------
+
+       while( length(order)<12 ) {
+
+       degree <- TRUE
+       DISTANCE <- distance
+
+       while( degree ) {
+
+               j <- which.min(DISTANCE[i,])

+               if (!(j %in% order)){
+                       order <- c(order, j)
+                       degree <- FALSE
+               }
+               else { DISTANCE[i,j] <- 100000 }
+       }
+
+       i = j
+       print(j)
```

```
+ }
+ order
+
+ #-------------------------------------------------------------
+ #
+ # PAIRWISE EXCHANGE
+ #
+ #-------------------------------------------------------------
+
+ adjacency_list <- list() # create adjacency list
+ k <- 1
+
+ for(k in 1:(N-1)){
+       adjacency_list[[k]] <- c(order[k],order[k+1])
+ }
+
+ adjacency_list[[N]] <- c(order[N], order[1])
+ adjacency_list
+
+ objective <- function(graph, distance){
+
+ cost <- 0
+ for(y in 1:length(graph)){
+       cost <- cost + distance[graph[[y]][1], graph[[y]][2]]
+ }
+ cost
+ }
+ #-------------------------------------------------------------------------------------------
+
+ ADJACENCY_LIST <- adjacency_list
+ OBJECTIVE_LIST <- objective(adjacency_list, distance )
+
+ for (k in 1:(N-1)){
+       for(h in (k+1):N){
+
+               #-----------------------------------------------------------------------
+               # EXCHANGE EDGES
+               #-----------------------------------------------------------------------
+
+               if ((ADJACENCY_LIST[[k]][1] != ADJACENCY_LIST[[h]][1]) && (ADJACENCY_LIST[[k]][1] !=
+                       ADJACENCY_LIST[[h]][2])) {
+
+                       ADJACENCY_LIST[[k]] <- c(adjacency_list[[k]][1], adjacency_list[[h]][1])
+                       ADJACENCY_LIST[[h]] <- c(adjacency_list[[k]][2], adjacency_list[[h]][2])
+
+               }
+
+               #-----------------------------------------------------------------------
+               # Check the goodness of the new adjacency list
+               #-----------------------------------------------------------------------
+
+               if ( objective(ADJACENCY_LIST, distance ) < objective(adjacency_list, distance ) ) {
+
+                       OBJECTIVE_LIST <- c( OBJECTIVE_LIST, objective(ADJACENCY_LIST, distance ))
+
+                       sub_order <- order[which(order==adjacency_list[[k]][2]): which(order== adjacency_list[[h]][1])]
+
+                       position <- NULL
+
+                       for(i in sub_order){
+                               position <- c(position, which(order == i))
+                       }
+
+                       rev_position <- rev(position)
+                       ORDER <- order
+
+                       for (t in 1:length(position)){
+                               ORDER[position[t]] <- order[rev_position[t]]
+                       }
+
+                       order <- ORDER
+                       for(k in 1:(N-1)){
+                               adjacency_list[[k]] <- c(order[k],order[k+1])
+                       }
+                       adjacency_list[[N]] <- c(order[N], order[1])
+               }
+               ADJACENCY_LIST <- adjacency_list
+       }
+ }
+ objective_list[[initial]] <- OBJECTIVE_LIST
+ print(OBJECTIVE_LIST)
+ }
```

The following results show the final Hamiltonian cycle obtained after applying the nearest neighbor methods, starting from each node.

```
1
10,  12,  7,  8,  5,  2,  6,  11,  3,  4,  9,
Objective function:  1790
```

```
2
10,  12,  7,  8,  5,  1,  9,  11,  6,  3,  4,
Objective function:  2298 2128 2098 1951
```

```
3
10,  12,  7,  8,  5,  2,  6,  11,  9,  1,  4,
Objective function:  2045 2015
```

```
4
10,  12,  7,  8,  5,  2,  6,  11,  3,  9,  1,
Objective function:  2657 2473 2468 2441 2078
```

```
5
10,  12,  7,  8,  9,  11,  6,  3,  4,  1,  2,
Objective function:  2865 2648 2277 2276
```

