Homework II
CVRP

Gabriele Galilei 302699
Arianna Abis 303876
Riccardo Kiefer 301286

14 maggio 2026

1 Introduzione

Per risolvere il problema CVRP sotto le ipotesi indicate, ovvero:

- Capacità monodimensionale
- Veicoli identici
- Problema simmetrico

abbiamo scelto di proporre due metodi costruttivi:

- Un metodo *cluster-first, route-second*, ovvero la combinazione di un metodo *Sweep* per la divisione in cluster e di un'euristica greedy *Nearest Neighbor*, che risolva il problema *TSP* internamente ad ogni cluster.
- Un metodo parallelo (più route vengono create contemporaneamente) che utilizza un criterio di *saving*, in particolare l'algoritmo proposto da Clarke-Wright nel 1964 nel loro articolo "*Scheduling of Vehicles from a Central Depot to a Number of Delivery Points"*.

Entrambi i metodi vengono migliorati impiegando il metodo iterativo meta-euristico *Tabu Search*. Si è preferito usare come numero di veicoli il minor numero di veicoli necessari a coprire la domanda di tutti i nodi e non un numero fisso deciso a

priori: tale decisione è dovuta al fatto che non si conosce il trade-off tra l'introduzione di un nuovo veicolo e il soddisfacimento della domanda. Inoltre, si è assunta la capacità dei veicoli come non minore del massimo delle domande, in modo tale che ogni veicolo possa soddisfare la domanda di ogni punto singolarmente.

Segue una spiegazione dei metodi utilizzati, il cui codice è interamente riportato in appendice all'elaborato.

1.a Sweeping Method & Nearest Neighbour heuristic

1.a.1 Sweep Method

Il metodo cosiddetto *sweep* viene proposto da Gillet e Miller nel loro articolo del 1974 "*A Heuristic Algorithm for the Vehicle Dispatch Problem*" ed è basato su un ragionamento geometrico, per cui si disegna un raggio che dal deposito spazza i clienti in un ordine che dipende dalla loro posizione: i clienti vengono assegnati ad un cluster fino a quando non viene superata la capacità data. Nella nostra implementazione la funzione *SweepClustering_cap* (A) prende in input le coordinate dei nodi che rappresentano i punti di ritiro, il valore della capacità di ogni singolo veicolo, il vettore delle domande di ritiro dei clienti e le coordinate del centro (anche se abbiamo supposto essere (0,0)). I clienti vengono ordinati in maniera crescente considerando la distanza angolare in senso antiorario dal deposito e si procede con la divisione in cluster come indicato precedentemente. Alla fine della funzione si taglia la matrice dei cluster in base al numero di cluster effettivamente creati e alla loro lunghezza effettiva.

1.a.2 NN heuristic

Una volta individuati i cluster, per ognuno di questi abbiamo implementato l'euristica più semplice possibile nella funzione *NN heuristic* (B) che a partire da un nodo all'interno di ogni cluster sceglie il successivo come il più vicino tra quelli disponibili. La funzione prende in input due vettori di coordinate x e y, supponendo che il deposito si trovi in (x(1), y(1)), e restituisce il vettore colonna corrispondente alla sequenza ordinata dei nodi da visitare a partire dal deposito e il valore del costo, ottenuto come somma delle distanze percorse. Poiché supponiamo sempre distanza euclidea utilizziamo una funzione ausiliaria (C) per il calcolo delle distanze.

1.b Savings criterion

L'idea principale dietro la funzione *Saving_boost* (D) è quella di partire da tante route quanti sono i nodi e diminuirne progressivamente il numero procedendo ad unire tra loro le route secondo un criterio di saving: ovvero dati due nodi i e j che si trovino agli estremi di due route diverse (entrambi testa, entrambi coda, uno testa e uno coda), calcoliamo il *saving* corrispondente come $c_{0i} + c_{i0} + c_{0j} + c_{j0} - c_{0i} - c_{ij} - c_{j0} = c_{i0} + c_{0j} - c_{ij}$ dove con 0 indichiamo il deposito e stiamo sommando le distanze tra i due nodi e il deposito e sottraendo la distanza tra i due nodi. Ovviamente procediamo ad unire le route in ordine decrescente di *saving*, facendo ogni volta un controllo sul rispetto della capacità dei veicoli. Alla funzione diamo in input le coordinate dei punti di ritiro e del deposito, il vettore delle domande nei punti di ritiro e il valore scalare della capacità dei veicoli disponibili. La funzione restituisce una matrice che contiene i nodi appartenenti ad ogni route sulle righe, un vettore che contiene il costo relativo ad ogni route, un vettore che contiene la lunghezza di ogni route, e infine un vettore con la capacità percentuale occupata per ogni veicolo. Per calcolare i costi viene utilizzata la funzione *RouteLength* (E).

1.c Tabu Search

Nella funzione *TabuSearch* (F) implementiamo un metodo iterativo con lo scopo di migliorare in termini di costo complessivo i risultati ottenuti con i metodi costruttivi. L'algoritmo prende in input le route ottenute con *sweep+NN* / *saving* e si occupa di migliorarle internamente (non sono previste modifiche inter route) attraverso i metodi proposti nella funzione *CreatePermActionList* (G), che prende in input un modello TSP (H) e crea una cell array di matrici che salva tutte le possibili azioni di modifica di una route: scambio di due indici, inversione della parte di route compresa tra due indici la cui distanza sia maggiore di uno, inserimento di un nodo successivamente ad un altro. Per ogni route si scorre la lista di azioni, si eseguono e se ne calcola il costo. Per capire a quale delle tre azioni corrisponde l'elemento della lista, ed effettivamente applicarla ad una coppia di nodi, si usano le 4 funzioni: *DoAction* (I), *DoSwap* (J), *DoReversion* (K), *DoInsertion* (L).

Ogni volta che il compimento di una determinata azione su una route porta ad un miglioramento del costo che la rende la nuova soluzione migliore disponibile, quell'azione viene bloccata per un numero predefinito di iterazioni. In questo modo si ammettono azioni peggioranti, che permettono di spostarsi da minimi locali, evitando però di ricadere negli stessi successivamente.

2 Risultati computazionali e confronto tra metodi

2.a Applicazione ad un dataset randomico

Per svolgere simulazioni abbiamo scelto di generare i punti da visitare e la domanda in modo randomico; in particolare si è scelto di simulare su 100 punti, generati uniformemente in [-15,30]x[-15,30], veicoli con capacità 300 unità e domanda generata uniformemente in [1,35]; il numero massimo di iterazioni per la Tabu Search è stato fissato a 50, mentre la Tabu Length è massima.

2.a.1 Sweep e Nearest Neighbour

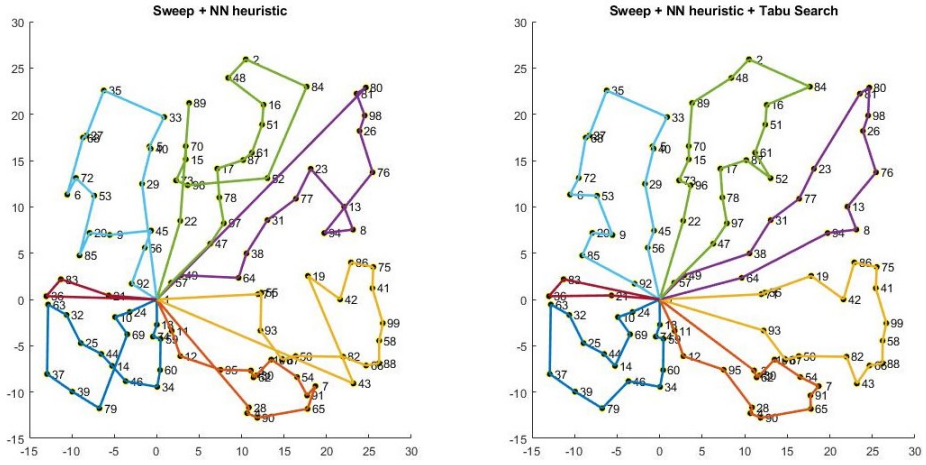

Figura 1: Rappresentazione delle route risultanti da 1) sweep e NN e 2) sweep, NN e Tabu search

