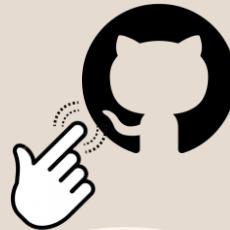




# Car pooling ibrido

Progettino n.35



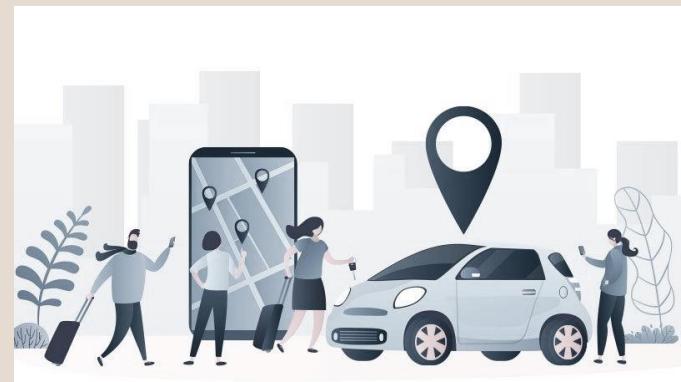
Full project on GitHub!



# Problema:

34. Car Pooling (2 persone): Sono noti i luoghi di origine delle persone che si recano quotidianamente al Polo Scientifico Tecnologico UniFe a inizio giornata (stessa ora di arrivo). Sono noti quelli disponibili ad operare come autisti in un servizio di Car Pooling. Tutti gli altri sono interessati a viaggiare come passeggeri e prendono la propria macchina solo se nessuno li serve: in tal caso non vogliono nessun altro a bordo. Ogni auto porta fino a 5 persone (conducente incluso). Si determini la soluzione che minimizza il numero di km percorsi dalle auto.

35. Car Pooling Ibrido (3 persone) Nel problema precedente, si consideri la variante in cui gli utenti possono effettuare una prima tratta su una linea di trasporto pubblico ed essere prelevati a una fermata. In questa variante del problema ogni utente passeggero ha più punti di prelievo possibili, la propria abitazione e le fermate del trasporto pubblico delle linee che hanno fermate vicino alla propria abitazione.



# Dati:

- Sono state generate in modo randomico le coordinate delle case delle persone, che sono i punti di partenza.
- Sono state generate in modo randomico le coordinate delle fermate delle linee di trasporto pubblico.
- Gli utenti hanno tutti la stessa destinazione che è l'Università.
- Gli utenti si dividono in:
  - autisti → possono prendere la loro macchina e prelevare altre persone.
  - passeggeri → possono prendere la loro macchina ma non sono disposti a portare altre persone.
- I passeggeri possono prendere un mezzo di trasporto e salire solo da una fermata vicina alla propria abitazione.
- Gli utenti che prendono un mezzo di trasporto verranno prelevati in una fermata d'arrivo da un autista.
- Le macchine hanno una capienza limitata a 5 persone.
- L'obiettivo è di minimizzare i km percorsi dalle auto.



Modelliamo il problema



# Modello Matematico

$G=(N,E)$  :grafo connesso non orientato

$E$  è l'insieme degli archi

$N$  è l'insieme dei nodi (case e fermate degli autobus)

$N$  è diviso in:

- $A \rightarrow$  sono i nodi disponibili a dare un passaggio
- $P \rightarrow$  nodi che richiedono un passaggio da casa
- $F \rightarrow$  nodi fermata

Booleani:

- $x_{ap}:\{1 \text{ se } a \in A \text{ dà passaggio a } p \in P, 0 \text{ altrimenti}\}$
- $x_{pc}:\{1 \text{ se } p \in P \text{ si fa venire a prendere a casa, 0 altrimenti}\}$
- $x_{pf}:\{1 \text{ se } p \in P \text{ si fa venire a prendere alla fermata } f \in F\}$
- $x_{ps}:\{1 \text{ se } p \in P \text{ va da solo}\}$



Vincoli:

$$\sum x_{an} \leq 4 \quad \forall a \in A, \quad \forall n \in P$$

$$x_{pc} + x_{pf} + x_{ps} = 1 \quad \forall p \in P$$

Obiettivo:

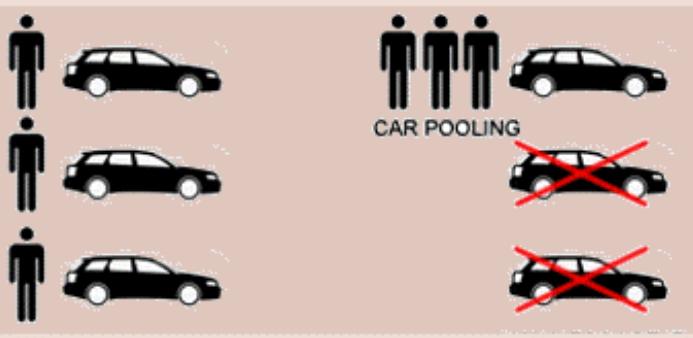
$c(i,j)$ : costo dell'arco che collega nodo  $i$  al nodo  $j$  in km

$R$  insieme delle routes

$$r' = (N', E')$$

$$c_r = \sum c(i,j) \quad \forall (i,j) \in E'$$

$$\text{Min} \sum_{i=1}^{\dim(R)} c_r^i$$





## Realizzazione del problema

---



# NOD○:

Utenti

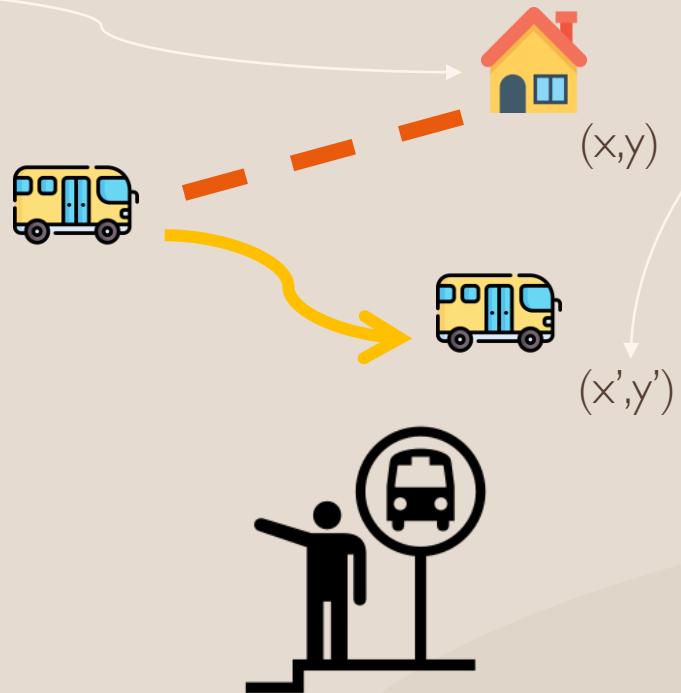
- coordinate x,y
- capacità=0 Unife
- capacità=5 autista
- capacità=1 passeggero
- identificativo numerico



class Nodo:

```
def __init__(self, x, y, c=0, i=0):  
    self.x=x  
    self.y=y  
    self.c=c  
    self.i=i
```

Utenti che prendono l'autobus



Fermate dei mezzi pubblici

- coordinate x,y
- linea del mezzo pubblico



class Fermata:

```
def __init__(self, x, y, l):  
    self.x=x  
    self.y=y  
    self.l=l
```



# GENERARE I NODI :



Il nodo UniFe ha coordinate fisse e capacità=0 → unife = nodo.Nodo(321, 765)

Le coordinate x e y di ogni nodo vanno da 1 a 1000



```
def generator():
    x= randint(1, 1000)
    y= randint(1, 1000)
    n=nodo.Nodo(x, y)
    prob = randint(1, 10)
    if (prob < 8):
        n.set_c(1)
    else:
        n.set_c(5)
    return n
```

C'è il 70% di probabilità che un nodo sia passeggero ( $c=1$ ) ed il 30% che sia un autista ( $c=5$ )



# GENERARE LE FERMATE :

Le coordinate x e y di ogni fermata vanno da 1 a 1000



```
def generator():
    x= randint(1, 1000)
    y= randint(1, 1000)
    l=randint(1, 6)
    f=fermata.Fermata(x, y, l)
    return f
```



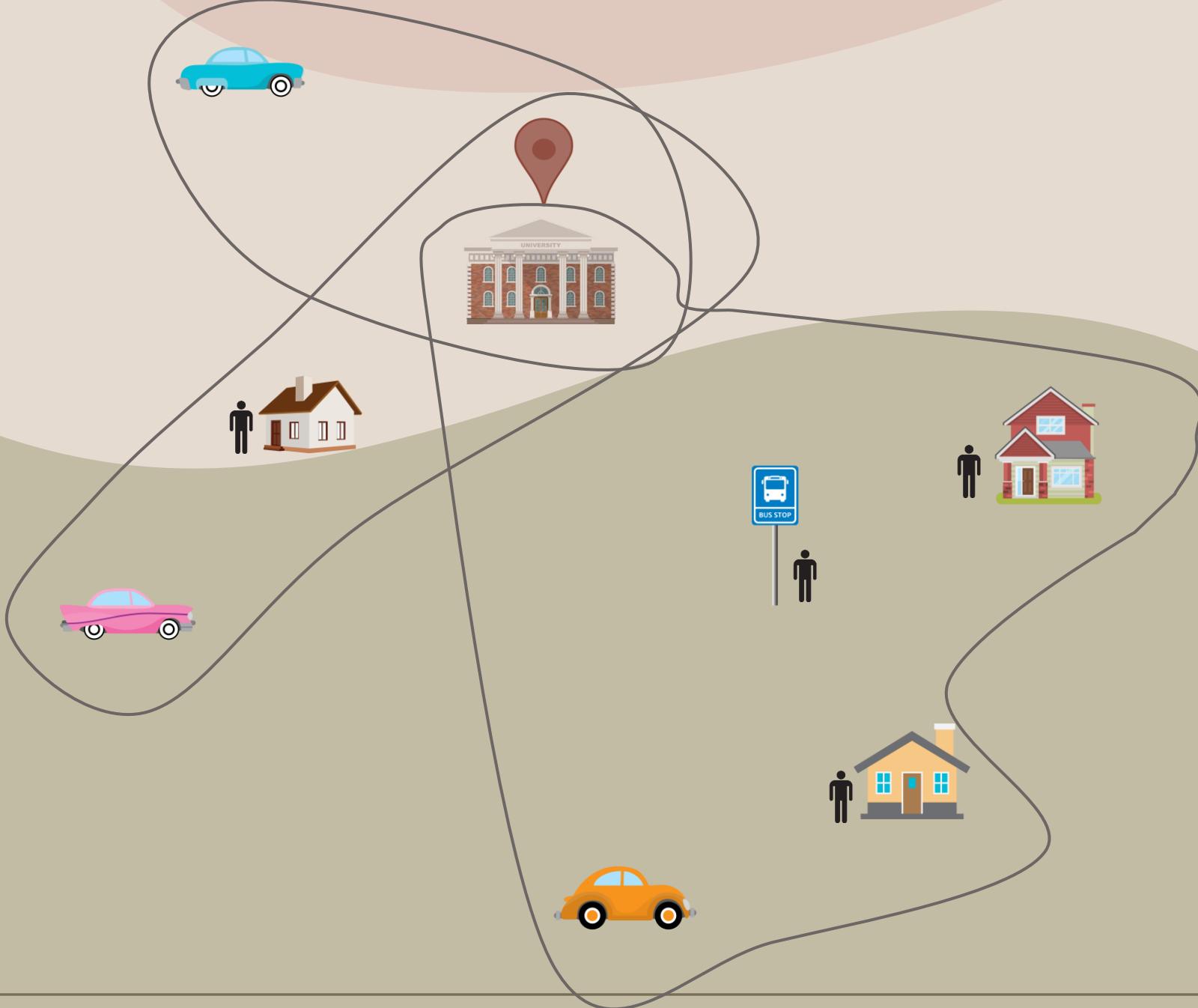
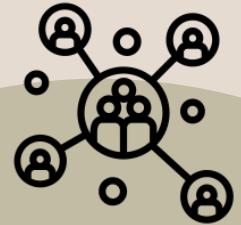
Ci sono 6 linee di mezzi pubblici e ogni fermata è assegnata ad una di queste in modo randomico