```
6
10,  12,  7,  8,  5,  2,  4,  3,  11,  9,  1,
Objective function:  2735 2656 2579 2480 2479 2122
```

```
7
10,  12,  2,  5,  8,  9,  11,  6,  3,  4,  1,
Objective function:  2673 2119 1951
```

```
8
10,  12,  7,  9,  11,  6,  3,  4,  1,  5,  2,
Objective function:  2849 2840 2415 2355 2354
```

```
9
10,  12,  7,  8,  5,  2,  6,  11,  3,  4,  1
Objective function:  2153 2045 2015
```

```
10
7,  12,  2,  5,  8,  9,  11,  6,  3,  4,  1,
Objective function:  2581 2557 2240 2088 2083
```

```
11
10,  12,  7,  8,  5,  2,  6,  3,  4,  9,  1
Objective function:  2819 2564 2114 2038
```

```
12
10,  1,  5,  2,  6,  11,  3,  4,  9,  8,  7,
Objective function:  1790
```

We have our best upper bound of the TSP, which we obtain starting the GREEDY from node 12 and improving such a solution by applying a pairwise exchange method. This upper bound is 1790.

4. Lagrangian relaxation to obtain a lower bound

In this session we are going to consider again a Mathematical Programming formulation of the TSP, but we are going to do it with the specific purpose of obtaining lower bound of such a problem by Lagrangian Relaxation. To understand this formulation consider a 1-spanning-tree, a connected graph with a unique cycle (or the graph we obtain by adding an edge to a tree) and ask which is the topology of a 1-tree which is forced to have all vertices with degree equal to 2.

Immediately, one realizes that the set of all 1-spanning-tree with n vertices which is forced to have a have all vertices with degree equal to 2 is exactly the set of all Hamiltonian cycles of a graph of *n* vertices.
Let $\mathbf{T}$ be the set of all 1-spanning-tree of a graph with *n* nodes. Then the following Mathematical Programming problem has the same feasible set of the one shown in the previous session.

$$\text{(problem 2)}\begin{cases} \min_x \sum_{j<i} c_{ij} x_{ij} & & \\ \text{s.t.} & & \\ \quad \sum_{j<i} x_{ji} + \sum_{i>j} x_{ij} = 2 & i = 1,\ldots,n & [1] \\ \quad x \in \mathbf{T} & & [2] \\ \quad x_{ij} \ \textit{binary}, & i, j = 1,\ldots,n & [3] \end{cases}$$

To apply Lagrangian relaxation to this problem, divide the constraints in two respective types: the ones which are easy to be solved and the other which cause higher complexity in computing the solution.

In our case, the easy constrains are the ones which impose to the solution to be a 1-spanning-tree, since the problem of finding the minimum-spanning-one-tree of a given graph, under a given cost function, can be solved in polynomial time Kruscal or Prime algorithm. Prime algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

1. Input: A non-empty connected weighted graph with vertices V and edges E.

2. Initialize: V_{new} = {x}, where x is an arbitrary node from V, E_{new} = {}

3. Repeat until V_{new} = V:
   i. Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not.
   ii. Add v to V_{new}, and (u, v) to E_{new}

4. Output: V_{new} and E_{new} describe a minimal spanning tree