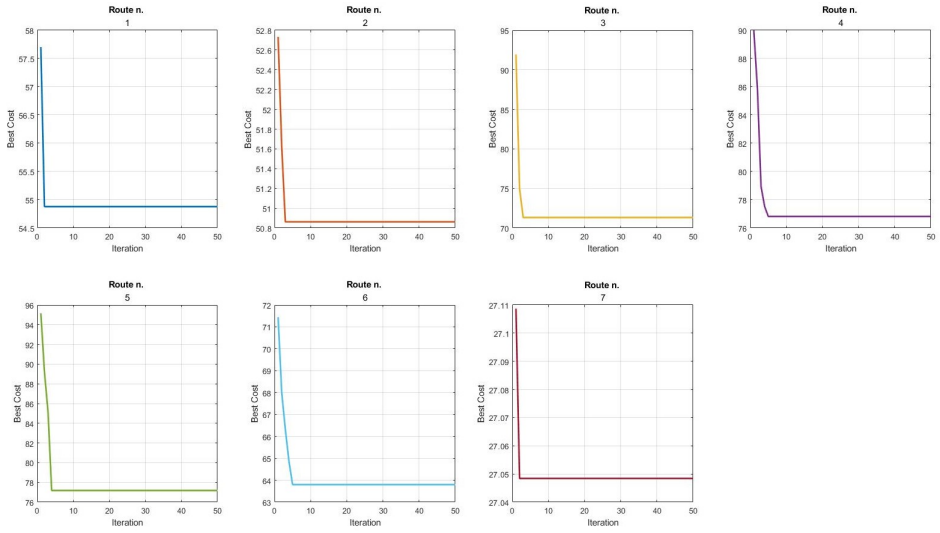

Figura 2: Costi delle singole route al variare delle iterazioni di Tabu Search

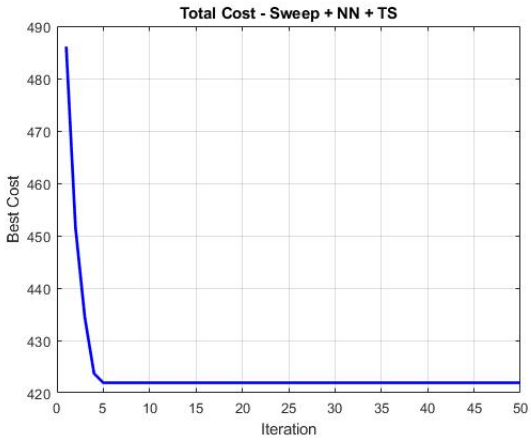

Figura 3: Costo totale al variare delle iterazioni di Tabu Search

route n.	Costo pre-TS	Costo post-TS	Diminuzione percentuale
1	57.7025	54.8789	4.8932
2	52.7331	50.8627	3.5470
3	91.9393	71.3288	22.4176
4	89.9976	76.8199	14.6422
5	95.1629	77.1816	18.8953
6	71.4444	63.7991	10.7011
7	27.1087	27.0484	0.2224
tot	486.0885	421.9194	13.2011

Tabella 1: Tabella dei costi risultanti da
1) sweep e NN 2) sweep, NN e TS 3) percentuale di diminuzione.

route n.	1	2	3	4	5	6	7
costs	98.1514	97.4958	96.8279	97.0596	97.9788	90.1052	31.1071

Tabella 2: Tabella delle percentuali di capacità occupata per route.

Sweep & NN	Sweep & NN & TS
0.031195 s	0.73426 s

Tabella 3: Tabella dei tempi di calcolo impiegati.

2.a.2 Savings criterion

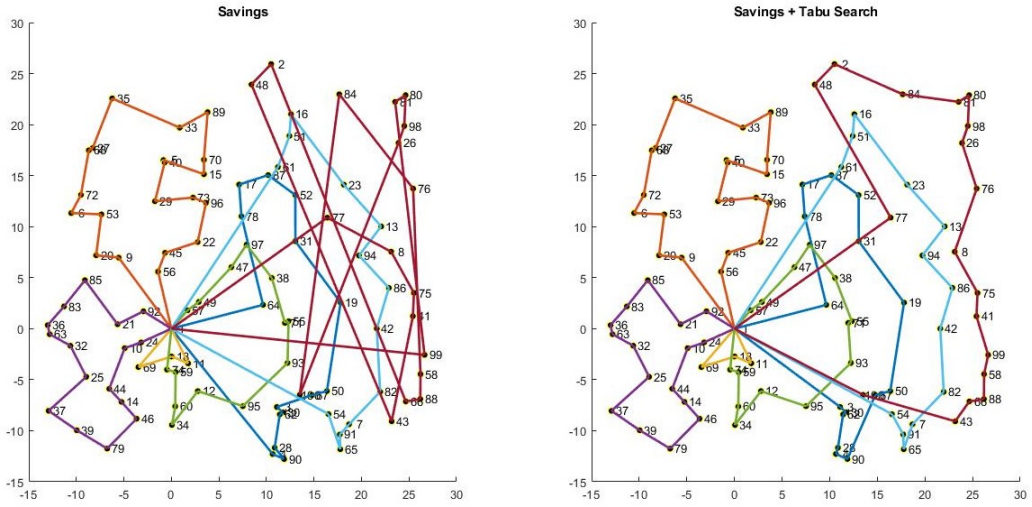

Figura 4: Rappresentazione delle route risultanti da 1) savings e 2) savings e Tabu search

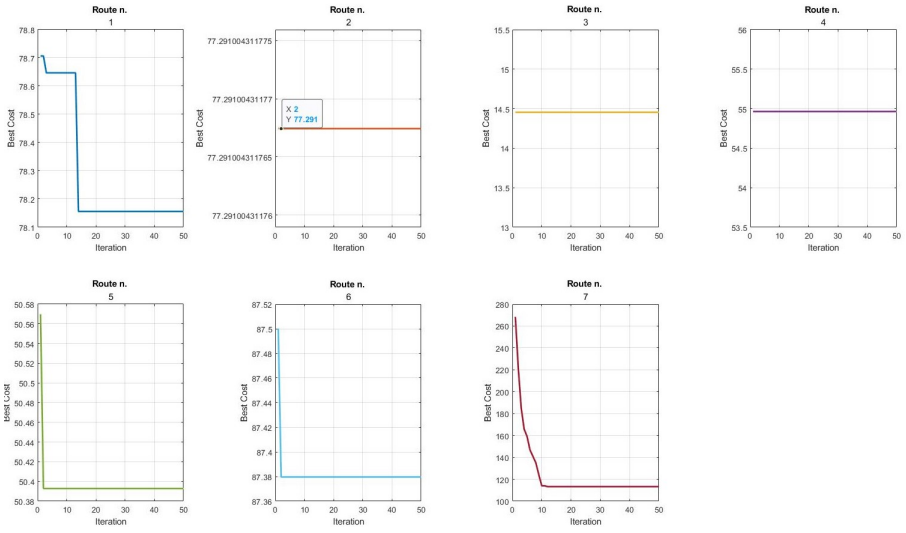

Figura 5: Costi delle singole route al variare delle iterazioni di Tabu Search

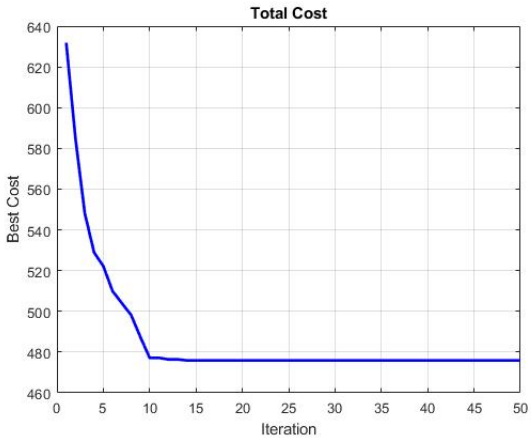

Figura 6: Costo totale al variare delle iterazioni di Tabu Search

route n.	Costo pre-TS	Costo post-TS	Diminuzione percentuale
1	78.7066	78.1560	0.6996
2	77.2910	77.2910	0.0000
3	14.4546	14.4546	0.0000
4	54.9647	54.9647	0.0000
5	50.5698	50.3928	0.3500
6	97.5002	87.3796	0.1377
7	268.3461	113.2753	57.7876
tot	631.8330	475.9141	24.6772