# Cluster

```
class Cluster:
```

```
    def __init__(self):  
        self.nodi=[ ]
```



# Cluster first, route second

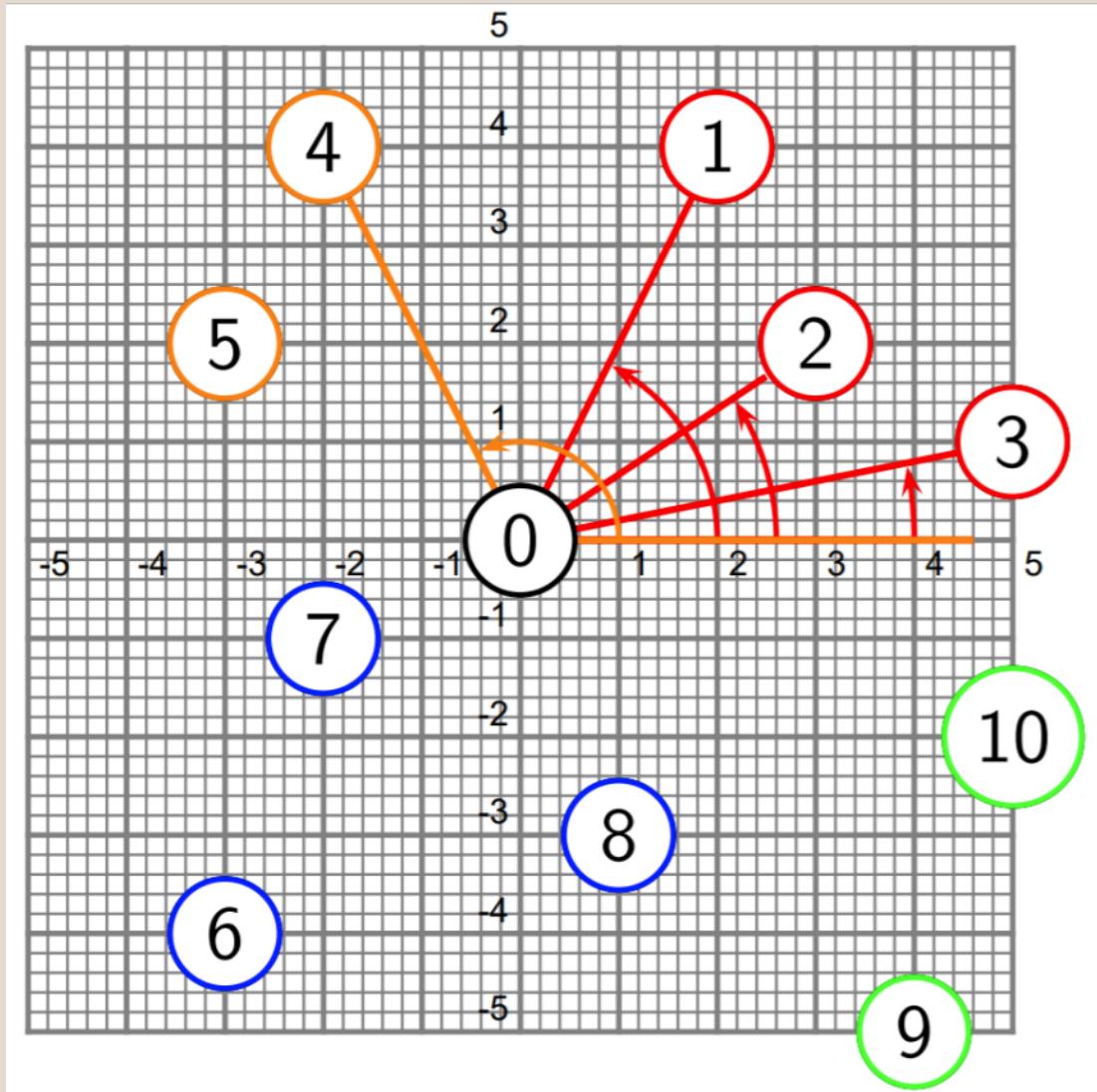


Molti approcci euristici risolvono i sottoproblemi di **assegnamento** e **sequenziamento** in un determinato ordine, dando origine agli approcci «**route** first, **cluster** second» oppure «**cluster** first, **route** second»

Ma come suddividere i nodi nei cluster?

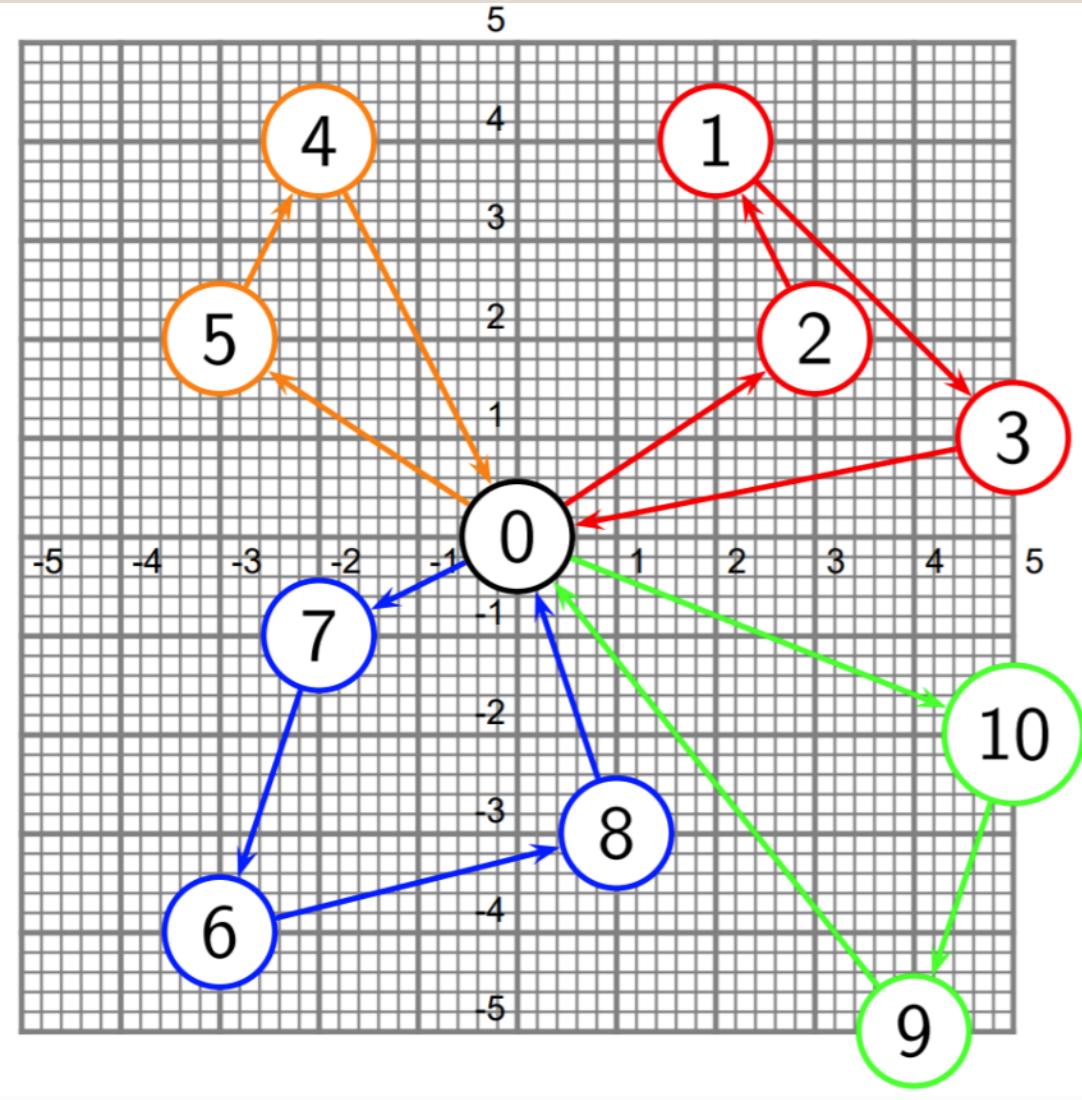


# Cluster first, route second



Sweep algorithm

# Cluster first, route second



Nearest neighbor

# Cluster first, route second

for nodo in lista\_nodi:

```
if ((nodo.get_c() ==1) and (c<4) and (nodo.get_x() != starter_node.get_x()) and (nodo.get_y() !=  
starter_node.get_y())):  
    if(s_x <= nodo.get_x() and nodo.get_x() <= u_x and s_y <= nodo.get_y() and nodo.get_y() <= u_y or \  
    s_x >= nodo.get_x() and nodo.get_x() >= u_x and s_y <= nodo.get_y() and nodo.get_y() <= u_y or \  
    s_x >= nodo.get_x() and nodo.get_x() >= u_x and s_y >= nodo.get_y() and nodo.get_y() >= u_y or \  
    s_x <= nodo.get_x() and nodo.get_x() <= u_x and s_y >= nodo.get_y() and nodo.get_y() >= u_y):  
        cluster.aggiungi_nodo(nodo)  
        ut.rimuovi(lista_nodi, nodo)  
        c=c+1
```

if(c<4):

for fermata in lista\_fermate:

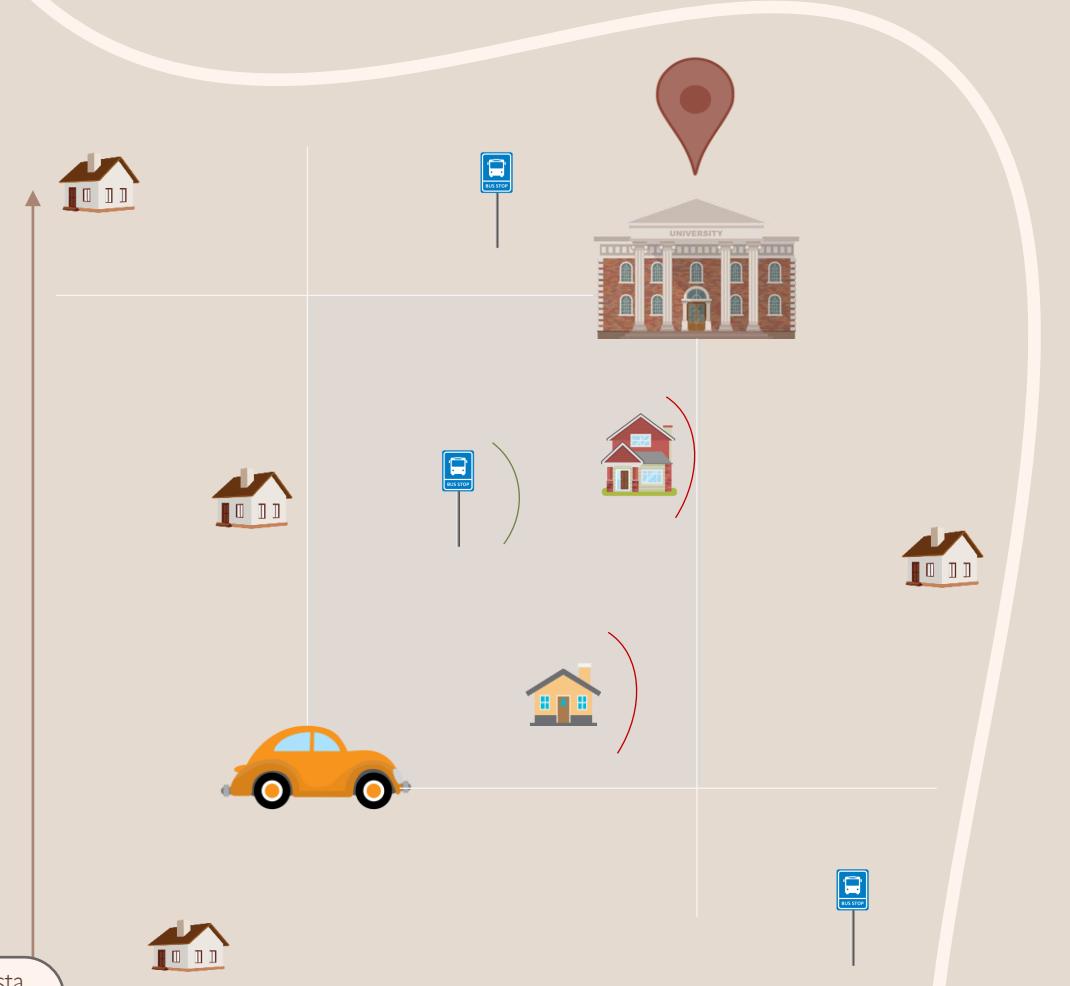
```
if(c<4):  
    if(s_x <= fermata.get_x() and fermata.get_x() <= u_x and s_y <= fermata.get_y() and fermata.get_y() <= u_y or \  
    s_x >= fermata.get_x() and fermata.get_x() >= u_x and s_y <= fermata.get_y() and fermata.get_y() <= u_y or \  
    s_x >= fermata.get_x() and fermata.get_x() >= u_x and s_y >= fermata.get_y() and fermata.get_y() >= u_y or \  
    s_x <= fermata.get_x() and fermata.get_x() <= u_x and s_y >= fermata.get_y() and fermata.get_y() >= u_y):
```