The idea is to relax constrains [1] adding a penalization in the objective function for the violation of them, and construct minimum-spanning-one-trees with a cost function which is parameterized by the above mention coefficient of penalizarion.

In some sense, what we obtain is a family of optimization problem parameterized by the penalization coefficients, or a family of minimum-spanning-one-trees parameterized by the penalization coefficients. The cost function of each minimum-spanning-one-tree will be the following.

$$\min_x \sum_{j<i} c_{ij} x_{ij} + \sum_{i=1}^{n} \lambda_i \left(2 - \sum_{j<i} x_{ji} + \sum_{i>j} x_{ij} \right)$$

It could also be seen a parameterized mapping $L:\mathbf{R}^n \to \mathbf{R}$, which associate to each value of the penalization coefficients, known as Lagrange multipliers, the cost of the associated minimum-spanning-one-tree.

$$L(\lambda) = \begin{cases} \min_x \sum_{j<i} c_{ij} x_{ij} + \sum_{i=1}^{n} \lambda_i \left(2 - \sum_{j<i} x_{ji} + \sum_{i>j} x_{ij} \right) \\ \text{s.t.} \\ \quad x \in \mathbf{T} \\ \quad x_{ij} \ \textit{binary} \end{cases} = \quad L(\lambda) = 2\sum_{i=1}^{n} \lambda_i + \begin{cases} \min_x \sum_{j<i} \left(c_{ij} - \lambda_i - \lambda_j \right) x_{ij} \\ \text{s.t.} \\ \quad x \in \mathbf{T} \\ \quad x_{ij} \ \textit{binary} \end{cases} .$$

It can be proved that for each $\lambda_0 \in \mathbf{R}^n$, $L(\lambda_0) \le f^*$, where f^* is the optimal value of problem 2, which means that the lagrangian function $L(\lambda)$ provide lower bounds of the problem. This means that the problem to be solved would be to find the $\lambda_0 \in \mathbf{R}$ which provide the best lower bound for problem 2, that is, $\arg\max L(\lambda)$.

The problem would then be to build an efficient algorithm which allow solving $\arg\max L(\lambda)$. Here we are going to consider the sub-gradient algorithm, which is a generalization of the gradient algorithm to the case of non-differentiable functions.

Let $L:\mathbf{R}^n \to \mathbf{R}$ be a convex function with domain $\mathbf{R}^n$. A classical subgradient method iterates

$$\lambda^{(t+1)} = \lambda^{(t)} - \alpha_t \nabla^t$$

where ∇^t denotes a subgradient of L at $\lambda^{(t)}$. If L is differentiable, then its only subgradient is the gradient vector $\nabla L(\lambda^t)$ itself. It may happen that $-\nabla^t$ is not a descent direction for L at $\lambda^{(t)}$. We therefore maintain a list L_{best} that keeps track of the lowest objective function value found so far, i.e. $L_{best}^{(t)} = \min\{L_{best}^{(t)}, L^{(t)}\}$. The pseudocode for the subgradient algorithm applied to the TSP problem is the following.

Initialization Take an initial point $\lambda^{(t)}$ and take the counter $t = 0$

Iterative step

compute ∇^t by finding a and minimum-spanning-one-tree

if $\nabla^t = 0$ $\Rightarrow$ $\lambda^{(t)}$ is the optimal

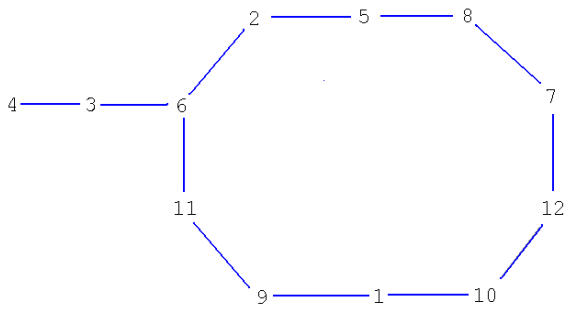
if $\nabla^t \neq 0$ $\Rightarrow$ calculate α_t and update $\lambda^{(t+1)} = \lambda^{(t)} - \alpha_t \nabla^t$

We implemented this procedure in Matlab. In function `prim`(C) we apply the Prim's algorithm for constructing a minimum-spanning-one-tree to a square matrix C.