Tabella 4: Tabella dei costi risultanti da
1) savings 2) savings e TS 3) percentuale di diminuzione.

route n.	1	2	3	4	5	6	7
costs	98.2739	95.4703	19.8581	98.0585	97.4297	99.8074	99.8279

Tabella 5: Tabella delle percentuali di capacità occupata per route.

Savings	Savings & TS
0.22566 s	1.0127 s

Tabella 6: Tabella dei tempi di calcolo impiegati.

2.a.3 Risultati conclusivi

Possiamo innanzitutto notare come l'applicazione di Tabu Search aumenta notevolmente i tempi computazionali, soprattutto nel caso dell'algoritmo di sweeping e NN.

In entrambi i casi il numero di veicoli richiesto per soddisfare la domanda è sette, questo ci permette di focalizzarci sui soli costi delle route per effettuare un confronto:

Sweep & NN	Sweep & NN & TS	Savings	Savings & TS
486.0885	421.9194	631.8330	475.9141

Tabella 7: Tabella dei costi.

Abbiamo osservato come, in generale, algoritmi su base *cluster-first*, *route-second* operino meglio di algoritmi senza clustering: si è pensato che ciò possa essere dovuto al numero di nodi piccolo ed alla omogeneità dei veicoli, in quanto tutte le route possono essere eseguite da tutti i veicoli.

Per la improvement heuristic Tabu Search, notiamo come questa migliori le soluzioni per entrambe le basi algoritmiche, portando un notevole risparmio nel caso del *savings criterion*, dove una route iniziale era considerevolmente non ottimale.

I tempi computazionali rimangono moderatamente bassi anche nel caso dell'aggiunta di Tabu Search, anche per un numero di nodi più grande, quindi è sempre preferibile utilizzarlo.

2.b Applicazione su un dataset non randomico

I test effettuati finora sono stati svolti su nodi generati in maniera randomica. Per avere un elemento di confronto, tuttavia, abbiamo voluto effettuare delle verifiche su dataset "reali" contenuti all'interno di librerie online (`http://vrp.atd-lab.inf.puc-rio.br/index.php/en/`). Per tutti i dataset scelti, oltre alla lista dei nodi, della domanda di ogni nodo e della capacità dei veicoli era riportato anche il valore della soluzione ottima da utilizzare come benchmark per i nostri esperimenti numerici.
Vediamone i risultati.

2.b.1 Istance: E-n101-k8 (Christophides and Eilon, Min no of trucks: 8, Best value: 815)

`http://vrp.atd-lab.inf.puc-rio.br/index.php/en/plotted-instances?data=E-n101-k8`

1. **Ottimo**

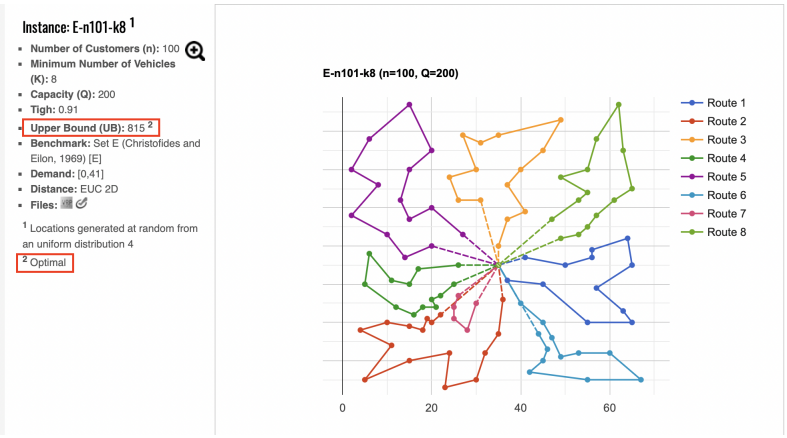

Figura 7: Rappresentazione della soluzione ottima

2. **Sweep e Nearest Neighbour**

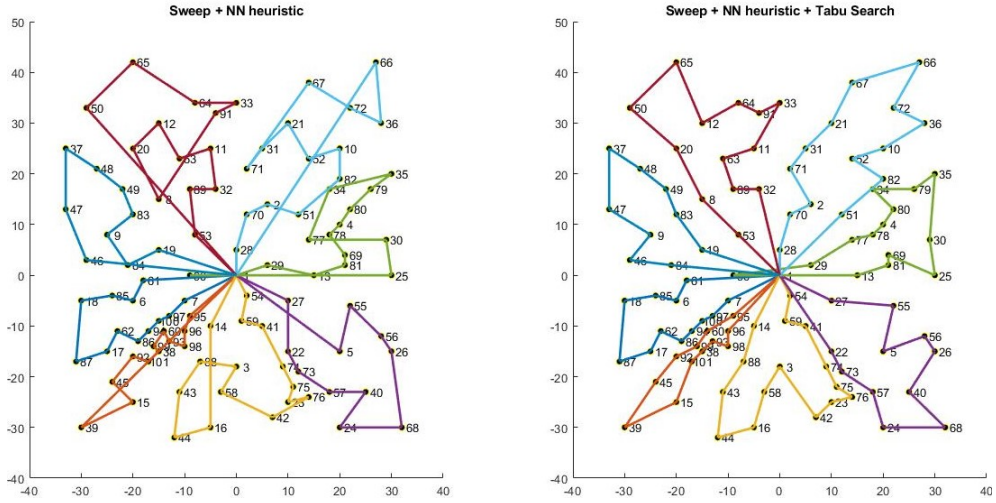

Figura 8: Rappresentazione delle route risultanti da 1) sweep e NN e 2) sweep, NN e Tabu search

