Business Analytics: CVRP

Matteo Bianco, Filippo Grobbo

Luglio 2022

Metodo risolutivo

Per la risoluzione del problema CVRP abbiamo deciso di strutturare il codice a partire da un main script. In questo viene inanzitutto caricato il dataset A-n33-k5 che fa parte del set A di Augerat del 1995 [1]. La soluzione ottimale ha una lunghezza totale percorsa pari a 661, utilizzando 5 veicoli. Il dataset presenta al suo interno 4 strutture dati che codificano rispettivamente:

- coordinate dei nodi
- domanda di ogni nodo
- numero di nodi
- capacità di ogni veicolo

Dopo aver definito la matrice delle distanze, vengono chiamati due funzioni che implementano i metodi che abbiamo deciso di utilizzare per risolvere il problema: metodo costruttivo di Clake-Wright e metodo iterativo 2-opt.
Abbiamo scelto di codificare i percorsi associati ad ogni veicolo con dei vettori di lunghezza differente contenenti l'elenco ordinato dei nodi visitati. Tutti questi path vengono collezionati all'interno di un cell-array chiamato "routes". Per calcolare la lunghezza di ogni percorso definiamo inoltre una funzione "lunghezza_percorso".
L'ultima parte del main script riguarda infine i plot dei risultati. Il codice matlab completo si trova in appendice.

Clarke-Wright

Il primo algoritmo che applichiamo è quello di Clarke-Wright. Si tratta di un'euristica costruttiva e di tipo parallelo, basata sul criterio dei savings. In particolare questo algoritmo non permette di decidere a priori il numero di veicoli da utilizzare, quindi verificheremo a posteriori di averne utilizzato il giusto numero (5).
Nell'implementazione pratica, consideriamo di avere n nodi da visitare e un nodo 1 di deposito. Inizializziamo dunque n percorsi del tipo 1 - i - 1 $\forall\ i \in \{2, ..., n+1\}$. Nel codice scegliamo di non inserire il deposito all'inizio e alla fine dei percorsi per maggiore semplicità delle operazioni.
Creiamo la matrice dei savings: si tratta di una matrice n x n avente per ogni entrata il valore

$$s_{ij} = c_{i1} + c_{1j} - c_{ij}$$

dove c_{ij} è la distanza tra i nodi i e j. Il valore s_{ij} rappresenta il risparmio che otteniamo sostituendo i due archi c_{i1} e c_{1j} con c_{ij}, ovvero assegnando i nodi i e j allo stesso veicolo. Osserviamo che per una matrice delle distanze simmetrica la matrice dei savings è simmetrica, quindi ne calcoliamo solo la parte triangolare superiore. Aggiungiamo inoltre

[1] http://vrp.galgos.inf.puc-rio.br/index.php/en/plotted-instances?data=A-n33-k5

un parametro θ per rendere più robusta l'euristica, modificando la matrice dei savings secondo

$$\tilde{s}_{ij} = s_{ij} - \theta c_{ij}$$

Facendo variare il valore di θ decidiamo quanto scoraggiare l'inserimento di archi troppo lunghi.
Dopo che abbiamo costruito la matrice dei savings, procediamo iterativamente nella maniera seguente

- Troviamo il massimo valore $\tilde{s}_{ij}$ fra i savings
- Verifichiamo se possiamo unire i nodi i e j in un unico percorso (vedi dopo) e in caso positivo fondiamo i percorsi
- Settiamo a 0 il valore di $\tilde{s}_{ij}$

Per quanto riguarda il punto due dell'elenco precedente notiamo che, prima di inserire un arco in un percorso bisogna fare 3 controlli

- I nodi i e j devono essere il primo o ultimo nodo visitato dal loro percorso (subito dopo o subito prima del deposito)
- I nodi i e j non devono appartenere allo stesso percorso
- Il nuovo percorso che creeremmo deve rispettare il vincolo di capacità dei veicoli

Una volta verificato che il saving selezionato rispetta questi vincoli, uniamo i nodi i e j in un percorso unico facendo attenzione all'orientamento delle due strade prima di fonderle. Infatti, in alcune situazioni può essere necessario invertire (con una rotazione) l'ordine di visita dei nodi di uno dei due percorsi. Ad esempio, dovendo collegare le routes [i,5,4,2] e [j,7,3], bisogna prima ruotare uno dei due percorsi. I due risultati possibili sono:

- [3,7,j,i,5,4,2] se ruoto il primo percorso
- [2,4,5,i,j,7,3] se ruoto il secondo

Notiamo che i due percorsi ottenuti sono uguali poichè la matrice delle distanze è simmetrica.
Dopo aver concluso il ciclo sulla matrice dei savings abbiamo ottenuto tutte le strade, dunque inseriamo il deposito all'inizio e alla fine di ogni percorso.