```
function  [adjacency_list adjacency] = prim(distance)


%-------------------------------------------------------------------------
% PRIM ALGORITHM
%-------------------------------------------------------------------------

cost = distance(2:12,2:12);              % eliminate the first node

n = length(distance);
N = length(cost);
T = 1;
S = [2:N];
adjacency = zeros(12,12);
adjacency_list = [];

for i=1:N
    cost(i,i) = 9999999;
end
for i=1:n
    distance(i,i) = 9999999;
end

DISTANCE = distance;

list = [];

while length(T) < 11


    list_distance = [];
    list_position = [];

    t = 1;

    for i = T

        j = find(cost(i,:) == min(cost(i,:)));

        if length(j) > 1
            j = j(1);
        end

        list_distance = [list_distance; cost(i,j)];
        list_position = [list_position; i j];

        t = t+1;

    end %for

    smallest_distance = find(list_distance == min(list_distance));
    edge = list_position(smallest_distance,:);

    DISTANCE(edge(1), edge(2)) = 9999999;
    DISTANCE(edge(2), edge(1)) = 9999999;

    for i = T
        cost(i, edge(2)) = 9999999;
        cost(edge(2), i) = 9999999;
    end %for

    list = [list; edge(1)+1 edge(2)+1];

    adjacency(edge(1)+1, edge(2)+1) = 1;
    adjacency(edge(2)+1, edge(1)+1) = 1;

    T = [T edge(2)];

end % while

    j1 = find(DISTANCE(1,:) ==  min(DISTANCE(1,:)));
    DISTANCE(1,j1) = 999999;

    j2 = find(DISTANCE(1,:) ==  min(DISTANCE(1,:)));

    if length(j1) > 1
        j1 = j1(1);
```

```
    end

    if length(j2) > 1
        j2 = j2(1);
    end

    list = [list; 1 j1];
    list = [list; 1 j2];

    adjacency_list = list;

    adjacency(1,j1) = 1;
    adjacency(j1,1) = 1;

    adjacency(1,j2) = 1;
    adjacency(j2,1) = 1;

end
```

The following figure show a one-tree obtained by processing our data by means of the Matlab code shown above.



Now, we try to iteratively obtain minimum-spanning-one-trees in accordance with a subgradient method, namely, by updating the costs of the edges.

We implemented the subgradient method using two different criterions for computing the step-length: the first one is a linear descendent step-length (namely, it linearly decreases along the iterations) and the second one is to uses the information of the upper bound and the current gradient, as it is required in the assignment.

$$\lambda^k = \delta^k \frac{\hat{z} - L(u^k)}{\left\| \gamma^k \right\|^2}$$

where

$$\delta^{k+1} := \begin{cases} \alpha\delta^k & \text{LB}^k = \text{LB}^l, l = k - l_{\max} + 1, \ldots, k-1 \\ \delta^k & \text{otherwise} \end{cases}$$

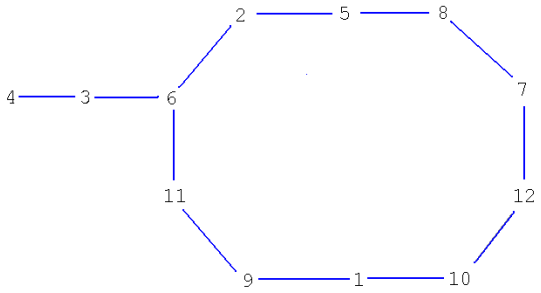
The following code implements the subgradient method for maximizing the lagrangian function using a linearly descendent step-length.

```
function  Subgradient(distance, alpha, initial_step, max_iter)

    N = size(distance);
    lambda = zeros(1,12);           % Initialize the lagrange multipliers
    NewCosts = distance;            % Initial costes
    iteration = 0;                  % iteration counter
    e = ones(1,12);
    max = max_iter;

    step = initial_step;
    alpha = 0.9;

    %--------------------------------------------------------------------------
    %
    % SUBGRADIENT ALGORITHM
    %
    %--------------------------------------------------------------------------

    while(iteration < max)

       iteration = iteration + 1;

       fprintf('-------------------------  ITERARION %d -------------------------\n', iteration );