for fermata\_collegata in lista\_fermate:

```
if((fermata_collegata.get_l()==fermata.get_l()) and (c<4)):  
    for nodo_p in lista_nodi:  
        if ((nodo_p.get_c() ==1) and (ut.distanza(nodo_p, fermata_collegata)<=50)and (c<4) and\  
        (ut.distanza(fermata, fermata_collegata)<ut.distanza(nodo_p, unife))):  
            nodo_p.set_x(fermata.get_x())  
            nodo_p.set_y(fermata.get_y())  
            cluster.aggiungi_nodo(nodo_p)  
            ut.rimuovi(lista_nodi, nodo_p)  
            c=c+1
```

Dato un autista che ha ancora spazio in macchina, cerco i passeggeri che hanno ordinata e ascissa compresa tra quella dell'autista e quella di UniFe. e li aggiungo al suo cluster.

Se c'è ancora spazio in macchina cerco le fermate comprese tra l'autista e UniFe. Se c'è un passeggero non più lontano di 50m da una fermata collegata a quella tra l'autista e UniFe e la distanza tra la fermata collegata e quella in cui passerebbe l'autista è minore della distanza tra il passeggero e UniFe, faccio spostare qui il passeggero aggiornando le sue coordinate con quelle della fermata. Aggiungo poi questo passeggero alla macchina dell'autista.



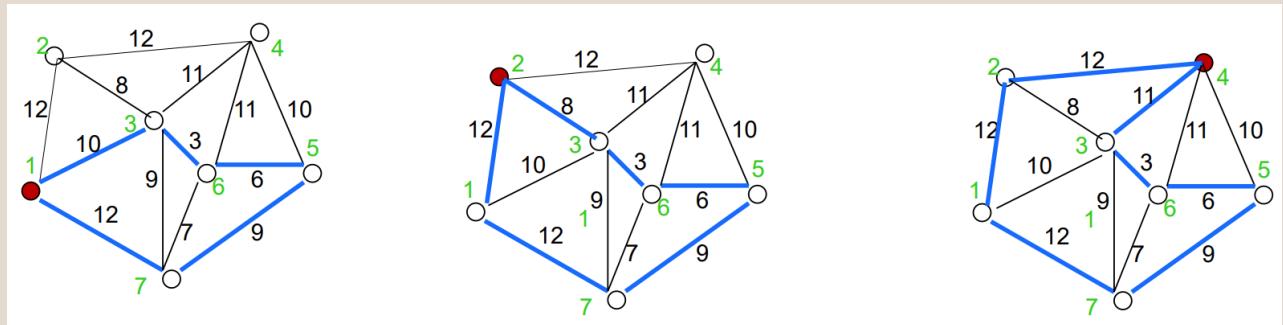
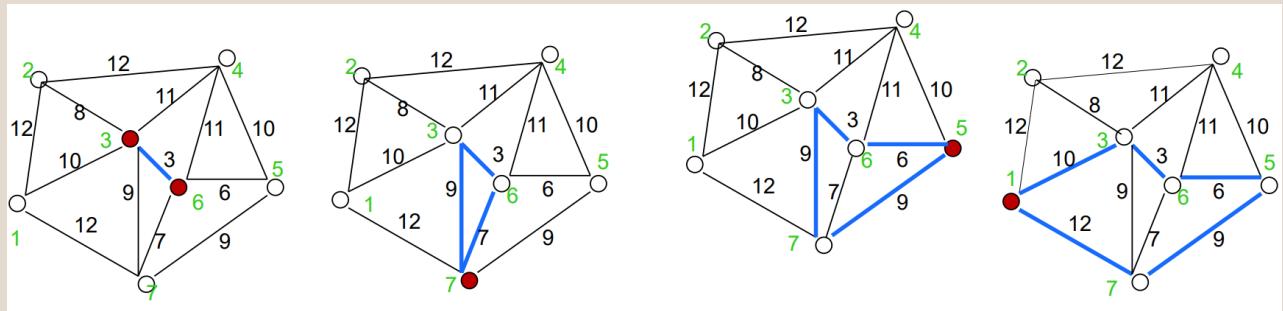
# Route

```
class Route(cl.Cluster):
```

```
    def __init__(self):  
        super().__init__()  
        self.km=0
```



# Cheapest insertion



- Parte dal circuito  $C=(i,j,i)$  di costo minimo (essendo  $(ij)$  l'arco di costo minimo)
- inserisce ad ogni passo un nuovo nodo  $k$  scegliendo il detour di costo minimo rispetto alla soluzione corrente.



# Cheapest insertion

```

def routing(cluster):
    route=rt.Route()
    if len(cluster.get_nodi())==2:
        route.aggiungi_nodo(cluster.get_nodi()[1])
        route.aggiungi_nodo(cluster.get_nodi()[0])
        route.set_km()

    else:
        for i in range(0, len(cluster.get_nodi())):
            if (cluster.get_nodi()[i].get_c()==5):
                route.aggiungi_nodo(cluster.get_nodi()[i])
                cluster.rimuovi_nodo(cluster.get_nodi()[i])
                break

        for i in range(0, len(cluster.get_nodi())):
            if (cluster.get_nodi()[i].get_c()==0):
                route.aggiungi_nodo(cluster.get_nodi()[i])
                cluster.rimuovi_nodo(cluster.get_nodi()[i])
                break

    while(len(cluster.get_nodi())!=0):
        routes = []
        for nodo in cluster.get_nodi():
            for i in range(1, len(route.get_nodi())):
                routes.append(rt.gen_route(route, nodo, i))
        for el in routes:
            el.set_km()

        distanza= routes[0].get_km()
        id = 0
        for i in range(1, len(routes)):
            if(routes[i].get_km() < distanza):
                distanza=routes[i].get_km()
                id=i
        route.set_nodi(routes[id].get_nodi())
        route.set_km2(routes[id].get_km())
        for i in range(0, len(cluster.get_nodi())):
            if cluster.get_nodi()[i] in route.get_nodi():
                cluster.rimuovi_nodo(cluster.get_nodi()[i])
                break

    return route

```

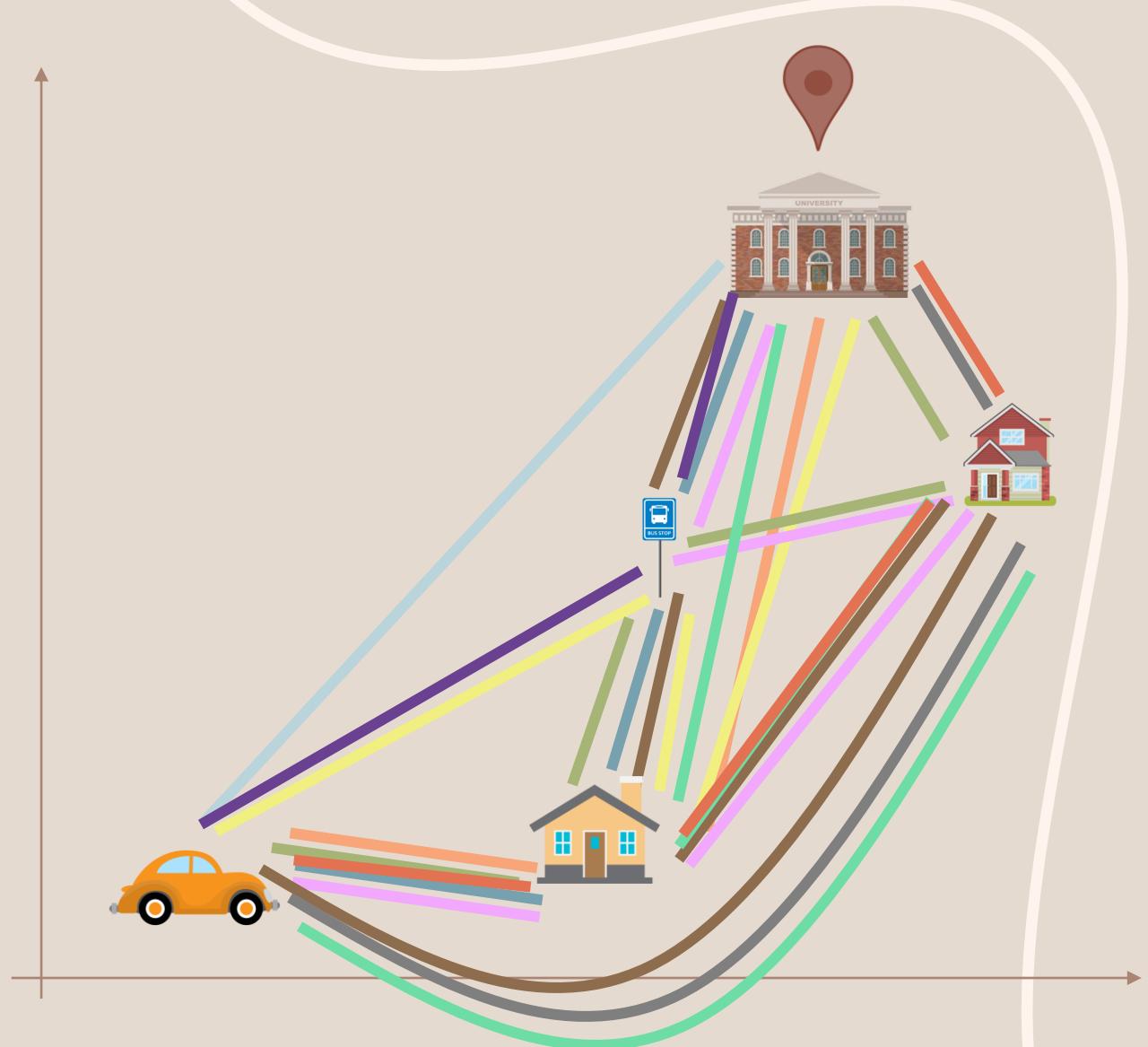
Se si ha un cluster con un utente che va da solo a UniFe si crea una route con solo l'utente e UniFe

Altrimenti ciclo il cluster per trovare l'autista e inserirlo in prima posizione nella route.