2-opt

Il secondo algoritmo applicato è il 2-opt. Questo è un metodo di ricerca locale che parte dal presupposto di avere a disposizione una soluzione del TSP e prova a migliorarla. Dovendo trattare tuttavia il caso del CVRP, lo applichiamo un percorso alla volta. Nel nostro caso, la soluzione di partenza è quella ottenuta tramite il Clarke-Wright.

L'idea alla base del 2-opt è di esaminare soluzioni alternative ottenute perturbando quella iniziale. Questo è possibile definendo un vicinato della soluzione di partenza. In particolare, dato un tour iniziale, consideriamo **due** qualsiasi archi non consecutivi A=(a,b) e C=(c,d), dove le lettere minuscole rappresentano i nodi. Generiamo nuovi archi "incrociando" i precedenti e ottenendo A'=(a,c) e C'=(b,d). La nuova soluzione si ottiene sostituendo A e C con A', C'. Avendo a disposizione diverse soluzioni alternative viene scelta la migliore in termini di funzione obiettivo, che nel nostro caso è la lunghezza del percorso.
Questo algoritmo può essere generalizzato rimpiazzando k archi, invece che solo 2, e prende il nome di k-opt. Tuttavia abbiamo preferito utilizzare il 2-opt perchè l'aumento dei costi computazionali e della complessità per k>2 non è giustificato da un significativo incremento della qualità della soluzione.
Avendo scelto di implementare ogni percorso con un vettore, per poter effettuare l'"incrocio" e generare nuovi archi invertiamo in parte l'ordine dei nodi nel vettore. Per evitare di prendere archi consecutivi consideriamo tutte le coppie di nodi distanti almeno 2 posizioni. Ad esempio, prendiamo la route [1,2,3,4,1] e applichiamo l'algoritmo selezionando i nodi 2 e 4. Ciò equivale a incrociare gli archi non consecutivi (2,3) e (4,1). Nella pratica questo si traduce in:

- conservare il percorso fino al nodo 2 compreso: [1,2]
- invertire di ordine i nodi tra il 2 (non compreso) e il 4 (compreso): [1,2,4,3]
- conservare il resto del percorso dal nodo 4 (escluso) in poi: [1,2,4,3,1]

Il risultato finale sarà la nuova route [1,2,4,3,1]. Otteniamo così tutti i vicini della configurazione iniziale, tra cui scegliamo il migliore. Questo procedimento viene iterato finchè i vicini di una data soluzione sono tutti peggiori della soluzione corrente in termini di funzione obiettivo. Di conseguenza non ci limitiamo ad esplorare i vicini della soluzione iniziale ma consideriamo anche i vicini dei vicini fino a quando non si ha più un miglioramento.

Analisi risultati

In un primo momento abbiamo deciso di implementare il Clarke-Wright ponendo $\theta = 0$. In questo modo otteniamo i risultati presenti nella prima riga della Tabella 1, ovvero il Caso 0. Possiamo osservare che il metodo iterativo permette di migliorare la soluzione ottenuta col metodo costruttivo. Inoltre, i risultati non si discostano troppo dalla soluzione ottimale: 661.
Proseguendo con l'analisi, abbiamo deciso di impostare una ricerca del miglior iperarametro θ che permetta di minimizzare ulteriormente i costi. A questo proposito è importante distinguere due casi:

- Caso 1: per ogni valore di θ applichiamo in successione Clarke-Wright e 2-opt. Scegliamo il θ che minimizza il risultato finale, ottenendo i valori presenti nella seconda riga della Tabella 1

- Caso 2: per ogni valore di θ applichiamo il Clarke-Wright e scegliamo il valore che minimizza la lunghezza. Solo dopo questo passaggio andiamo ad applicare il 2-opt per migliorare la soluzione corrente ottenendo i risultati presenti nella terza riga della Tabella 1

Valore θ	Risultato Clarke-Wright	Miglioramento con 2-opt
0	918.77	826.68
0.16	788.33	727.41
0.34	773.31	748.37