       [adjacency_list X] = prim(NewCosts); % GENERATE A MINIMUM SPANNING ONE_TREE

       lambda

       Lagrangian = (sum(sum(distance.*X))/2) - lambda*(e*2-sum(X))'

       gradient = (e*2-sum(X))';

       gradient_transpose = gradient'

       if(gradient.*gradient)==0
            break
       end % if

       lambda = lambda + step*gradient';

       step = step*alpha

       adjacency_list

       for i=1:N
           for j=1:N
               if i == j
                   NewCosts(i,j)=0;
               else
                   NewCosts(i,j) = distance(i,j) - (lambda(i) + lambda(j));
               end
           end
       end

    end % while

end % function
```

The following table show the results for the problem we are considering: >> `Subgradient(C, 0.9, 30, 9)`

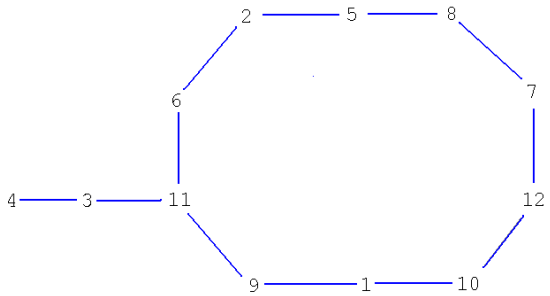
ITERARION 1

```
lambda =
     0     0     0     0     0     0     0     0     0     0     0     0

Lagrangian =
        1631

gradient_transpose =
     0     0     0     1     0    -1     0     0     0     0     0     0

step =
    27

adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6    11
     6     3
     3     4
    11     9
     1    10
     1     9
```



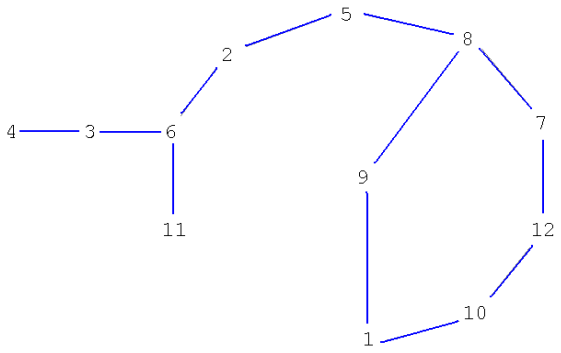
ITERARION 2

```
lambda =
     0     0     0    30     0   -30     0     0     0     0     0     0

Lagrangian =
        1624

gradient_transpose =
     0     0     0     1     0     0     0     0     0     0    -1     0

step =
   24.3000


adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6    11
    11     3
     3     4
    11     9
     1    10
     1     9
```



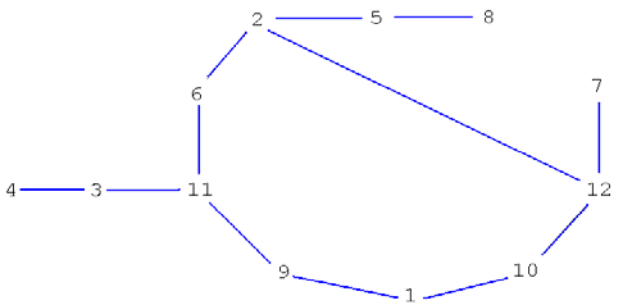
ITERARION 3

```
lambda =

     0     0     0    57     0   -30     0     0     0     0   -27     0

Lagrangian =

        1584

gradient_transpose =

     0     0     0     1     0    -1     0    -1     0     0     1     0

step =

   21.8700

adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6    11
     6     3
     3     4
     8     9
     1    10
     1     9
```



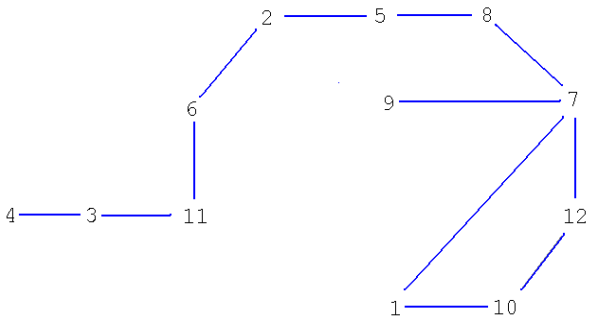
ITERARION 4

```
lambda =
0      0      0   81.3000      0  -54.3000      0  -24.3000      0      0   -2.7000      0