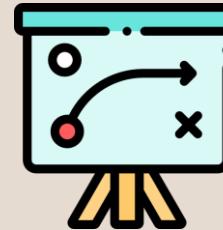
Successivamente ciclo il cluster per trovare UniFe e inserirlo in seconda posizione nella route.

Uno alla volta aggiungo ciascun nodo passeggero del cluster nella route nelle possibili posizioni tra l'autista e UniFe

Tra l'insieme delle possibili route generate per ogni passeggero mantengo solo quella che mi minimizza i km percorsi.



# Visualizziamo i risultati



```
import matplotlib.pyplot as plt
```

```
def grafico(lista_nodi, lista_fermate, lista_route, nomefile):
```

```
    plt.figure(figsize=(30, 30))  
    plt.grid()  
    plt.xticks(range(0, 1001, 50))  
    plt.yticks(range(0, 1001, 50))
```

```
    for el in lista_nodi:
```

```
        plt.plot(el.get_x(), el.get_y(), color='pink', marker='o')  
        plt.annotate(el.get_id(), (el.get_x(), el.get_y()), textcoords="offset points", xytext=(0,10), ha='center')
```

Impostiamo le dimensioni del grafico  
in uscita e la visualizzazione a griglia

In rosa grafichiamo le case di  
coloro che si spostano tramite  
mezzi pubblici

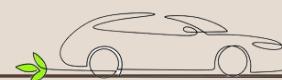
# Visualizziamo i risultati



```
for el in lista_route:  
    lista=el.get_nodi()  
    lista_x=[]  
    lista_y=[]  
    lista_i=[]  
    for el2 in lista:  
        lista_x.append(el2.get_x())  
        lista_y.append(el2.get_y())  
        lista_i.append(el2.get_id())  
    plt.plot(lista_x, lista_y, color='blue', marker='o', linewidth=0.5)  
    for i, label in enumerate(lista_i):  
        plt.annotate(label, (lista_x[i], lista_y[i]), textcoords="offset points", xytext=(0,10), ha='center')  
  
for el in lista_fermate:  
    plt.plot(el.get_x(), el.get_y(), color='orange', marker='o')  
    plt.annotate(el.get_l(), (el.get_x(), el.get_y()), textcoords="offset points", xytext=(10,0), ha='center')  
    plt.plot(321, 765, color='red', marker='o')  
  
plt.savefig(nomefile)
```

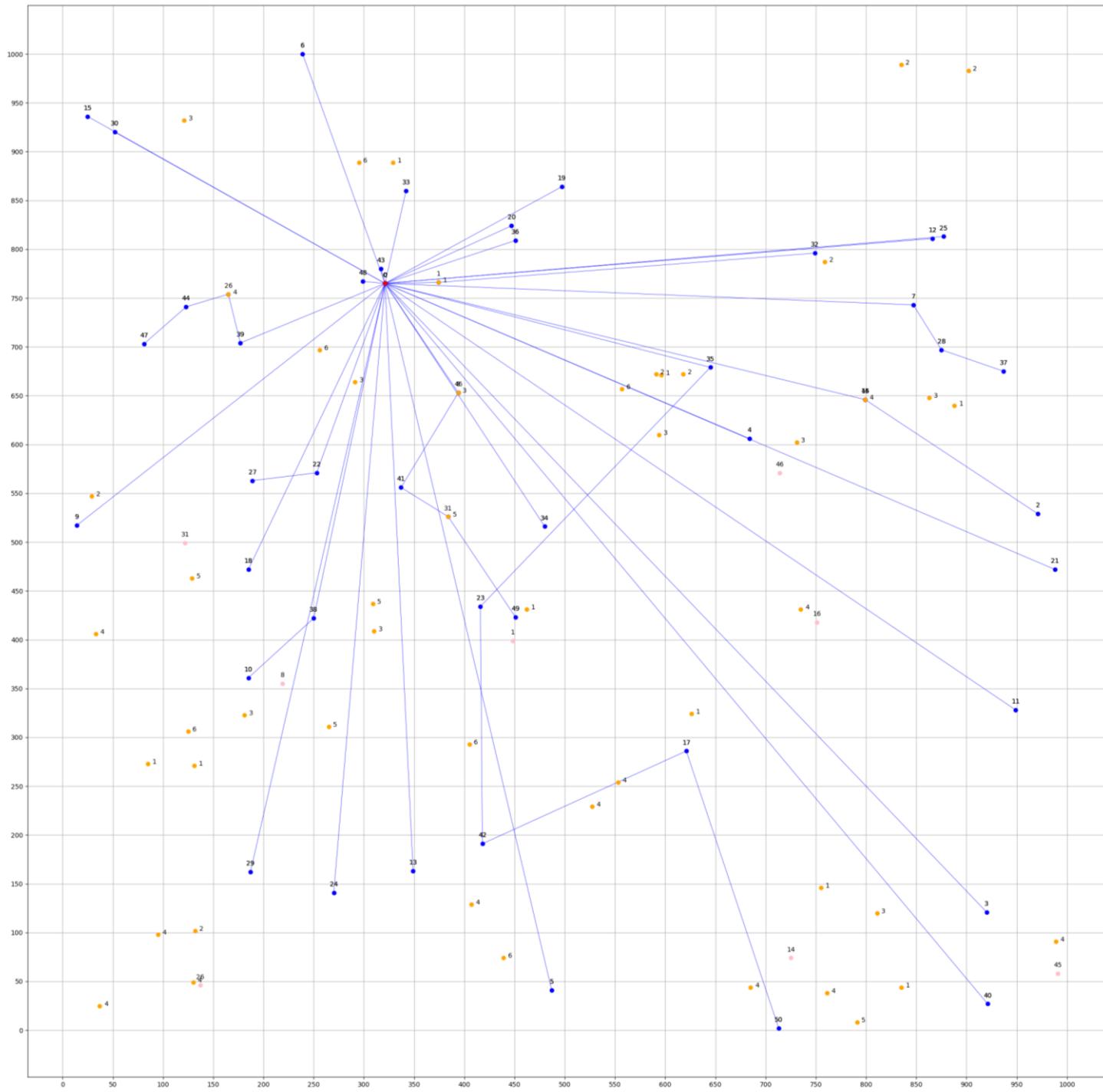
In blu grafichiamo le case di coloro che prendono la macchina o sono prelevati da casa