Tabella 1. Risultati ottenuti

Abbiamo scelto di provare valori di $\theta \in [0, 0.62]$ con un passo di 0.02. Il motivo di questa decisione è che abbiamo osservato che per valori maggiori il metodo di Clarke-Wright restituisce una soluzione con 6 veicoli.
Osserviamo dunque che la miglior soluzione ottenuta è 727.41, molto vicina all'ottimo globale 661, nonostante i metodi utilizzati siano delle euristiche. Inoltre notiamo come la soluzione migliore non si ottiene applicando il 2-opt al miglior risultato trovato dal Clarke-Wright (riga 3 Tabella 1), ma applicando il 2-opt ad una soluzione leggermente peggiore (riga 2 Tabella 1).
Infine, andando ad analizzare nello specifico le lunghezze dei percorsi dei 5 veicoli nel caso 1 e 2 (vedi Tabella 2), osserviamo che i path associati alla miglior soluzione (caso 1) hanno una lunghezza più disomogenea. In un contesto reale questo potrebbe portare a preferire la soluzione del caso 2, nonostante costi leggermente di più, per garantire maggiore uniformità tra i percorsi assegnati ai vari veicoli. I risultati grafici confermano quanto detto.

Numero percorso	Caso 1	Caso 2
1	63.21	233.14
2	233.33	106.90
3	204.07	135.25
4	135.25	179.35
5	91.54	93.74