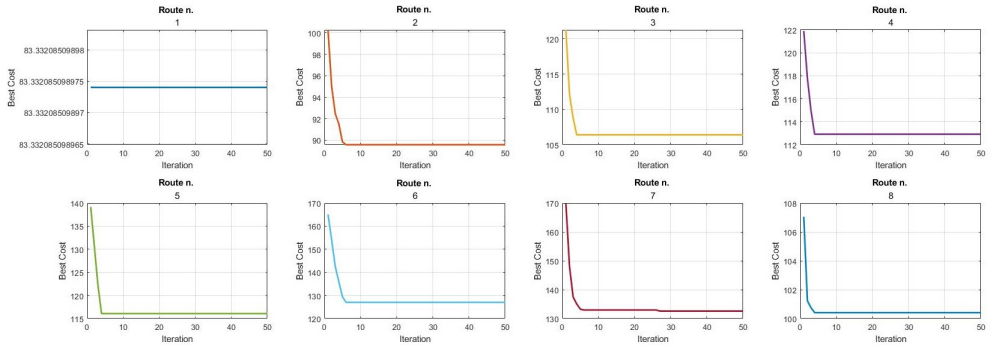

Figura 9: Costi delle singole route al variare delle iterazioni di Tabu Search

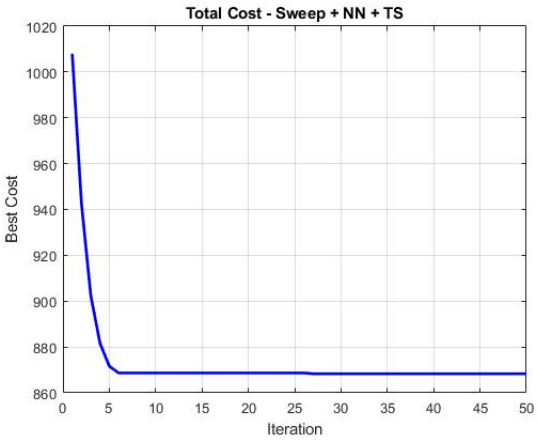

Figura 10: Costo totale al variare delle iterazioni di Tabu Search

route n.	Costo pre-TS	Costo post-TS	Diminuzione percentuale
1	83.3321	83.3321	0.0000
2	100.2794	89.5796	10.6700
3	121.2676	106.4066	12.2547
4	121.9090	112.9156	7.3772
5	139.1702	116.0636	16.6032
6	165.0872	127.0055	23.0676
7	169.8721	132.5786	21.9539
8	107.0583	100.4075	6.2123
tot	1007.9759	868.2891	13.8581

Tabella 8: Tabella dei costi risultanti da
1) sweep e NN 2) sweep, NN e TS 3) percentuale di diminuzione.

route n.	1	2	3	4	5	6	7	8
costs	95.5	89.5	91.0	94.5	97.5	100.0	94.5	66.5

Tabella 9: Tabella delle percentuali di capacità occupata per route.

Sweep & NN	Sweep & NN & TS
0.034482 s	0.78452 s

Tabella 10: Tabella dei tempi di calcolo impiegati.

3. **Savings**

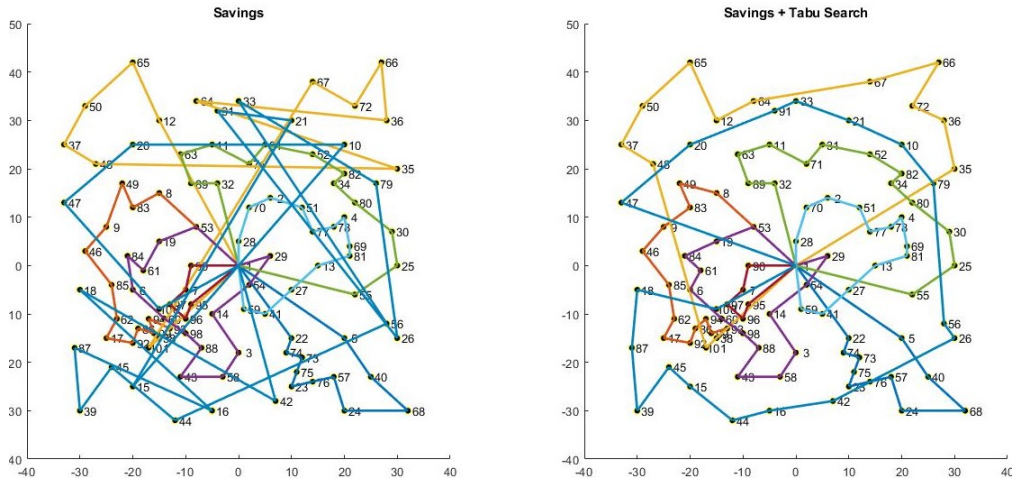

Figura 11: Rappresentazione delle route risultanti da 1) Saving e 2) Saving e Tabu search

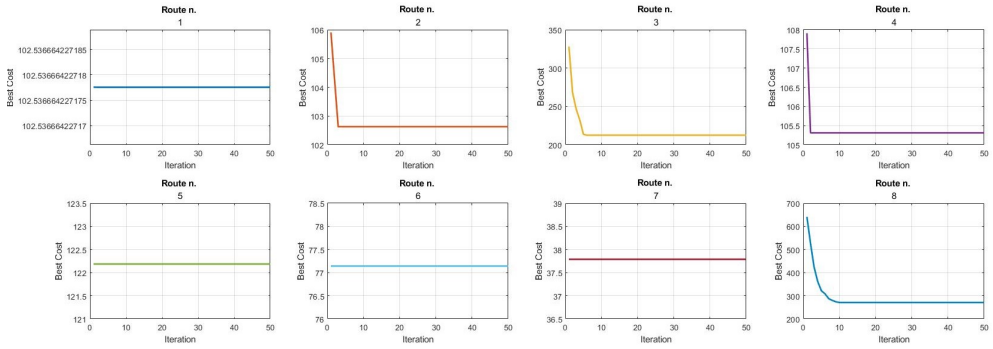

Figura 12: Costi delle singole route al variare delle iterazioni di Tabu Search

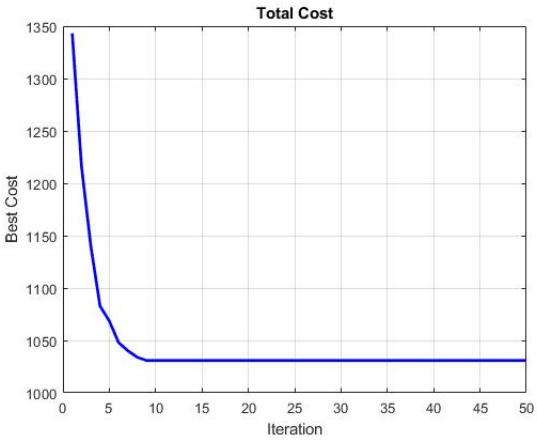

Figura 13: Costo totale al variare delle iterazioni di Tabu Search

route n.	Costo pre-TS	Costo post-TS	Diminuzione percentuale
1	102.5367	102.5367	0.0000
2	105.9076	102.6293	3.0954
3	328.0992	212.5005	35.2328
4	107.9106	105.3116	2.4084
5	122.1857	122.1857	0.0000
6	77.1414	77.1414	0.0000
7	37.7882	37.7882	0.0000
8	641.3142	270.7876	57.7761
tot	1522.8836	1030.8809	32.3073

Tabella 11: Tabella dei costi risultanti da
1) savings 2) savings e TS 3) percentuale di diminuzione.

route n.	1	2	3	4	5	6	7	8
costs	99.5	98.5	100.0	99.5	98.5	95.0	38.0	100.0

Tabella 12: Tabella delle percentuali di capacità occupata per route.

Savings	Savings & TS
0.35682 s	1.2030 s

Tabella 13: Tabella dei tempi di calcolo impiegati.

2.b.2 Considerazioni finali

Come primo risultato interessante notiamo che il numero di veicoli richiesti per soddisfare la domanda coincide con il numero di veicoli utilizzati nella soluzione ottima. Inoltre, i osserva come l'applicazione di tabu search produca soluzioni che si avvicinano notevolmente all'ottimo, sopratutto nel metodo *cluster-first, route-second* dove arriviamo ad un differenza tra costo totale ed ottimo inferiore alla centinaia. Ciò a conferma del fatto che gli algoritmi che si basano su questo principio lavorano meglio nel caso di veicoli omogenei e numero di nodi non troppo elevato.

Ottimo	Sweep & NN	Sweep & NN & TS	Savings	Savings & TS
815.0000	1007.9759	868.2891	1522.8836	1030.8809
Differenza	192.9759	53.2890	707.8835	215.8809

Tabella 14: Tabella della differenza tra i costi totali dei vari metodi e l'ottimo.