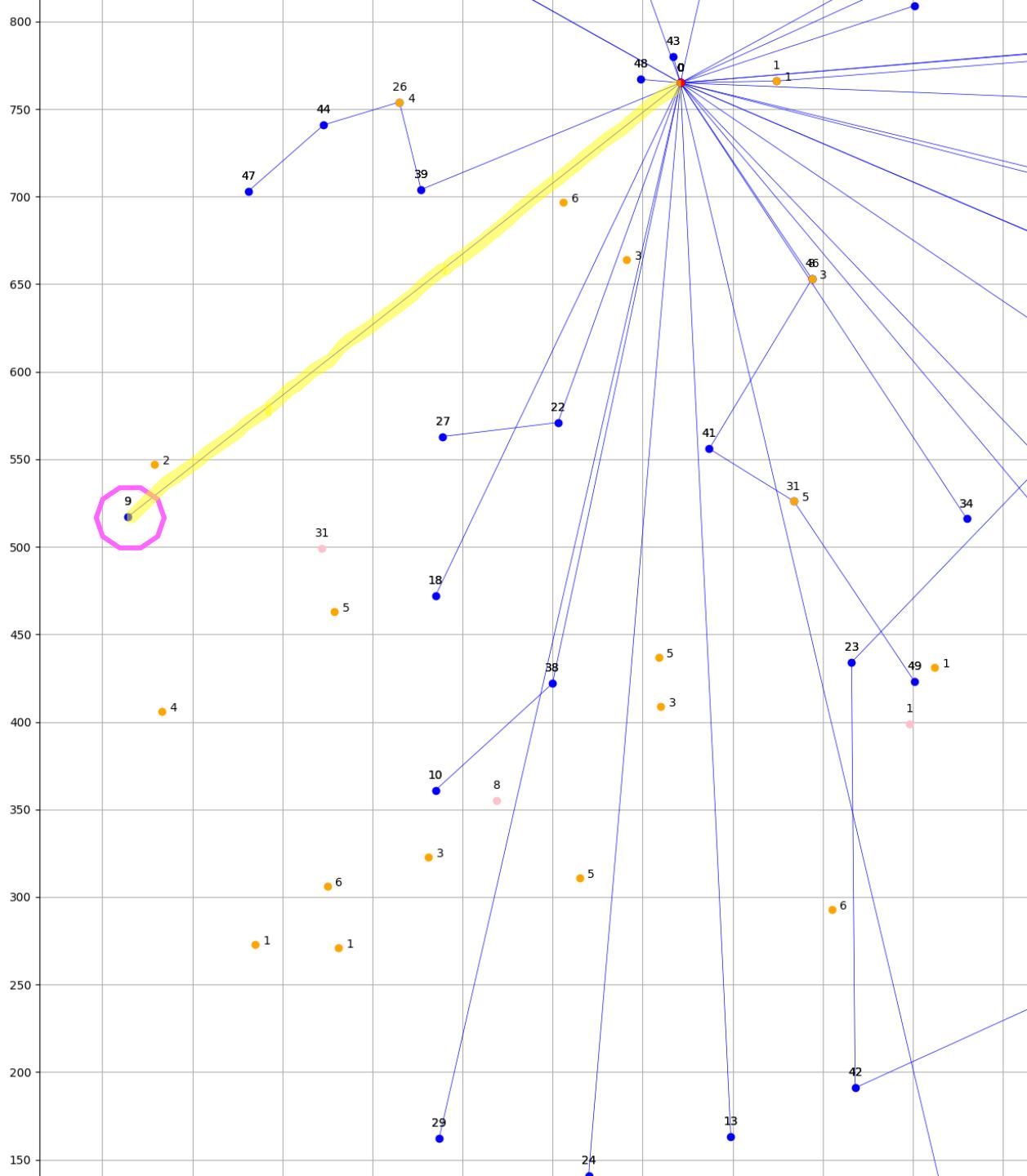
In arancione grafichiamo le fermate



## Cheapest insertion

totale km  
cheapest insertion:  
**14597.187234266958**

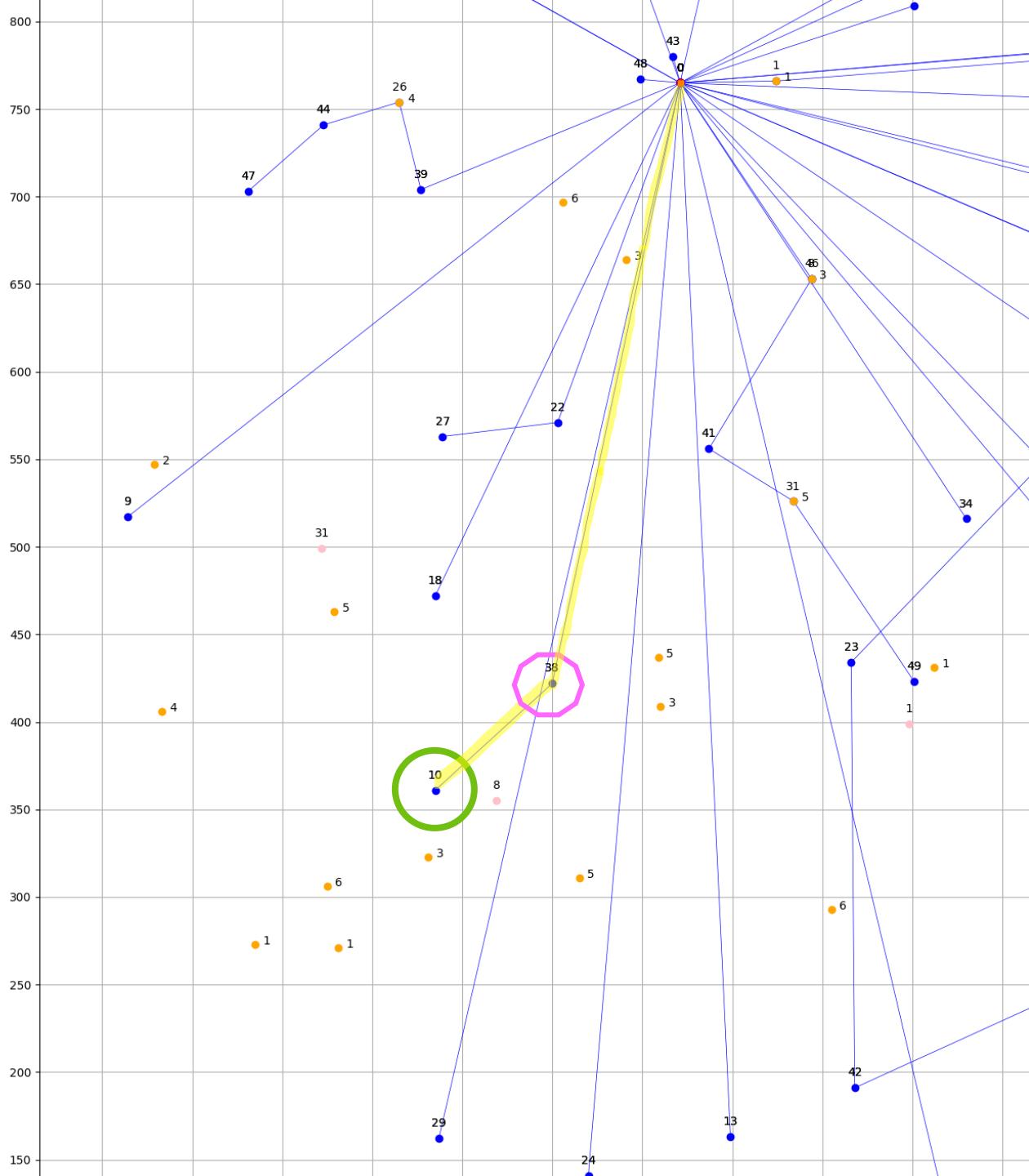




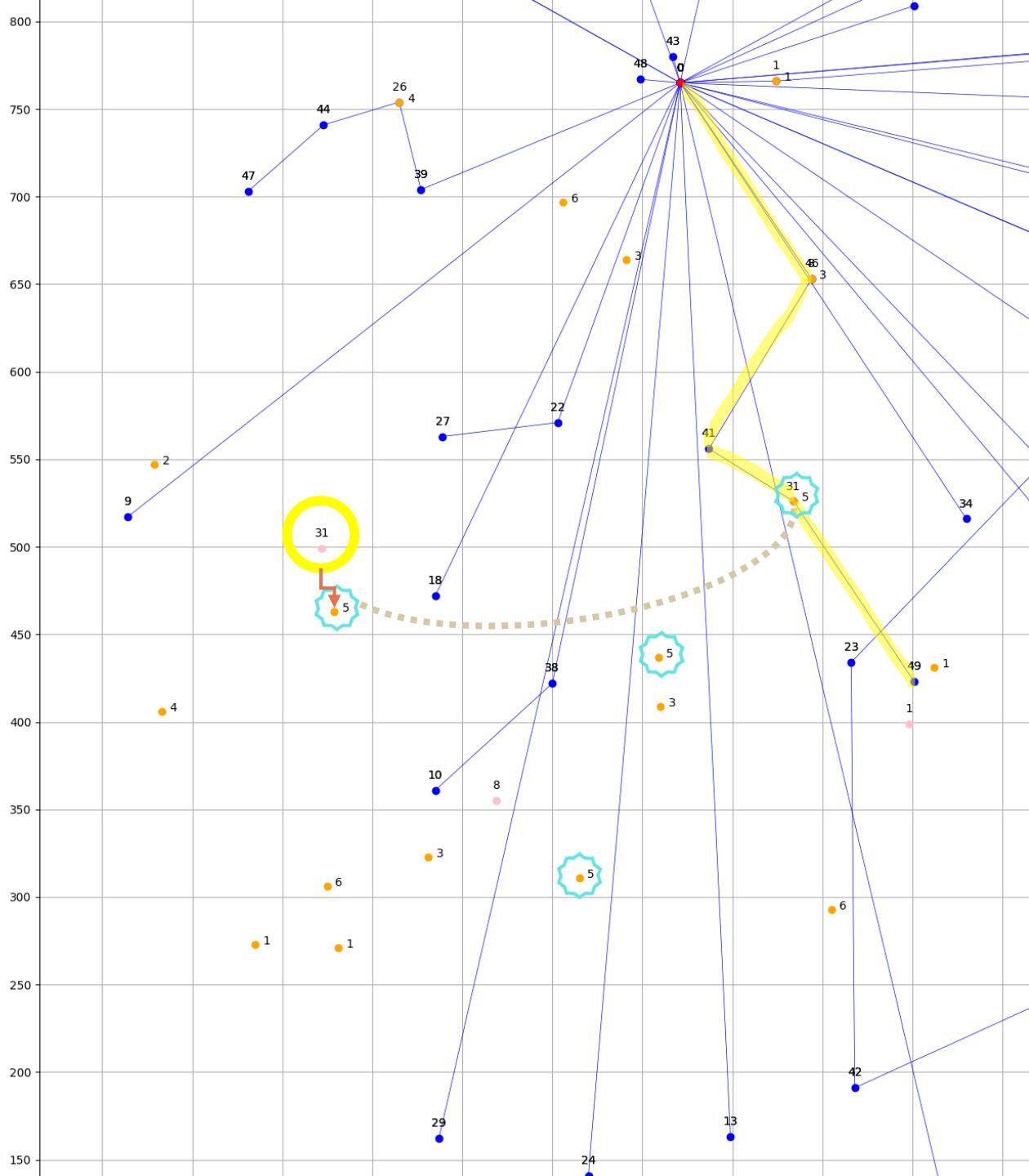
## Cheapest insertion



## Cheapest insertion



## Cheapest insertion



# La soluzione è migliorabile?



Cheapest insertion

Merge single nodes

Merge incomplete routes

Variable neighborhood search

Variable Neighborhood Descend

Variable Neighborhood Descend

Local search:  
Inter-Route

Variable Neighborhood Descend

Local search:  
Intra-Route

Local search:  
Intra-Route

Local search:  
Intra-Route



Cheapest insertion

Merge single nodes

Merge incomplete routes

Variable neighborhood search

Variable Neighborhood Descend

Variable Neighborhood Descend

Local search:  
Inter-Route

Variable Neighborhood Descend

Local search:  
Intra-Route

Local search:  
Intra-Route

Local search:  
Intra-Route

# Local search

Per ottimizzare le routes utilizziamo delle metaeuristiche che utilizzano la local search.

La ricerca locale (local search, LS) è l'algoritmo di riferimento per tutte le (meta)euristiche basate sul concetto di intorno (vicinato, neighborhood)

Ad ogni iterazione k-esima si tratta di risolvere un problema di ottimizzazione uguale al problema ( $F, c, \min$ ), dove  $F$  è la regione ammissibile, ristretto all'intorno  $N(x_k)$  della soluzione corrente  $x_k$  restituendo una qualsiasi soluzione migliore della corrente se questa esiste



Procedure Local Search ( $\min, F, c, x$ ):

Begin

$k=1$

$x_k := \text{Ammissibile}(F);$

while ( $\exists x^* \in N(x_k), x^* \neq x_k : cx^* < cx_k$ ) do

$k := k+1$

$x_k := x^*$

return( $x_k$ )