Tabella 2. Confronto casi 1 e 2

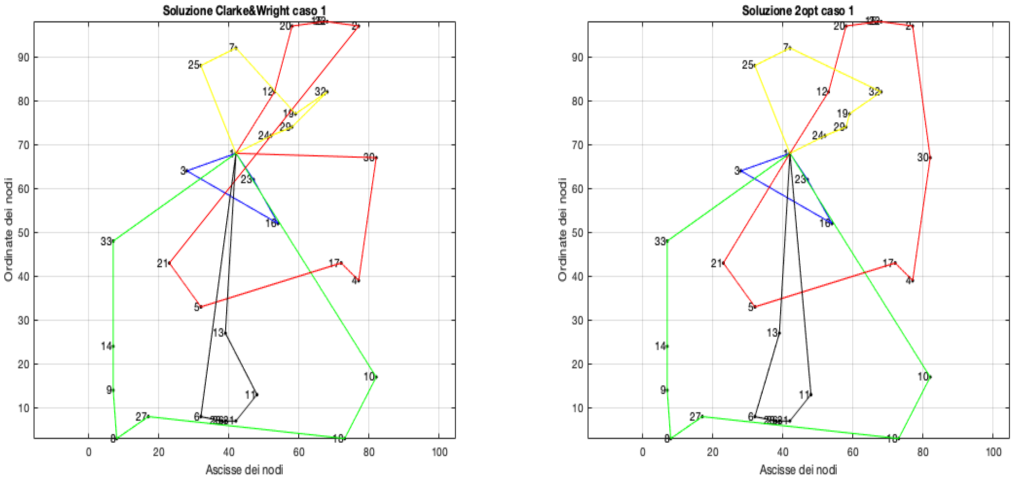

Figura 1. Soluzione caso 1, $\theta = 0.16$

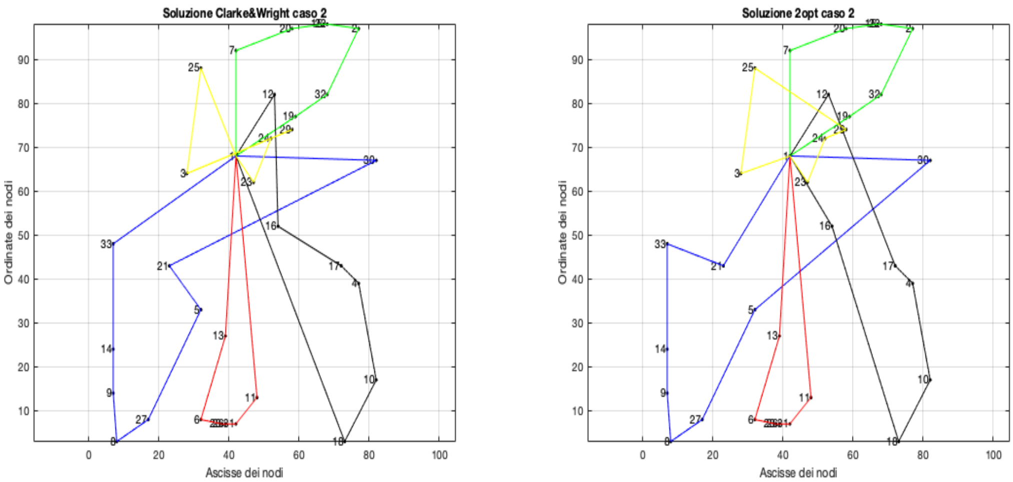

Figura 2. Soluzione caso 2, $\theta = 0.34$

Appendice A

Codice MATLAB

A.1 Main script

```
clear all
close all
clc

%Caricamento dataset
load('dataset.mat')

%Calcolo matrice delle distanze
distance=dist(coord');

%Inizializzazioni
ris_clark_wright = cell(32,1);
sum_clark_wright = zeros(32,1);
ris_2opt = cell(32,1);
sum_2opt = zeros(32,1);

%Ricerca del miglior iperparametro theta
for k = 1:32  %per valori di k maggiori otteniamo 6 strade

    %Metodo costruttivo Clarke-Wright
    routes=clarke_wright(distance,vehicle_capacity,demand,0.02*(k-1));
    dist_cw = zeros(length(routes),1);
    for t = 1:length(routes)
        dist_cw(t)=lunghezza_percorso(routes{t},distance);
    end
    %Risultati lunghezza percorsi
    ris_clark_wright{k} = routes;
    sum_clark_wright(k) = sum(dist_cw);

    %Metodo iterativo 2-opt
    routes_2opt=cell(length(routes),1);
    dist_2opt=zeros(length(routes),1);
```

```
    for i=1:length(routes)
        [routes_2opt{i},dist_2opt(i)]=two_opt(routes{i},distance);
    end
    %Risultati lughezza percorsi
    ris_2opt{k} = routes_2opt;
    sum_2opt(k) = sum(dist_2opt);
end


%%%%%%%%%%%%
%%% PLOT %%%
%%%%%%%%%%%%

%Troviamo i migliori risultati in generale (CASO 1)
[min_2opt,best] = min(sum_2opt);
routes = ris_clark_wright{best};
routes_2opt = ris_2opt{best};

%Plot dei risultati Clarke&Wright caso 1
names = ...
    {'1','2','3','4','5','6','7','8','9','10','11','12','13','14', ...
    '15','16','17','18','19','20','21','22', '23', '24', '25', '26', ...
    '27', '28', '29', '30', '31', '32', '33'};
color=['b', 'g', 'r', 'k', 'y'];
p1 = plot(coord(:,1),coord(:,2), 'k.', 'DisplayName','Nodi');
hold on
text(coord(:,1) - 0.25, coord(:,2) + 0.25, names, ...
    'HorizontalAlignment', 'right');
axis equal
hold on
for k=1:length(routes)
    for i = 1:length(routes{k})-1
        p2(i,i+1) = plot([coord(routes{k}(i),1), ...
            coord(routes{k}(i+1),1)], [coord(routes{k}(i),2), ...
            coord(routes{k}(i+1),2)], color(k), ...
            'DisplayName','Clarke&Wright');
        hold on
    end
end
title('Soluzione Clarke&Wright caso 1')
grid on
xlabel('Ascisse dei nodi')
ylabel('Ordinate dei nodi')

%Plot dei risultati 2opt caso 1
figure(2)
p1 = plot(coord(:,1),coord(:,2), 'k.', 'DisplayName','Nodi');
hold on
text(coord(:,1) - 0.25, coord(:,2) + 0.25, names, ...
    'HorizontalAlignment', 'right');
axis equal
hold on
for k=1:length(routes_2opt)
```

```
    for i = 1:length(routes_2opt{k})-1
        p2(i,i+1) = plot([coord(routes_2opt{k}(i),1), ...
            coord(routes_2opt{k}(i+1),1)], ...
            [coord(routes_2opt{k}(i),2), ...
            coord(routes_2opt{k}(i+1),2)], color(k), ...
            'DisplayName','2opt');
        hold on
    end
end
title('Soluzione 2opt caso 1')
grid on
xlabel('Ascisse dei nodi')
ylabel('Ordinate dei nodi')

%Troviamo miglior risultato clark_wright e applichiamo 2_opt (CASO 2)
[min_clark_wright,best2] = min(sum_clark_wright);
routes = ris_clark_wright{best2};
routes_2opt = ris_2opt{best2};

%Plot dei risultati Clarke&Wright caso 2
figure(3)
p1 = plot(coord(:,1),coord(:,2), 'k.', 'DisplayName','Nodi');
hold on
text(coord(:,1) - 0.25, coord(:,2) + 0.25, names, ...
    'HorizontalAlignment', 'right');
axis equal
hold on
for k=1:length(routes)
    for i = 1:length(routes{k})-1
        p3(i,i+1) = plot([coord(routes{k}(i),1), ...
            coord(routes{k}(i+1),1)], [coord(routes{k}(i),2), ...
            coord(routes{k}(i+1),2)], color(k), ...
            'DisplayName','Clarke&Wright');
        hold on
    end
end
title('Soluzione Clarke&Wright caso 2')
grid on
xlabel('Ascisse dei nodi')
ylabel('Ordinate dei nodi')

%Plot dei risultati 2opt caso 2
figure(4)
p1 = plot(coord(:,1),coord(:,2), 'k.', 'DisplayName','Nodi');
hold on
text(coord(:,1) - 0.25, coord(:,2) + 0.25, names, ...
    'HorizontalAlignment', 'right');
axis equal
hold on
for k=1:length(routes_2opt)
    for i = 1:length(routes_2opt{k})-1
```

```
        p4(i,i+1) = plot([coord(routes_2opt{k}(i),1), ...
            coord(routes_2opt{k}(i+1),1)], ...
            [coord(routes_2opt{k}(i),2), ...
            coord(routes_2opt{k}(i+1),2)], color(k), ...
            'DisplayName','2opt');
        hold on
    end
end
title('Soluzione 2opt caso 2')
grid on
xlabel('Ascisse dei nodi')
ylabel('Ordinate dei nodi')


%Calcolo lunghezza singoli percorsi nei casi 1 e 2 (per Tabella 2)
for i = 1:5
    caso_1(i) = lunghezza_percorso(ris_2opt{best,1}{i,1}, distance);
    caso_2(i) = lunghezza_percorso(ris_2opt{best2,1}{i,1}, distance);
end
```