A Sweep

Funzione che implementa il metodo *sweep*:

```
function [clusters, lengths] = SweepClustering_cap(x, y,
    capacity, collections, center)
%INPUTS:
%x: vettore che contiene le coordinate x dei punti di ritiro
%y: vettore che contiene le coordinate y dei punti di ritiro
%capacity: scalare per la capacit  dei veicoli
%collections: vettore che contiene la domanda per ogni veicolo
%center: coordinate del deposito

%OUTPUTS:
%clusters: matrice di tante righe quanti sono i cluster, ogni
    riga contiene i nodi di un cluster
%lengths: vettore che riporta le lunghezze di ogni cluster


%%      INIZIALIZZAZIONE
n = length(x);
clusters = zeros(n,n); % matrice che ha per ogni riga i clusters
lengths = zeros(n,1); % vettore che ha per ogni entry i la
    lunghezza del cluster i
occ_cap = zeros(n,1); % vettore cha ha per ogni entry i la
    capacit  occupata per il veicolo che serve il cluster i

x = x(:);
y = y(:);
center = center(:);

%%      CALCOLO DEGLI ANGOLI IN [-pi,pi]
angles = atan((y(:)-ones(n,1).*center(2))./(x(:)-ones(n,1).*
    center(1)));

for i = 1:n
    if x(i) < 0 && y(i) < 0
        angles(i) = angles(i) - pi;
    elseif x(i) < 0 && y(i) > 0
        angles(i) = angles(i) +pi;
    end
end

% ordinamento degli angoli (crescente)
[~,ind] = sort(angles);

```

```
38
39 %%      DIVISIONE IN CLUSTER
40
41 i = 1; % contatore dei cluster
42 j = 1; % contatore dei nodi ordinati per angolo
43 while i<=n && j<=n
44     cont = 0; % contatore del numero di nodi inseriti nel
       cluster i
45     while occ_cap(i) + collections(ind(j)) <= capacity && j<=n
46         % controllo sulla capacit
47         cont = cont+1;
48         clusters(i,cont) = ind(j); % inserimento nel cluster
49         occ_cap(i) = occ_cap(i) + collections(ind(j)); %
       aggiornamento della capacit  occupata
50         j = j+1;
51         if j == n+1
52             break;
53         end
54     end
55     lengths(i) = cont; % lunghezza del cluster i
56     i=i+1;
57 end
58
59 %%      OUTPUT
60
61 clusters = clusters(1:i,1:max(lengths)+1);
62 for j = 1:i
63     clusters(j,lengths(j)+1)=0;
64 end
65
66 lengths = lengths+ones(size(lengths,1),1); % aggiungiamo 1 alla
       lunghezza per tener conto dell'origine
67 end
```

B NN

Funzione che implementa l'euristica *NN*:

```
1 function [route, cost] = NNheuristic(x, y)
2 %INPUTS:
3 %x: vettore delle coordinate x dei nodi del cluster
4 %y: vettore delle coordinate y dei nodi del cluster
5 %OUTPUTS:
6 %route: vettore ordinato dei nodi da visitare in sequenza
```

```
%cost: scalare che rappresenta il costo totale
%%      INIZIALIZZAIONE
% Il deposito sia in posizione 1 sia nelle x che nelle y
n = length(x);
distances = distanceMatrix(x,y);
w = distances(:,1); % salviamo perch  in seguito verr
    modificata
route = zeros(n+1,1);
route(1) = 1;

%%      RICERCA DEL NEAREST NEIGHBOUR
cost = 0;
for i = 1:n
    distances(i,i) = NaN;
end

for i = 2:n
    % ricerca del nodo pi  vicino all'ultimo inserito in route
    [add_cost,ind] = min(distances(route(i-1),:),[],'omitnan');
    route(i) = ind;
    % si deve far s  che il nodo inserito non sia pi
    considerato
    distances(route(i-1),:) = NaN(1,n);
    distances(:,route(i-1)) = NaN(n,1);
    cost = cost + add_cost; % aggiornamento dei costi
end
cost = cost + w(route(n)); % aggiunge ai costi il ritorno all'
    origine

end
```

C distanceMatrix

Funzione per il calcolo delle distanze euclidee:

```
function W = distanceMatrix(x,y)
%INPUTS:
%x: vettore coordinate x dei nodi
%y: vettore coordinate y dei nodi
%OUTPUTS:
%W: matrice delle distanze a coppie
n = length(x);
W = zeros(n,n);
for i = 1:n
```

```
    for j = i+1:n
        W(i,j) = sqrt((x(i)-x(j))^2 + (y(i)-y(j))^2);
        W(j,i) = W(i,j);
    end
end
end
```