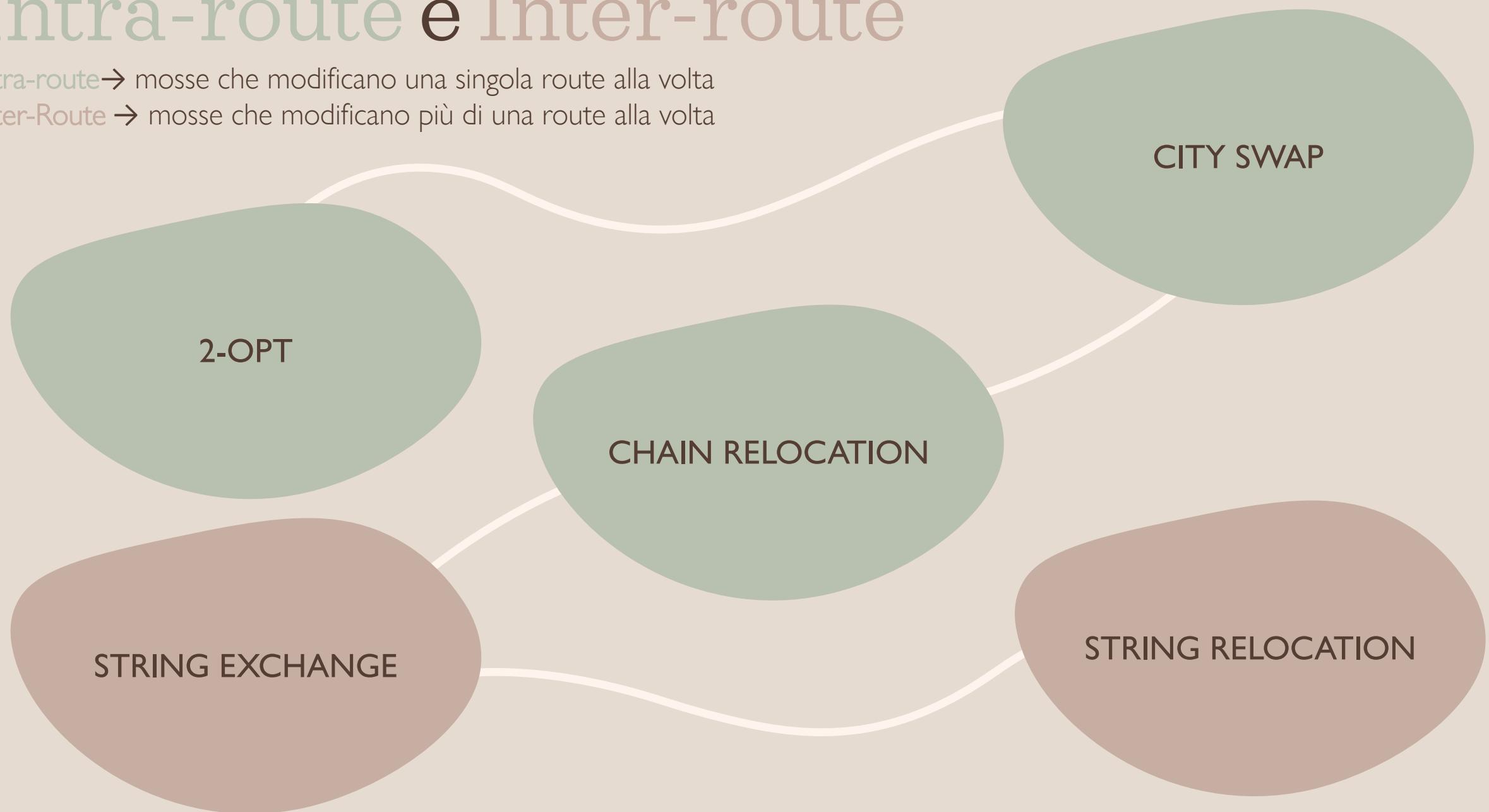
end



# Intra-route e Inter-route

Intra-route → mosse che modificano una singola route alla volta

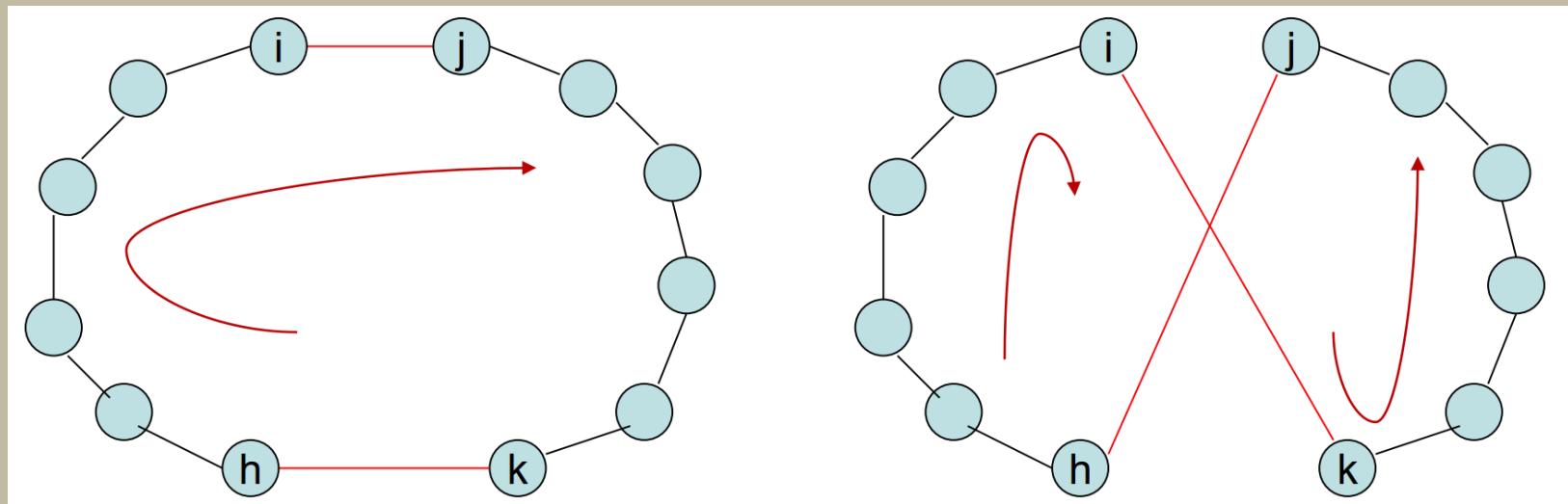
Inter-Route → mosse che modificano più di una route alla volta



# Intorno 2-OPT



L'intorno 2-opt è dato da tutti i cicli hamiltoniani (cammini che toccano tutti i vertici del grafo una e una sola volta) che differiscono dal ciclo corrente descritto dalla soluzione  $x$ , per una coppia di archi non adiacenti. Per ognuno degli  $n$  archi in soluzione ho  $n-3$  scelte (va escluso l'arco stesso e i due adiacenti) a disposizione per determinare il secondo arco.



OUT  
(h,k) (i,j)

IN  
(i,k) (h,j)

# Intorno 2-OPT



La route può essere modificata solo se la sua lunghezza è maggiore di 4.

```

def due_opt(route):
    routes = []
    if len(route.get_nodi()) < 4:
        routes.append(route)
    else:
        for i in range(1, len(route.get_nodi())-2):
            routes.extend(generate_routes_2opt(route, i))
    for el in routes:
        el.set_km()
    return routes

def generate_routes_2opt(route, i):
    r1 = []
    for j in range(i+1, len(route.get_nodi())-1):
        r1.append(swap_2opt(route, i, j))
    return r1

def swap_2opt(route, i, j):
    r1 = rt.Route()
    for k in range(0, i):
        r1.aggiungi_nodo(route.get_nodi()[k])
    for z in range(j, i-1, -1):
        r1.aggiungi_nodo(route.get_nodi()[z])
    for t in range(j+1, len(route.get_nodi())):
        r1.aggiungi_nodo(route.get_nodi()[t])
    return r1

```

Per ogni utente  $i$  non autista della mia route (escluso l'ultimo).

E per ogni utente  $j$  successivo ad  $i$ .

Prende in ingresso la route e due indici  $i$  e  $j$ . Inserisce in una nuova route tutti i nodi della route in ingresso invertendo però l'ordine dei nodi tra  $i$  e  $j$ .

# Intorno 2-OPT



Autsta1234Unife utilizzando 2-opt:

metto 1 in tutte le posizioni possibili successive a lui e specchio i valori tra la sua vecchia posizione e quella nuova:

A2134U OUT :(A,1)(2,3) IN :(A,2)(1,3)

A3214U

A4321U

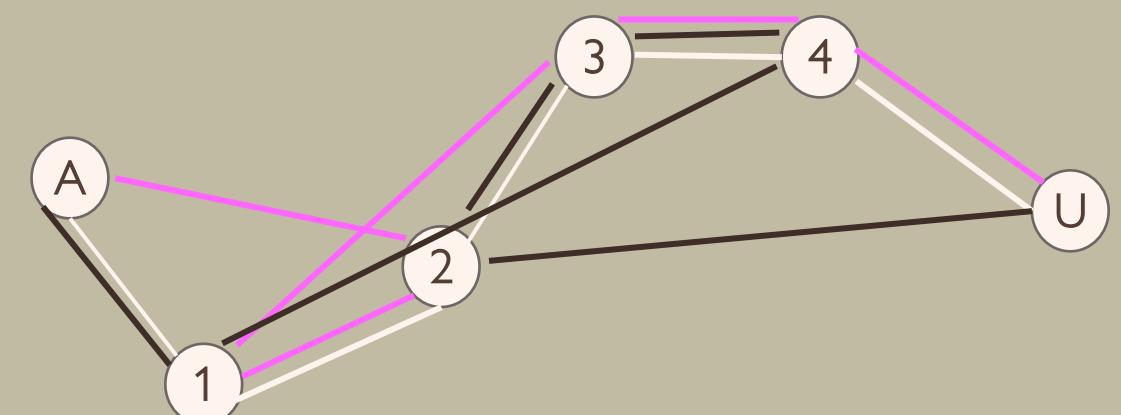
metto il 2 nelle posizioni possibili successive a lui e specchio i valori tra la sua vecchia posizione e quella nuova

A1324U

A1432U OUT :(1,2) (4,U) IN :(1,4) (2,U)

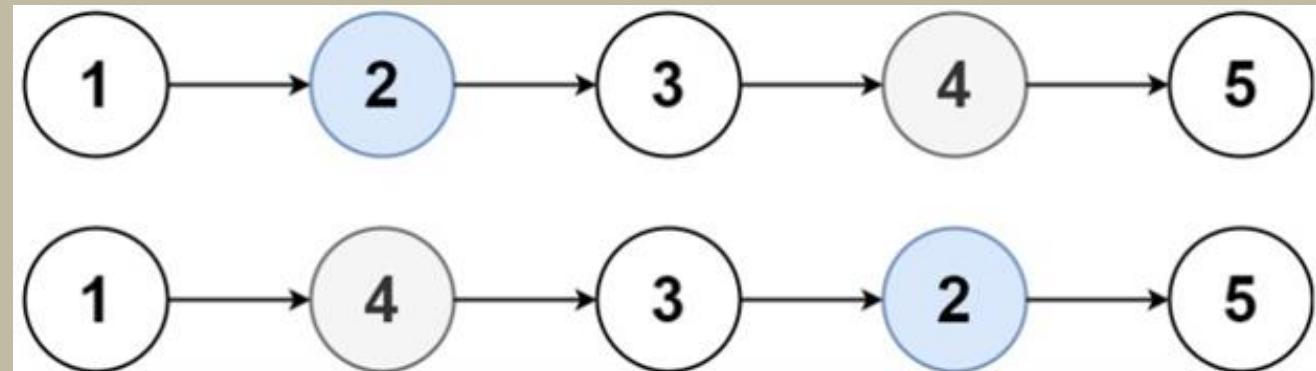
metto il 3 nelle posizioni possibili successive a lui specchio i valori tra la sua vecchia posizione e quella nuova

A1243U



# City-Swap ↲ ↳

City SWAP è l'intorno ottenuto dallo scambio di 2 nodi (anzichè 2 archi) nel ciclo.



# City-Swap

Intra-route

```
def node_swap(route):
    routes=[]
    if len(route.get_nodi()) <4:
        routes.append(route)
    else:
        for i in range(1, len(route.get_nodi())-2):
            routes.extend(generate_routes_node_swap(route, i))
        for el in routes:
            el.set_km()
    return routes

def generate_routes_node_swap(route, i):
    r1=[]
    for j in range(i+1, len(route.get_nodi())-1):
        r1.append(swap_node(route, i, j))
    return r1

def swap_node(route, i, j):
    r1=rt.Route()
    for k in range(0, i):
        r1.aggiungi_nodo(route.get_nodi()[k])
    r1.aggiungi_nodo(route.get_nodi()[j])
    for k in range(len(r1.get_nodi()), j):
        r1.aggiungi_nodo(route.get_nodi()[k])
    r1.aggiungi_nodo(route.get_nodi()[i])
    for k in range(len(r1.get_nodi()), len(route.get_nodi())):
        r1.aggiungi_nodo(route.get_nodi()[k])
    return r1
```

La route può essere modificata solo se la sua lunghezza è maggiore di 4.

Per ogni utente i della route escluso l'autista e l'ultimo passeggero

E per ogni utente j della route successivo ad i escluso UniFe.

Aggiungo nella route finale tutti i nodi prima di i. Poi aggiungo il nodo j. Poi tutti i nodi fino alla vecchia posizione di j. Poi il nodo i ed infine tutti i nodi successivi alla vecchia posizione di j.  
Otteniamo quindi la route iniziale avendo però scambiato il nodo i con il nodo j.

# ↔ City-Swap

Esempio: Autista,1,2,3,4,Unife , utilizzando City-Swap:

parto da 1:

A2134U  $1 \leftrightarrow 2$

A3213U

A4231U

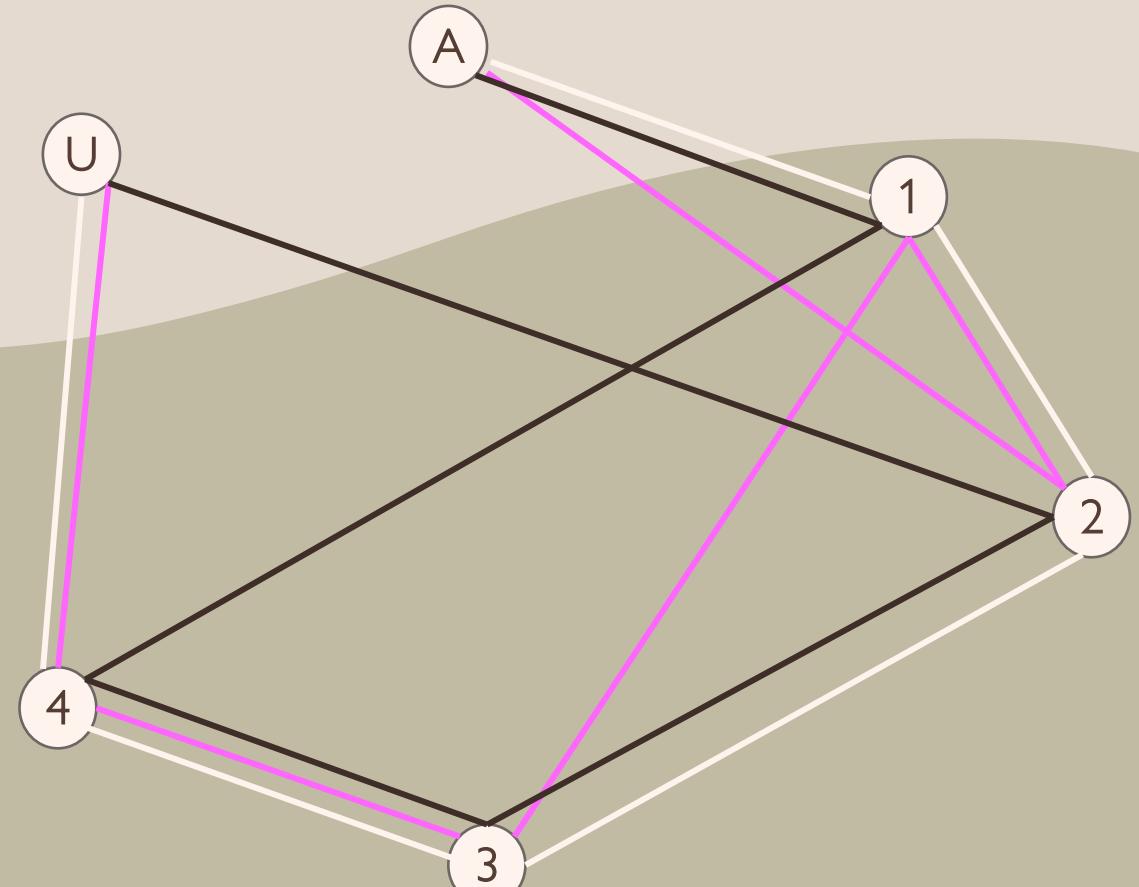
scambio il 2 con quelli in avanti:

A1324U

A1432U  $2 \leftrightarrow 4$

scambio il 3:

A1243U

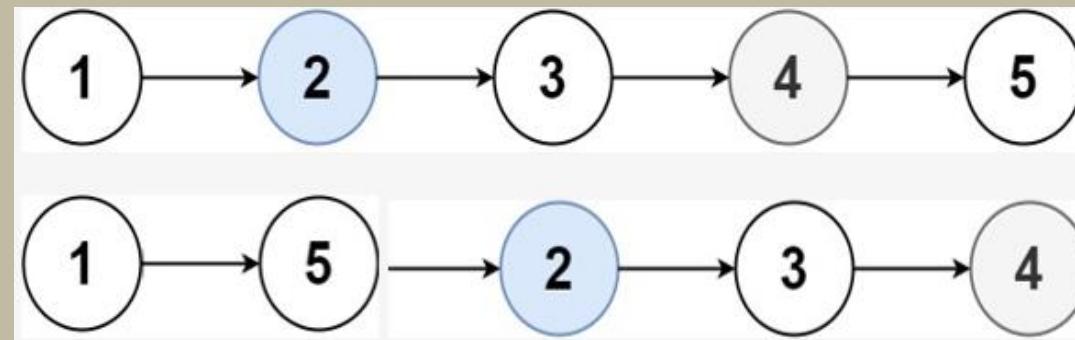




# Chain relocation

Chain Relocation è l'intorno ottenuto dalla rimozione di una sequenza di archi dal ciclo e dal suo reinserimento in altro punto.

Si eliminano i due archi prima e dopo la sequenza, e un terzo laddove si va a inserire. Si inseriscono 2 archi per riconnettere la sequenza al ciclo e un terzo per riconnettere il ciclo laddove è stata rimossa la sequenza.





# Chain relocation

La route può essere modificata solo se la sua lunghezza è maggiore di 4.

```
def chain_relocation(route):
    routes = []
    if (len(route.get_nodi()) < 5):
        routes.append(route)
    else:
        for i in range (1, len(route.get_nodi())-2):
            for j in range(i+1, len(route.get_nodi())-1):
                routes.extend(generate_routes_chain_relocation(route, i, j))
    for el in routes:
        el.set_km()
    return routes
```

Per ogni utente  $i$  (non autista) escluso l'ultimo e per ogni utente  $j$  successivo ad  $i$  (escluso Unife). La sequenza tra  $i$  e  $j$  è dunque quella che voglio spostare all'interno della route.

Aggiungo nella route  $relo\_nodes$  i nodi compresi tra  $i$  e  $j$  (inclusi). Ovvero stiamo selezionando la sequenza di nodi da riallocare.

Considero ogni possibile posizione  $k$  in cui posso inserire la mia sequenza  $i-j$

Aggiungo alla route  $r1$  tutti i nodi che non sono compresi tra  $i$  e  $j$  ( $nodes$ ) con indice diverso dalla posizione  $k$  in cui voglio allocare la sequenza  $i-j$ .

Se si considera il nodo  $t$  nella posizione  $k$  in cui voglio inserire la sequenza, aggiungo ad  $r1$  tutti i nodi della mia sequenza( $i-j$ ) e successivamente l'elemento  $t$ -esimo

```
def generate_routes_chain_relocation(route, i, j):
    r1 = []
    relo_nodes=rt.Route()
    for k in range(i, j+1):
        relo_nodes.aggiungi_nodo(route.get_nodi()[k])
    nodes=rt.Route()
    for k in range(0, i):
        nodes.aggiungi_nodo(route.get_nodi()[k])
    for k in range(j+1, len(route.get_nodi())):
        nodes.aggiungi_nodo(route.get_nodi()[k])
    for k in range(1, len(route.get_nodi())):
        temp=single_chain_relocation(nodes, relo_nodes, k, i)
        if temp.get_km()>0:
            r1.append(temp)
    return r1
```

```
def single_chain_relocation(nodes, relo_nodes, k, i):
    r1 = rt.Route()
    for t in range(0, len(nodes.get_nodi())):
        if(t!=k):
            r1.aggiungi_nodo(nodes.get_nodi()[t])
        else:
            if(k!=i):
                for z in range(0, len(relo_nodes.get_nodi())):
                    r1.aggiungi_nodo(relo_nodes.get_nodi()[z])
                r1.aggiungi_nodo(nodes.get_nodi()[t])
            r1.set_km2(1)
        else:
            r1.set_km2(-1)
    return r1
```

Se invece ho che  $t=k$  e  $k=i$  allora setto i km della route  $r1$  a -1 e dunque non considererò più questa route in quanto non corretta.

Aggiungo nella route  $nodes$  tutti i nodi prima di  $i$  (escluso) e dopo  $j$  (escluso). (ovvero tutti i nodi che non appartengono alla sequenza)



# Chain relocation

Esempio: Autista,1,2,3,4,Unife , utilizzando Chain relocation:

A3124U

A3412U

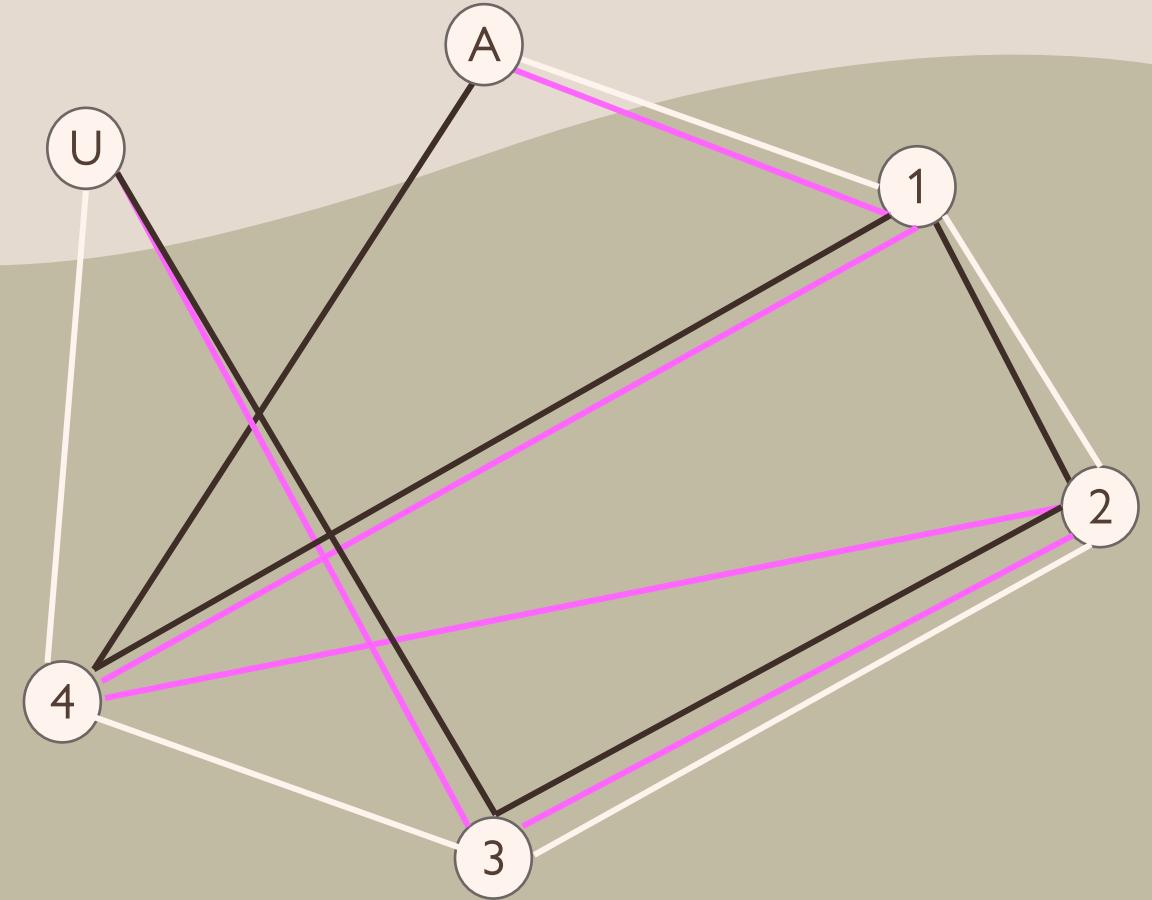
A4123U

A2314U

A1423U

A2341U

A1342U

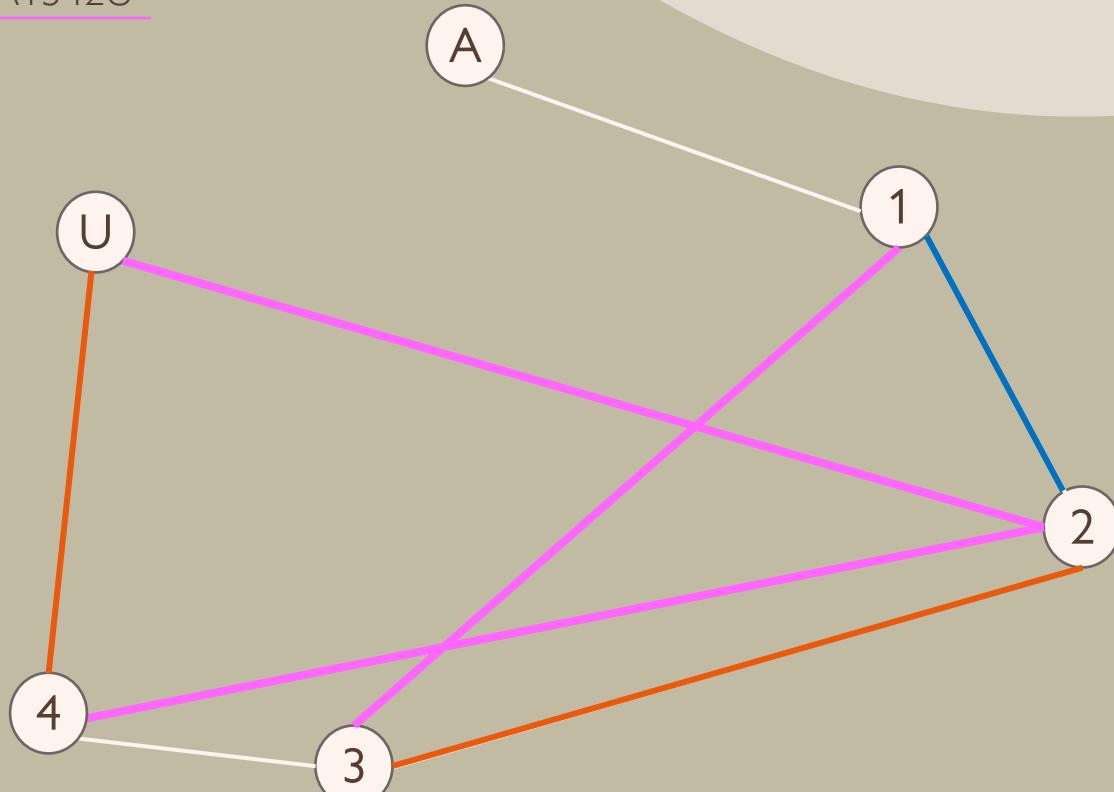




# Chain relocation

Esempio: Autista, 1,2,3,4,Unife , utilizzando Chain relocation:

A1342U

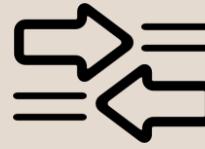


Si eliminano i **due archi prima e dopo la sequenza**, e **un terzo laddove si va a reinserire**. Si inseriscono 2 archi per riconnettere la sequenza al ciclo e un terzo per riconnettere il ciclo laddove è stata rimossa la sequenza.