A.2 Funzione clarke_wright

```
function [routes]=clarke_wright(distance,vehicle_capacity,demand,theta)
%
% [routes]=clarke_wright(distance,vehicle_capacity,demand)
%
% Metodo costruttivo di Clarke-Wright per la ricerca di una ...
    soluzione sub
% ottimale del problema CVRP. Algoritmo parallelo basato sul
% criterio dei savings
%
% INPUTS:
% distance = matrice delle distanx ze fra i nodi
% vehicle_capacity = valore scalare della capacit  di ogni veicolo
% demand = vettore delle domande di ogni nodo
% theta = parametro di penalizzazione strade lunghe
%
% OUTPUTS:
% routes = cell-array contenente i vettori dei diversi percorsi ottenuti

%Se non passo theta lo suppongo nullo
if nargin == 3
    theta = 0;
end

%Trovo numero nodi
dimension=size(distance,1);

%Creo e inizializzo cell-array delle routes
routes = cell(dimension-1,1);
```

```
for i = 1:dimension-1
    routes{i} = i+1;
end

%Creo matrice savings
savings = zeros(dimension, dimension);
for i = 2:dimension
    for j = i+1:dimension
        savings(i,j) = distance(i,1) + distance(1,j) - ...
            (theta+1)*distance(i,j);
    end
end

%Inizializzazione
path_1 = 0;
path_2 = 0;
k = 0;  %numero iterazioni

%counter violazione vincoli
capacita_superata = 0;
stesso_percorso = 0;
non_estremali = 0;

%Ciclo fino a che la matrice dei savings non    nulla
while max(savings,[],'all') > 0

    k = k+1;

    %Trovo miglior saving
    max_save = max(savings,[],'all');
    [i,j] = find(savings==max_save);

    %Gestione ties
    i = i(1);
    j = j(1);

    %Setto a 0 il saving considerato
    savings(i,j) = 0;

    %Cerco se ci sono due feasible path che contengono i e j
    for el = 1:length(routes)
        if routes{el}(1) == i
            path_1 = routes{el};
            flag_1 = 0;  %flag se i si trova a inizio (0) o fine (1) ...
                path
            pos_1 = el;
        elseif routes{el}(end) == i
            path_1 = routes{el};
            flag_1 = 1;
            pos_1 = el;
        elseif routes{el}(1) == j
            path_2 = routes{el};
            flag_2 = 0; %flag se j si trova a inizio (0) o fine (1) path
```

```
            pos_2 = el;
        elseif routes{el}(end) == j
            path_2 = routes{el};
            flag_2 = 1;
            pos_2 = el;
        end
    end

    % Controllo se i e j sono nodi estremali di percorsi
    if sum(path_1) == 0 || sum(path_2) == 0
        non_estremali = non_estremali+1;
        continue
    end

    %Controllo se i e j appartengono allo stesso percorso
    if path_1(1) == path_2(1) || path_1(1) == path_2(end)
        stesso_percorso = stesso_percorso+1;
        continue
    end

    %Controllo vincolo capacit
    if sum(demand(path_1)) + sum(demand(path_2)) > vehicle_capacity
        capacita_superata = capacita_superata +1;
        continue
    end

    %Unione path di i e j in base ai casi
    if flag_1 == 0
        if flag_2 == 0
            path_2 = rot90(path_2,2);  %ruoto il path_2 perch  ...
                voglio che j sia a fine del path 2 (essendo i ...
                all'inizio del path 1)
            new_path = path_2;
            new_path(end+1:end+length(path_1)) = path_1;
            routes{pos_1} = new_path;
            routes(pos_2) = [];
        elseif flag_2 == 1
            new_path = path_2;
            new_path(end+1:end+length(path_1)) = path_1;
            routes{pos_1} = new_path;
            routes(pos_2) = [];
        end
    elseif flag_1 == 1
        if flag_2 == 0
            new_path = path_1;
            new_path(end+1:end+length(path_2)) = path_2;
            routes{pos_1} = new_path;
            routes(pos_2) = [];
        elseif flag_2 == 1
            path_2 = rot90(path_2,2);
            new_path = path_1;
            new_path(end+1:end+length(path_2)) = path_2;
            routes{pos_1} = new_path;
```

```
            routes(pos_2) = [];
        end
    end

    %Reset dei path
    path_1 = 0;
    path_2 = 0;
end

%Aggiungiamo il deposito a inizio e fine di ogni percorso
for i=1:length(routes)
    routes{i}=[1 routes{i} 1];
end

end
```