D Savings

Funzione che implementa il metodo *Saving*:

```
function [routes, lengths, Costs, occ_cap] = Savings_boost(x,y,
    capacity,collections)
%INPUTS:
%x: vettore contenente coordinate x dei nodi
%y: vettore contenente coordinate y dei nodi
%capacity: scalare per la capacit  di ogni veicolo
%collections: vettore con la domanda dei punti di ritiro

%OUTPUTS:
%routes: matrice che salva su ogni riga una route
%lengths: vettore con il numero di nodi per ogni route
%Costs: vettore con il costo di ogni route
%occ_cap: vettore con la capacit  occupata su ogni route

%%      INIZIALIZZAZIONE
n = length(x);
routes = zeros(n-1,n);
routes(:,1) = ones(n-1,1);
routes(:,2) = [2:n]'; % inseriamo le route del tipo 1-i-1 con i
    = 2:n
occ_cap = collections(routes(:,2)); % inseriamo la domanda per i
     nodi appena inseriti
lengths = 2.*ones(n-1,1); % vettore che nell'entry i ha il
    numero di nodi nella route i
W = distanceMatrix(x,y);

%%      CALCOLO E ORDINAMENTO DEI SAVINGS
savs = NaN(n,n);
for i = 1:n
    for j = 1:n
        savs(i,j) = W(i,1)+W(1,j)-W(i,j); % risparmio dato dall'
    unione di
        % due route data dalla connessione di i e j
```

```
    end
    savs(i,1:i) = zeros(1,i);
end
savs(1,:) = zeros(1,n);
% alcuni elementi sono stati resi nulli per evitare il
   ripetimento dovuto
% alla simmetria del problema

[R,C] = ndgrid(1:size(savs,1),1:size(savs,2));

% ordinamento dei savings
[sort_savs,idx] = sort(savs(:),'descend');

R = R(idx);
C = C(idx);

sort_savs = sort_savs(1:find(sort_savs(:)==0)-1);
R = R(1:length(sort_savs));
C = C(1:length(sort_savs));

% [R, C]    una lista di coppie di indici che  d  la classifica
   dei nodi da
% mergiare per ottenere savings maggiore

%%      MERGING DELLE ROUTE IN ORDINE DI SAVINGS DECRESCENTE
% quelli che seguono sono semplici passaggi di taglia e cuci di
   route:
% - scorriamo la classifica dei savings e abbiamo la coppia (i,j
   )
% - cerchiamo due route che abbiano come capo o coda i nodi i o
   j
% - se non le troviamo andiamo avanti nella classifica,
   altrimenti le mergiamo
% - ci si ferma quando si finisce di scorrere la classifica
k = 1;
while k <= length(sort_savs)
    first = [];
    head1 = true; % flag che ci dice se la route va attaccata
   dalla head o dalla tail;
    head2 = true;
    j = 1;
    while isempty(first) && j<size(routes,1)
        if routes(j,2) == R(k)
            first = j;
        end
```

```
67          j = j+1;
68      end
69
70      % se non ci sono route che iniziano cos  cerchiamo alla
        fine
71      if isempty(first)
72          found = 0;
73          j = 0;
74          while ~found && j<size(routes,1)
75              j = j+1;
76              if routes(j,lengths(j)) == R(k)
77                  found = 1;
78              end
79          end
80          first = j;
81          head1 = false;
82      end
83
84      second = [];
85      j = 1;
86      while isempty(second) && j<size(routes,1)
87          if routes(j,2) == C(k)
88              second = j;
89          end
90          j = j+1;
91      end
92
93      % se non ci sono route che iniziano cos  cerchiamo alla
        fine
94      if isempty(second)
95          found = 0;
96          j = 0;
97          while ~found && j<size(routes,1)
98              j = j+1;
99              if routes(j,lengths(j)) == C(k)
100                 found = 1;
101             end
102         end
103         second = j;
104         head2 = false;
105     end
106
107     if isempty(first) || isempty(second) || first == second
108         disp('Salto la coppia di savings perch  non trovo i
        corrispettivi nodi o sono nella stessa route');
```

```
109     elseif head1 && head2
110         if occ_cap(first) + occ_cap(second) <= capacity
111             merge_1 = routes(first,2:lengths(first));
112             merge_2 = routes(second,2:lengths(second));
113             merge_1 = fliplr(merge_1);
114             routes(first,1:(lengths(first)+lengths(second)-1)) =
     [1 merge_1 merge_2];
115             routes(second, :) = zeros(1,n);
116 
117             occ_cap(first) = occ_cap(first) + occ_cap(second);
118             occ_cap(second) = 0;
119 
120             lengths(first) = lengths(first) + lengths(second) -
    1; %il deposito    ripetuto due volte
121             lengths(second) = 0;
122         end
123     elseif head1 && ~head2
124         if occ_cap(first) + occ_cap(second) <= capacity
125             merge_1 = routes(first,2:lengths(first));
126             merge_2 = routes(second,1:lengths(second));
127             routes(second,1:(lengths(first)+lengths(second)-1))
    = [merge_2 merge_1];
128             routes(first, :) = zeros(1,n);
129 
130             occ_cap(second) = occ_cap(first) + occ_cap(second);
131             occ_cap(first) = 0;
132 
133             lengths(second) = lengths(first) + lengths(second) -
     1; %il deposito    ripetuto due volte
134             lengths(first) = 0;
135         end
136     elseif ~head1 && head2
137         if occ_cap(first) + occ_cap(second) <= capacity
138             merge_1 = routes(first,1:lengths(first));
139             merge_2 = routes(second,2:lengths(second));
140             routes(first,1:(lengths(first)+lengths(second)-1)) =
     [merge_1 merge_2];
141             routes(second, :) = zeros(1,n);
142 
143             occ_cap(first) = occ_cap(first) + occ_cap(second);
144             occ_cap(second) = 0;
145 
146             lengths(first) = lengths(first) + lengths(second) -
    1; %il deposito    ripetuto due volte
147             lengths(second) = 0;
```

```
        end
    elseif ~head1 && ~head2
        if occ_cap(first) + occ_cap(second) <= capacity
            merge_1 = routes(first,1:lengths(first));
            merge_2 = routes(second,2:lengths(second));
            merge_2 = fliplr(merge_2);
            routes(first,1:(lengths(first)+lengths(second)-1)) =
    [merge_1 merge_2];
            routes(second, 1:n) = zeros(1,n);

            occ_cap(first) = occ_cap(first) + occ_cap(second);
            occ_cap(second) = 0;

            lengths(first) = lengths(first) + lengths(second) -
   1; %il deposito    ripetuto due volte
            lengths(second) = 0;
        end
    end
    k=k+1;

    routes = routes(find(routes(:,1)~=0),:);
    occ_cap = occ_cap(find(occ_cap~=0));
    lengths = lengths(find(lengths~=0));
end

%%      OUTPUT

routes = routes(:,1:max(lengths)+1);
Costs = zeros(size(routes,1),1);

CostFunction = @(route) RouteLength(route, W);    % Cost
    Function
for i = 1:length(Costs)
    tmp = routes(i,1:lengths(i));
    Costs(i) = CostFunction(tmp);
end
end
```

E RouteLength

Funzione che calcola il costo complessivo di una route:

```
function L = RouteLength(route, W)
%INPUTS:
```

```
%route: vettore di nodi
%W: matrice delle distanze tra nodi
%OUTPUTS:
%L: costo della route
    sz = length(route);
    route=[route 1]; % aggiungi il ritorno al deposito

    L=0;
    for k=1:sz
        i = route(k);
        j = route(k+1);
        L = L + W(i,j);
    end
end
```

F TabuSearch

Funzione che applica il metodo Tabu ad un insieme di route date allo scopo di diminuire il costo delle stesse:

```
function [new_routes, BestCosts, iters] = TabuSearch(x, y,
    routes, lengths, W, MaxIt)
%INPUTS:
%x: vettore delle coordinate x dei nodi
%y: vettore delle coordinate y dei nodi
%routes: matrice che contiene una route in ogni riga
%lenghts: vettore delle lunghezze di una singola route
%W: matrice delle distanze
%MaxIt: numero di iterazioni massimo della Tabu Search
%OUTPUTS:
%new_routes: matrice che contiene le route aggiornate su ogni
    riga
%BestCosts: matrice che salva i costi di ogni route ad ogni
    iterazione
%iters: vettore che salva le iterazioni necessarie a raggiungere
     il minimo globale per ogni route

% numero di route
m = size(routes,1);

CostFunction = @(route) RouteLength(route, W); % funzione di
    costo della route

```

```
new_routes = zeros(m,max(lengths)+1); % nuova matrice delle
    route risultanti
BestCosts = zeros(m,MaxIt); % matrice che ha sulla colonna j i
    costi di ogni route all'iterazione j
iters = zeros(m,1); % segna il numero di iterazioni necessarie a
     raggiungere il minimo globale per ogni route

for cont = 1:m % effettua scambi intra route quindi possiamo
    considerare le route singolarmente
     xx = x(routes(cont,2:lengths(cont))); % esclude l'origine
    dalla route in quanto non vanno effettuate azioni su questa
     yy = y(routes(cont,2:lengths(cont)));
     route = routes(cont,2:lengths(cont));
     model = CreateModel(xx,yy); % Crea un modello TSP

     ActionList = CreatePermActionList(model.n); % Action List
     nAction = numel(ActionList); % Numero di azioni
     TL = round(nAction); % Tabu Length

     %%      INIZIALIZZAZIONE
     empty_individual.Position = [];
     empty_individual.Cost = [];

     % Crea una soluzione iniziale
     sol = empty_individual;
     sol.Position = route;
     sol.Cost = CostFunction([1 sol.Position]); % nel calcolo dei
      costi si riaggiunge l'origine

     % Inizializza il minimo fino ad ora trovato
     BestSol = sol;
     BestCost = zeros(MaxIt,1); % vettore che tiene conto del
    miglior costo trovato ad ogni iterazione per la route
     TC = zeros(nAction,1); % Tabu Counter per ogni route: timer
    che per
     % ogni azione ti dicono se si pu  utilizzare (TC(i)==0) o
     % bisogna aspettare (TC(i)!=0)

     %%      ITERAZIONI DI TABU SEARCH

     for it = 1:MaxIt

         bestnewsol.Cost = inf;

         for i = 1:nAction
```

```
56            if TC(i) == 0
57                % crea una nuova soluzione usando l'azione i
58                newsol.Position = DoAction(sol.Position,
    ActionList{i});
59                newsol.Cost = CostFunction([1 newsol.Position]);
60                newsol.ActionIndex = i;
61                % se la nuova soluzione    migliore aggiorna la
    miglior
62                % soluzione
63                if newsol.Cost <= bestnewsol.Cost
64                    bestnewsol = newsol;
65                end
66            end
67        end
68
69        sol = bestnewsol;
70
71        % Aggiornamento Tabu List: se per migliorare la
    soluzione    stata
72        % usata l'azione i su questa viene messo un timer TL che
     verr
73        % diminuito di un'unit  ogni iterazione e che impedir
     di riusare
74        % la stessa azione per TL iterazioni
75        for i = 1:nAction
76            if i == bestnewsol.ActionIndex
77                TC(i) = TL;               % Add To Tabu List
78            else
79                TC(i) = max(TC(i)-1,0);   % Reduce Tabu Counter
80            end
81        end
82
83        if sol.Cost <= BestSol.Cost
84            BestSol = sol;
85        end
86
87        BestCost(it) = BestSol.Cost;
88        new_routes(cont,1:lengths(cont)) = [1 BestSol.Position
    (1:end)];
89        % se    stato raggiunto un minimo globale
90        if BestCost(it) == 0
91            break;
92        end
93    end
94
```

```
    %%      OUTPUT DEI RISULTATI

    for i = 1:size(routes,1)
        new_routes(i, lengths(i)+1) = 0;
    end
    BestCosts(cont,1:it) = BestCost(1:it);
    iters(cont) = it;
end
end
```