Lagrangian =
  1.6173e+003

gradient_transpose =
     0    -1     0     1     0     0     1     1     0     0    -1    -1

step =
   19.6830

adjacency_list =

     2     5
     5     8
     2    12
    12    10
    12     7
     2     6
     6    11
    11     3
     3     4
    11     9
     1    10
     1     9
```



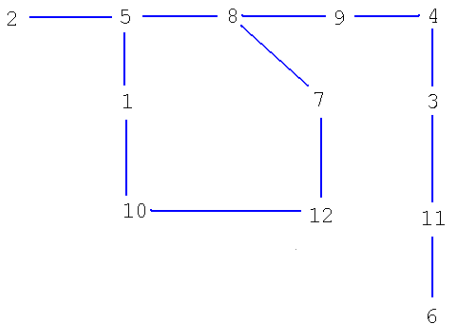
ITERARION 5

```
lambda =
0  -21.8700    0  103.1700     0  -54.3000   21.8700   -2.4300     0     0  -24.5700  -21.8700


Lagrangian =
  1.6206e+003


gradient_transpose =
     0     0     0     1     0     0    -2     0     1     0     0     0


step =
   17.7147

adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     7     9
     2     6
     6    11
    11     3
     3     4
     1    10
     1     7
```



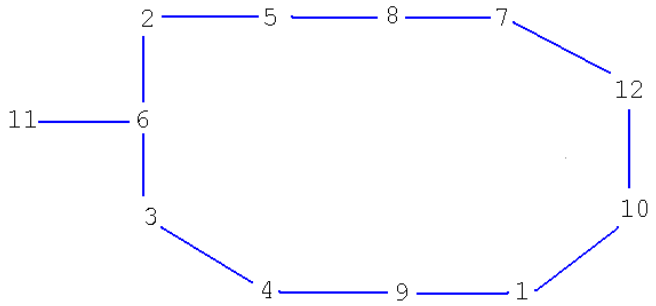
TERARION 6

```
lambda =
0  -21.8700    0   122.8530   0  -54.3000  -17.4960   -2.4300   19.6830    0  -24.5700  -21.8700

Lagrangian =
  1.7777e+003

gradient_transpose =
  0     1     0     0    -1     1     0    -1     0     0     0     0

step =
   15.9432


adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     8     9
     9     4
     4     3
     3    11
    11     6
     1     5
     1    10
```



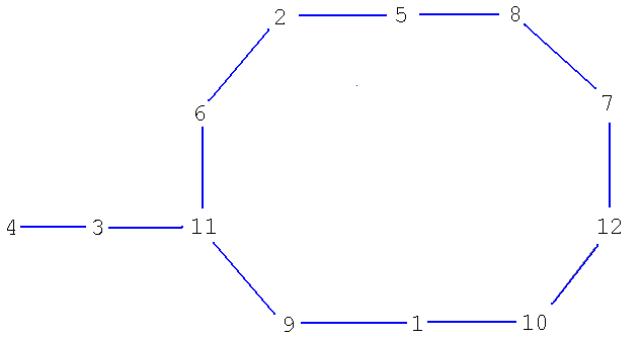
ITERARION 7

```
lambda =
0   -4.1553   0  122.8530  -17.7147  -36.5853  -17.4960  -20.1447   19.6830   0  -24.5700  -21.8700

Lagrangian =
  1.7550e+003


gradient_transpose =
     0     0     0     0     0    -1     0     0     0     0     1     0

step =
   14.3489

adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6    11
     6     3
     3     4
     4     9
     1    10
     1     9
```



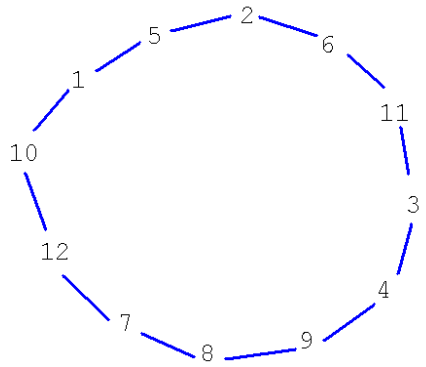
ITERARION 8