A.3 Funzione two_opt

```
function [percorso_out, min_length] = two_opt(percorso,matdist)
%
% [percorso_out, min_length] = two_opt(percorso,matdist)
%
% Metodo iterativo di ricerca locale 2-opt per il miglioramento di una
% soluzione del problema TSP
%
% INPUTS:
% percorso = vettore contenente i nodi di un percorso chiuso
% matdist = matrice delle distanze tra i nodi
%
% OUTPUTS:
% percorso_out = vettore contenente i nodi del percorso ottimizzato
% min_length = lunghezza percorso ottimizzato

%Setto flag per il criterio di stop
trovato_nuovo=1;

while trovato_nuovo
    %Inizializzo matrice dei neighbours
    nuovo_percorso=percorso;
    %Ciclo per trovare tutti i neighbours di un percorso
    for i=1:(length(percorso)-3)
        for j=i+2:(length(percorso)-1)
            %evito ciclo inverso
            if i==1 && j==length(percorso)-1
                continue
            end
            %Aggiungo il neighbour trovato alla lista dei neighbours
            nuovo_percorso(end+1,:)=[percorso(1:i) ...
                rot90(percorso(i+1:j),2) percorso(j+1:end)];
        end
```

```
    end

    %Calcolo lunghezza dei percorsi neighbours
    dist_2opt=zeros(size(nuovo_percorso,1),1);
    for i=1:size(nuovo_percorso,1)
        dist_2opt(i)=lunghezza_percorso(nuovo_percorso(i,:),matdist);
    end

    %Trovo il percorso con lunghezza minima
    [min_length, index_min] = min(dist_2opt);
    percorso=nuovo_percorso(index_min,:);

    %Esco quando i percorsi non migliorano p i
    if index_min==1
        trovato_nuovo=0;
        percorso_out = percorso;
    end
end

end
```

A.4 Funzione lunghezza_percorso

```
function [dist] = lunghezza_percorso(percorso,mat_dist)
%
% [dist] = lunghezza_percorso(percorso,mat_dist)
%
% Funzione per il calcolo della lunghezza di un percorso
%
% INPUTS:
% percorso = vettore contenente i nodi del percorso chiuso di cui si ...
    vuole
% calcolare la lunghezza
% mat_dist = matrice delle distanze dei nodi
%
% OUTPUTS:
% dist = lunghezza del percorso

dist=0;

for i = 1:length(percorso)-1
    dist= dist + mat_dist(percorso(i),percorso(i+1));
end

end
```