G CreatePermActionList

Funzione che crea la lista di azioni applicabili ad una route di n nodi:

```
function ActionList=CreatePermActionList(n)
%INPUTS:
%n: numero di nodi della route
%OUTPUTS
%ActionList: lista di azioni possibili
    nSwap = n*(n-1)/2; % combinazioni
    nReversion = n*(n-1)/2; % combinazioni
    nInsertion = n^2; % permutazioni
    nAction = nSwap+nReversion+nInsertion; % numero massimo
   possibile di azioni

    ActionList = cell(nAction,1); % creiamo un cell array di
   nAction matrici
    c=0;

    % SWAP: scambia due nodi i e j di una route
    for i=1:n-1
        for j=i+1:n
            c=c+1;
            ActionList{c}=[1 i j];
            %il primo elemento dell'array    il codice dell'
   operazione mentre secondo e terzo i nodi al quale si applica
        end
    end

    % REVERSION: prende la sequenza di nodi tra i e j e la
   inverte
    for i=1:n-1
        for j=i+1:n
            if abs(i-j)>2
```

```
                c=c+1;
                ActionList{c}=[2 i j];
            end
        end
    end

    % INSERTION: inserisce il nodo i successivamente al nodo j o
    viceversa
    for i=1:n
        for j=1:n
            if abs(i-j)>1
                c=c+1;
                ActionList{c}=[3 i j];
            end
        end
    end

    ActionList=ActionList(1:c);
    % c conta quante azioni sono state effettuate sul totale

end
```

H Modello TSP

Funzione che crea il modello TSP relativo alla singola route:

```
function model=CreateModel(x,y)
%INPUTS:
%x: vettore delle coordinate x dei nodi
%y: vettore delle coordinate y dei nodi
%OUTPUTS
%model: classe dei parametri del modello
n = length(x);
    d = distanceMatrix(x,y);

    xmin = min(x);
    xmax = max(x);

    ymin = min(y);
    ymax = max(y);

    model.n=n;
    model.x=x;
    model.y=y;
```

```
    model.d=d;
    model.xmin=xmin;
    model.xmax=xmax;
    model.ymin=ymin;
    model.ymax=ymax;

end
```

I DoAction

Funzione che in base al primo indice del vettore corrispondente ad un elemento della lista di azioni decide quale azione eseguire:

```
function q = DoAction(p,a)
%INPUTS:
%p: route
%a: matrice elemento di una lista
%OUTPUTS
%q: route modificata
switch a(1)
        case 1
            q=DoSwap(p,a(2),a(3));

        case 2
            q=DoReversion(p,a(2),a(3));

        case 3
            q=DoInsertion(p,a(2),a(3));
    end

end
```

J DoSwap

Funzione che performa l'azione di scambio tra due nodi:

```
function q=DoSwap(p,i1,i2)
%INPUTS:
%i1: primo nodo della coppia
%i2: secondo nodo della coppia
%p: route iniziale
%OUTPUTS
```

```
q: route aggiornata
    q=p;
    q([i1 i2])=p([i2 i1]);
end
```

K DoReversion

Funzione che performa l'azione di "rovesciamento" di un tratto di route:

```
function q=DoReversion(p,i1,i2)
%INPUTS:
%i1: primo nodo della coppia
%i2: secondo nodo della coppia
%p: route iniziale
%OUTPUTS
%q: route aggiornata
   q=p;
    if i1<i2
        q(i1:i2)=p(i2:-1:i1);
    else
        q(i1:-1:i2)=p(i2:i1);
    end
end
```

L DoInsertion

Funzione che performa l'azione di inserimento di un nodo dopo l'altro:

```
function q = DoInsertion(p,i1,i2)
%INPUTS:
%i1: primo nodo della coppia
%i2: secondo nodo della coppia
%p: route iniziale
%OUTPUTS
%q: route aggiornata
   q=p;
    if i1<i2
        q(i1:i2)=p(i2:-1:i1);
    else
        q(i1:-1:i2)=p(i2:i1);
    end
end
```

M NN_run

Main per simulare su algoritmi di tipo sweep e Nearest Neighbor e plottaggio di grafici e tabelle di confronto con la successiva applicazione della Tabu Search:

```
%%      INIZIALIZZAZIONE
n = 100; % numero di nodi
MaxIt = 50; % numero massimo di iterazioni della Tabu Search (TS)
col_min = 1; % domanda minima dei nodi
col_max = 35; % domanda massima dei nodi
capacity = 300; % capacit  dei furgoncini
range = 40; % ampiezza dello spazio dei nodi

%%      GENERAZIONE DELLE VARIABILI RANDOMICHE
rng(12)
collections = (col_max-col_min).*rand(n,1)+col_min.*ones(n,1); % vettore delle domande
x = range.*rand(n,1) - (range/3).*ones(n,1); % coordinate dei nodi
y = range.*rand(n,1) - (range/3).*ones(n,1);
x(1) = 0; %  inseriamo come primo nodo l'origine, che funge da deposito
y(1) = 0;

costs = zeros(n,1); % vettore dei costi per ogni route
occ_cap = zeros(n,1); % vettore della capacit  occupata per ogni route
routes = zeros(n,n); % matrice che ha sulle righe le route

%%      SWEEP METHOD PER CLUSTERIZZARE
tic
[clusters, lengths] = SweepClustering_cap(x(2:end), y(2:end), capacity, collections(2:end), [x(1); y(1)]);

m = size(clusters,1)-1; % numero di clusters
% segue un passaggio utile a riscalare gli indici dei nodi clusterizzati,
% in quanto non si considera l'origine, che ora va riinclusa
for i = 1:m
    clusters(i,1:lengths(i)) = clusters(i,1:lengths(i)) + ones(1,lengths(i));
end

%%      MYOPIC TSP PER CLUSTER (NN heuristic)
```

```
33
34 for j = 1:m
35     [route, cost] = NNheuristic([0; x(clusters(j,1:lengths(j)))
       ], [0; y(clusters(j,1:lengths(j)))]);
36     routes(j,1:(lengths(j)+2)) = route';
37     costs(j) = cost;
38     k = 2;
39     % segue un passaggio che traduce gli indici ordinati del
       cluster in
40     % indici ordinati dell'intero grafo
41     while k <= find(routes(j,2:end)==0)
42         routes(j,k) = clusters(j,routes(j,k)-1);
43         k = k+1;
44     end
45 end
46 % tronchiamo le matrici e i vettori per quanto utilizzati
47 costs = costs(1:m);
48 routes = routes(1:m,1:max(lengths)+2);
49 timesweep = toc;
50 routes(:,2) = [];
51
52 %%      TABU SEARCH
53 W = distanceMatrix(x,y);
54
55 tic
56 % applichiamo la TS alla soluzione trovata tramite sweep + NN
       heuristic
57 [new_routes, BestCosts, iters] = TabuSearch(x,y,routes,lengths,W
       ,MaxIt);
58 timetabu = toc;
59
60 %%      DATA VISUALIZATION
61
62 figure;
63 subplot(1,2,1);
64 PlotSolution(x,y,routes);
65 title('Sweep + NN heuristic');
66 subplot(1,2,2);
67 PlotSolution(x,y,new_routes);
68 title('Sweep + NN heuristic + Tabu Search');
69
70 figure;
71 TotCost = zeros(MaxIt,1);
72 for i = 1:MaxIt
73     TotCost(i) = sum(BestCosts(:,i));
```

```
end

plot([sum(costs) TotCost'],'LineWidth',2,'Color','blue');
title('Total Cost - Sweep + NN + TS')
xlabel('Iteration');
ylabel('Best Cost');
xlim([0 MaxIt]);
grid on;

colors = ["#0072BD";"#D95319";"#EDB120";"#7E2F8E";"#77AC30";"#4
    DBEEE";...
     "#A2142F";"#0082BD";"#D75319";"#EDB320";"#5E2F8E";"#27AC30
    ";"#3DBEEE";"#A2342F"];

figure;
for i = 1:size(routes,1)
    subplot(fix(size(routes,1)/4)+1, 4, i);
    plot([costs(i) BestCosts(i,1:iters(i))],'LineWidth',2,'Color
    ',colors(i));
    title('Route n. ',num2str(i))
    xlabel('Iteration');
    xlim([0 MaxIt]);
    ylabel('Best Cost');
    grid on;
end

% Controllo sui costi delle singole route
disp('Costi di Sweep e TS - percentuale di diminuzione')
disp([costs BestCosts(:,end) ((ones(length(costs),1)-BestCosts
    (:,end)./costs).*(100.*ones(length(costs),1)))])

% Controllo sulle capacit  occupata per route
for cont = 1:m
    for j = 2: lengths(cont)
    occ_cap(cont) = occ_cap(cont) + collections(routes(cont,j));
    end
end

disp ('Livello di capacit\'a %')
for i = 1:size(routes,1)
    disp((occ_cap(i)/capacity)*100)
end

% Controllo sui tempi computazionali
```

```
mex1 = ['Algoritmo sweep + NN ha impiegato ' num2str(timesweep)
    ' s'];
disp(mex1)

mex2 = ['Algoritmo sweep + NN + TS ha impiegato ' num2str(
    timesweep+timetabu) ' s'];
disp(mex2)
```