```
lambda =
0   -4.1553  0  122.8530  -17.7147  -52.5285  -17.4960  -20.1447  19.6830  0   -8.6268  -21.8700

Lagrangian =

  1.5225e+003


gradient_transpose =
     0     0     0     1     0     0     0     0     0     0    -1     0

step =
   12.9140

adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6    11
    11     3
     3     4
    11     9
     1    10
     1     9
```



```
ITERARION 9
```

```
lambda =
0   -4.1553   0  137.2019  -17.7147  -52.5285  -17.4960  -20.1447   19.6830   0  -22.9757  -21.8700

Lagrangian =
        1790


gradient_transpose =
     0     0     0     0     0     0     0     0     0     0     0     0


step =
   11.6226


adjacency_list =

     2     5
     5     1
     1     10
     10    12
     12    7
     7     8
     8     9
     9     4
     4     3
     3     11
     11    6
     6     2
```



We now consider the case of a step length with computed as a function of the current distance from the upper bound and the current gradient. The following MatLab code implement such a case, which constitute a modified version of the previous code.

```
function  CorrectedSubgradient(distance,delta,lmax,alfa,UB, max_iter)

    N = size(distance);
    LB = 0;                         % Lower bound
    l = 0;                          % Number of iterations without improving the lower bound
    lambda = zeros(1,12);           % Initialize the lagrange multipliers
    subgradient = zeros(1,12);      % Initialize subgradient vector
    NewCosts = distance;            % Initial costes
    iteration = 0;                  % iteration counter
    e = ones(1,12);
    max = max_iter;

    %-------------------------------------------------------------------------
    %
    % SUBGRADIENT ALGORITHM
    %
    %-------------------------------------------------------------------------


    while(iteration < max)

       iteration = iteration + 1;

       fprintf('--------------------------  ITERARION %d ------------------------\n', iteration );

       [adjacency_list X] = prim(NewCosts); % GENERATE A MINIMUM SPANNING ONE_TREE

       lambda
```

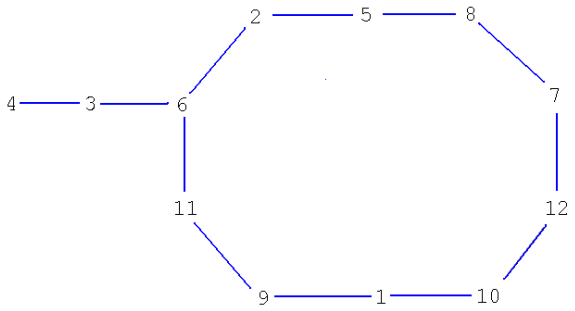
```
 Lagrangian = (sum(sum(distance.*X))/2) - lambda*(e*2-sum(X))'

       gradient = (e*2-sum(X))';

       gradient_transpose = gradient'

       if(gradient'*gradient)==0
            break
       end % if


       adjacency_list

        % UPDATE LOWER BOUND
        if (Lagrangian > LB)
            LB = Lagrangian;
            l = 0;
        else
            l = l+1; % update the counter of the number of iteration without improving
        end

        % UPDATE THE STEP-LENGH
        if(l == lmax)
            delta = alfa*delta;
            l=0;
        end

        delta

        gradient'*gradient

        step = abs(Lagrangian-UB)/(gradient'*gradient)


        % UPDATE LAGRANGE MULTIPLYERS
        lambda = lambda + step*gradient';

        % UPDATE COSTS
        % NewCosts(i,j) = PreviousCosts(i,j) - (lambda(i) + lambda(j));

        for i=1:N
           for j=1:N
               if i == j
                   NewCosts(i,j)=0;
               else
                   NewCosts(i,j) = distance(i,j) - (lambda(i) + lambda(j));
               end
           end
        end


    end % while

end % function
```

The result we obtain with the parameters $\delta 0 = 3$, $\alpha = 0.1$ and lmax=3 are shown in the following tables:

```
>>CorrectedSubgradient(C,3,3,0.1,1790, 50)
```

ITERARION 1

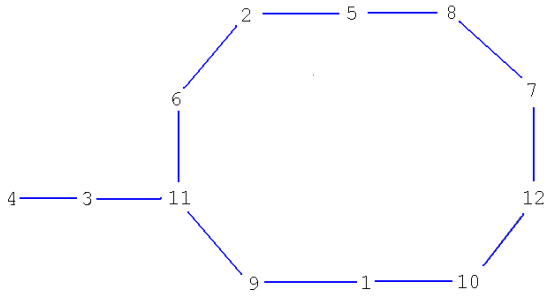
```
lambda =
0     0     0     0     0     0     0     0     0     0     0     0

Lagrangian =
        1631

gradient_transpose =
 0     0     0     1     0    -1     0     0     0     0     0     0

delta =
     3

step =

   79.5000

adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6    11
     6     3
     3     4
    11     9
     1    10
     1     9
```



ITERARION 2

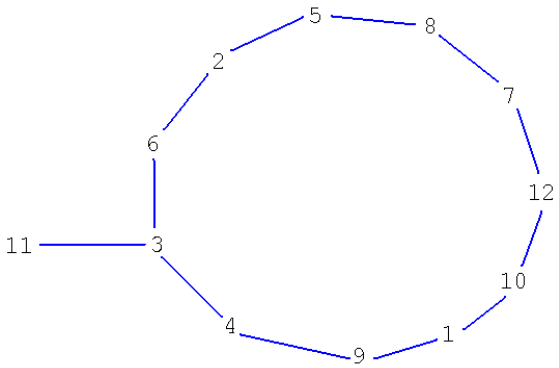
```
lambda =
0      0      0   79.5000      0  -79.5000      0      0      0      0      0      0

Lagrangian =
  1.5745e+003


gradient_transpose =
     0     0     0     1     0     0     0     0     0     0    -1     0

delta =
     3

step =

  107.7500
```

```
adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6    11
    11     3
     3     4
    11     9
     1    10
     1     9
```



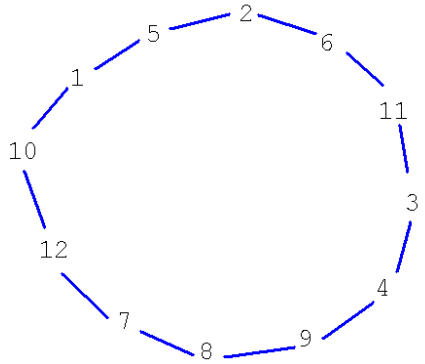
ITERARION 3

```
lambda =
0      0      0  187.2500      0  -79.5000      0      0      0      0 -107.7500      0

Lagrangian =
  1.9358e+003


gradient_transpose =
0     0    -1     0     0     0     0     0     0     0     1     0

delta =
     3

step =
   72.8750


adjacency_list =

     2     5
     5     8
     8     7
     7    12
    12    10
     2     6
     6     3
     3     4
     4     9
     3    11
     1    10
     1     9
```



ITERARION 4

```
lambda =
0     0  -72.8750  187.2500    0  -79.5000     0     0     0     0  -34.8750     0

Lagrangian =
        1790

delta =
     3

gradient_transpose =
     0     0     0     0     0     0     0     0     0     0     0     0

adjacency_list =

     2     5
     5     1
     1     10
     10    12
     12    7
     7     8
     8     9
     9     4
     4     3
     3     11
     11    6
     6     2
```



Both algorithms (the one with linearly decreasing step-length and the other which computes the step-length in accordance with the distance of the current solution from the upper bound) converge to the same Hamiltonian cycle, but the second one needs only 4 iterations whereas the first needs 9 iterations.

It is interested to see that after obtaining the best lower bound, by maximizing the dual function, one realize that such an maximum is 1790, which is exactly the value of the best upper bound we obtained by the heuristic methods.

This means that by duality, we prove that the solution provided by the Local Search we applied in the beginning of this document to our instance of the TSP is the optimal one.