N Saving_run

Main per simulare su algoritmi di tipo saving e plottaggio di grafici e tabelle di confronto con la successiva applicazione della Tabu Search:

```
clear all
close all
clc
format short

%%      INIZIALIZZAZIONE
n = 100; % numero di nodi
MaxIt = 100; % numero massimo di iterazioni della Tabu Search (
    TS)
col_min = 1; % domanda minima dei nodi
col_max = 35; % domanda massima dei nodi
capacity = 300; % capacit  dei furgoncini
range = 40; % ampiezza dello spazio dei nodi

%%      GENERAZIONE DELLE VARIABILI RANDOMICHE
rng(12)
collections = (col_max-col_min).*rand(n,1)+col_min.*ones(n,1); %
     vettore delle domande
x = range.*rand(n,1) - (range/3).*ones(n,1); % coordinate dei
    nodi
y = range.*rand(n,1) - (range/3).*ones(n,1);
x(1) = 0; %  inseriamo come primo nodo l'origine, che funge da
    deposito
y(1) = 0;

%% CREAZIONE DELLA SOLUZIONE COSTRUTTIVA TRAMITE SAVINGS
    CRITERION
tic
[routes, lengths, costs, occ_cap] = Savings_boost(x,y,capacity,
    collections);
timesavs = toc;
```

```
26
27 %% CREAZIONE DELLA SOLUZIONE ITERATIVA TRAMITE TABU SEARCH
28 tic
29 W = distanceMatrix(x,y);
30 [new_routes, BestCosts, iters] = TabuSearch(x,y,routes,lengths,W
       ,MaxIt);
31 timetabu = toc;
32
33 %%      DATA VISUALIZATION
34 figure;
35 subplot(1,2,1);
36 PlotSolution(x,y,routes);
37 title('Savings');
38 subplot(1,2,2);
39 PlotSolution(x,y,new_routes);
40 title('Savings + Tabu Search');
41
42 figure;
43 TotCost = sum(BestCosts);
44 plot(TotCost,'LineWidth',2,'Color','blue');
45 title('Total Cost')
46 xlabel('Iteration');
47 ylabel('Best Cost');
48 grid on;
49
50 colors = ["#0072BD";"#D95319";"#EDB120";"#7E2F8E";"#77AC30";"#4
       DBEEE";...
51     "#A2142F";"#0082BD";"#D75319";"#EDB320";"#5E2F8E";"#27AC30
       ";"#3DBEEE";"#A2342F"];
52 figure;
53 for i = 1:size(routes,1)
54     subplot(fix(size(routes,1)/4)+1, 4, i);
55     plot([costs(i) BestCosts(i,:)],'LineWidth',2,'Color',colors(
       i));
56     title('Route n. ',num2str(i))
57     xlabel('Iteration');
58     ylabel('Best Cost');
59     grid on;
60 end
61
62 % Controllo sui costi delle singole route
63 disp('Costi di Savings e TS - percentuale di diminuzione')
64 disp([costs BestCosts(:,end) ((ones(length(costs),1)-BestCosts
       (:,end)./costs).*(100.*ones(length(costs),1)))])
65
```

```
disp ('Livello di capacit   %')
for i = 1:size(routes,1)
    disp((occ_cap(i)/capacity)*100)
end

% Controllo sui tempi computazionali
mex1 = ['Algoritmo savings ha impiegato ' num2str(timesavs) ' s'
    ];
disp(mex1)

mex2 = ['Algoritmo savings + TS ha impiegato ' num2str(timesavs+
    timetabu) ' s'];
disp(mex2)
```

O PlotSolution

Funzione per disegnare i cluster e i percorsi:

```
function PlotSolution(x,y,routes)
%INPUTS:
%x: vettore delle coordinate x dei nodi
%y: vettore delle coordinate y dei nodi
%routes: matrice in cui ogni riga corrisponde a una route
%OUTPUTS:
%Disegno delle route
n = length(x);
m = size(routes,1);

colors = ["#0072BD";"#D95319";"#EDB120";"#7E2F8E";"#77AC30";"#4
    DBEEE";"#A2142F";"#0082BD";"#D75319";"#EDB320";"#5E2F8E";"#27
    AC30";"#3DBEEE";"#A2342F"];
%colors = ["red","green","blue","cyan","yellow","magenta","black
    "]';
hold on
scatter(x,y,30,'MarkerEdgeColor','y','MarkerFaceColor','black')
a = num2str([1:n]');
b = cellstr(a);
hold on
text(x+0.1,y+0.1,b)

for j=1:m  % vai veicolo per veicolo
    k = 0; % indice del nodo da prendere in esame
    while routes(j,k+1) %finch\'e la route non finisce
        k = k+1; %passa al nodo successivo
```

```
        if routes(j,k+1)~=0
            xx = [x(routes(j,k)) x(routes(j,k+1))];
            yy = [y(routes(j,k)) y(routes(j,k+1))];
            hold on
            line(xx,yy,'Color',colors(j),'LineWidth',2)
        end
    end
    % plotta il ritorno al deposito
    xx = [x(routes(j,k)) x(routes(j,1))];
    yy = [y(routes(j,k)) y(routes(j,1))];
    hold on
    line(xx,yy,'Color',colors(j),'LineWidth',2)
end
end
```