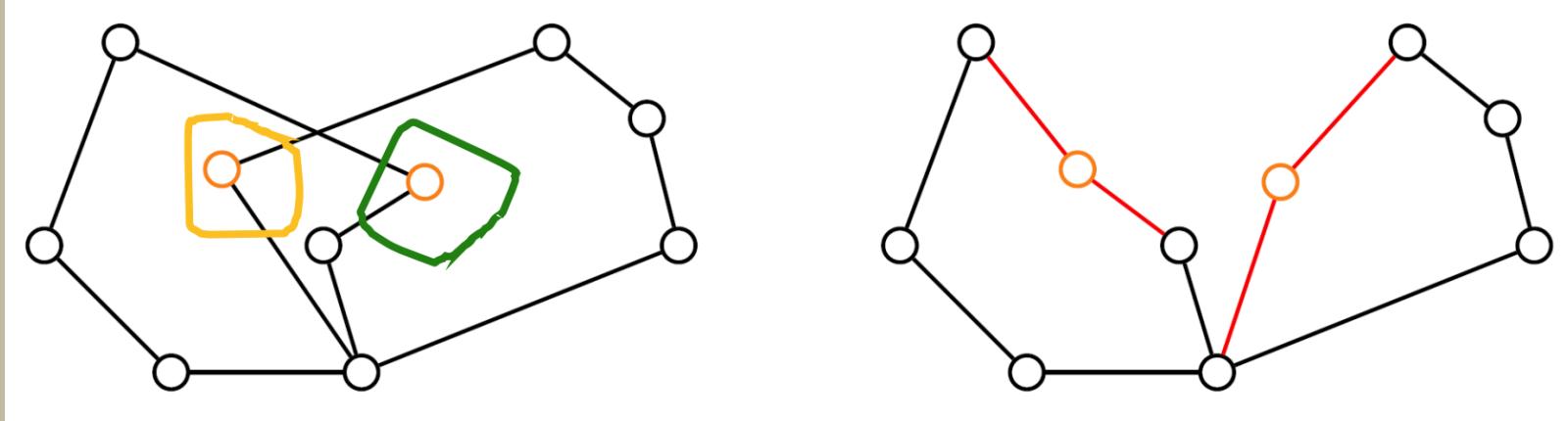


# String exchange

(swap)



Due stringhe formate da almeno k vertici sono scambiate tra due route.



# String exchange



## Inter-route

Se la lunghezza di una delle due route (o entrambe) è 3 chiamo la string\_exchange\_3 mettendo come primo argomento la route di lunghezza 3.

```
def string_exchange(r1, r2):
    routes = []
    if(len(r1.get_nodi())==3):
        routes = string_exchange_3(r1, r2, True)
    else:
        if(len(r2.get_nodi())==3):
            routes=string_exchange_3(r2, r1, False)
        else:
            if((len(r1.get_nodi())>3) and (len(r2.get_nodi())>3)):
                index = 0
                for i in range(1, len(r1.get_nodi())-1):
                    for k in range(1, len(r2.get_nodi())-1):
                        routes.insert(index, make_string_exchange(r1, r2, k, i))
                        index = index+1
                for i in range(1, len(r1.get_nodi())-2):
                    for k in range(1, len(r2.get_nodi())-2):
                        routes.insert(index, make_string_exchange_bis(r1, r2, k, i))
                        index=index+1
            return routes
```

Seleziono l'unico utente che non fa la macchina dalla route di dim=3. E per ogni utente che non fa la macchina della seconda route provo a scambiarli.

Se entrambe le route hanno dimensione maggiore di 3 gestisco lo scambio di uno o due nodi tra le route.

```
def string_exchange_3(r1, r2, flag):
    routes = []
    n = r1.get_nodi()[1]
    index=0
    for i in range(1, len(r2.get_nodi())-1):
        routes.insert(index, exchange_dim_3(r1, r2, n, i, flag))
        index=index+1
    return routes
```

```
def exchange_dim_3(r1, r2, n, i, flag):
    nodes1 = []
    nodes1.append(r2.get_nodi()[i])
    nodes2 = []
    nodes2.append(n)
    r = []
    if flag:
        r.insert(0, make_route_string_exchange(r1, nodes1, 1))
        r.insert(1, make_route_string_exchange(r2, nodes2, i))
    else:
        r.insert(0, make_route_string_exchange(r2, nodes2, i))
        r.insert(1, make_route_string_exchange(r1, nodes1, 1))
    return r
```

Scambio un nodo i di r1 con un nodo k di r2

```
def make_string_exchange(r1, r2, k, i):
    r = []
    nodes1 = []
    nodes2 = []
    nodes1.append(r2.get_nodi()[k])
    nodes2.append(r1.get_nodi()[i])
    r.insert(0, make_route_string_exchange(r1, nodes1, i))
    r.insert(1, make_route_string_exchange(r2, nodes2, k))
    return r
```

Scambio due nodi consecutivi (i e i+1) di r1 con due nodi consecutivi (k e k+1) di r2

```
def make_string_exchange_bis(r1, r2, k, i):
    r = []
    nodes1 = []
    nodes2 = []
    nodes1.append(r2.get_nodi()[k])
    nodes1.append(r2.get_nodi()[k+1])
    nodes2.append(r1.get_nodi()[i])
    nodes2.append(r1.get_nodi()[i+1])
    r.insert(0, make_route_string_exchange(r1, nodes1, i))
    r.insert(1, make_route_string_exchange(r2, nodes2, k))
    return r
```

Aggiungo nella posizione i della route r i nodi di nodes.

# ➡➡ String exchange

Esempio:

R1 → Autista,1,Unife

R2 → Autista',2,3,4,Unife

parto scambiando 1 con 2:

R1 → A2U

R2 → A'134U

poi scambio 1 con 3:

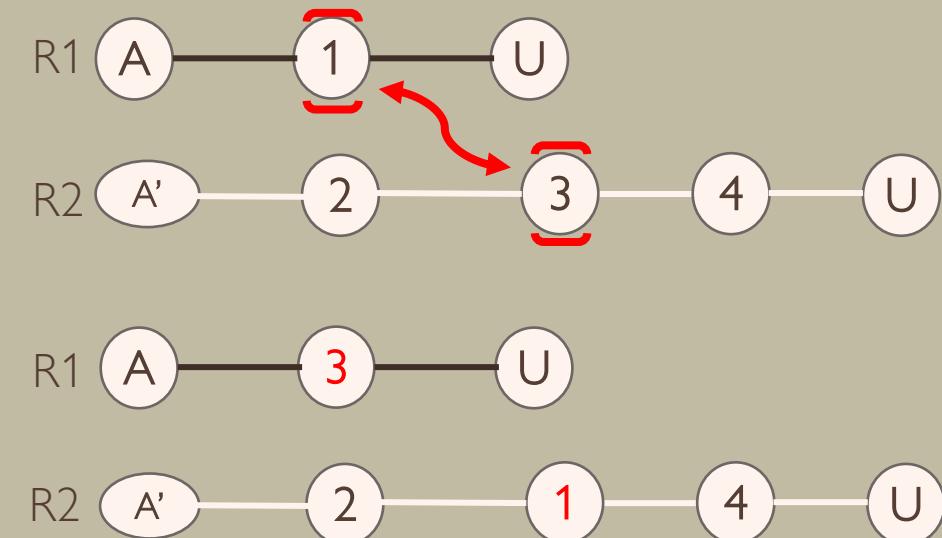
R1 → A3U

R2 → A'214U

poi scambio 1 con 4:

R1 → A4U

R2 → A'231U



# String exchange ➡➡

Esempio

R1 → Autista,1,2,3,Unife

R2 → Autista',4,5,6,7,Unife

parto con gli scambi di un nodo:

R1 → A423U ; R2 → A'1567U

R1 → A523U ; R2 → A'4167U

R1 → A623U ; R2 → A'4517U

R1 → A723U ; R2 → A'4561U

R1 → A143U ; R2 → A'2567U

R1 → A153U ; R2 → A'4267U

R1 → A163U ; R2 → A'4527U

R1 → A173U ; R2 → A'4562U

R1 → A124U ; R2 → A'3567U

R1 → A125U ; R2 → A'4367U

R1 → A126U ; R2 → A'4537U

R1 → A127U ; R2 → A'4563U

Passo agli scambi di due nodi:

R1 → A453U ; R2 → A'1267U

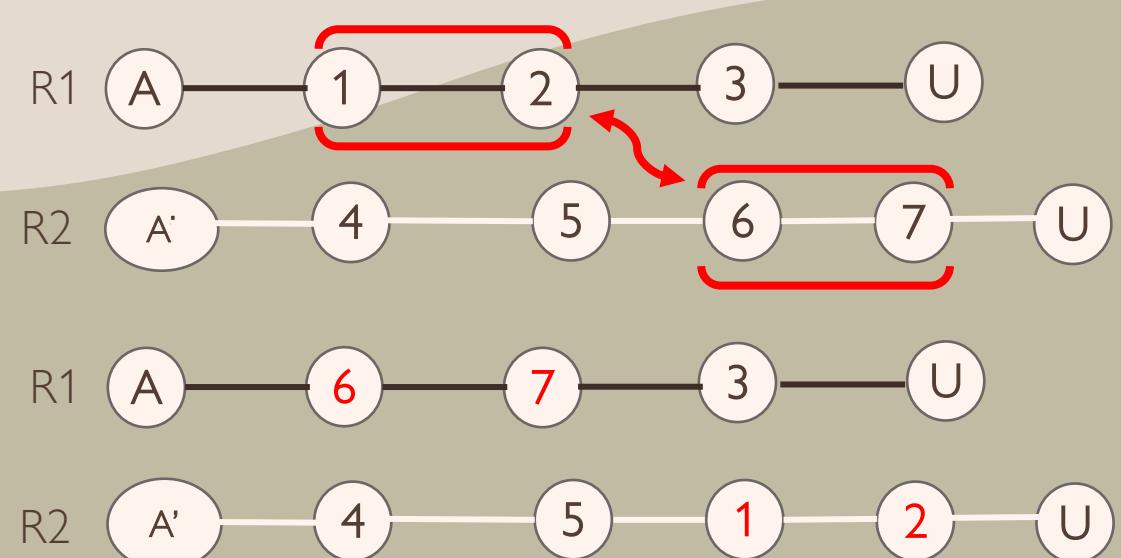
R1 → A563U ; R2 → A'4127U

R1 → A673U ; R2 → A'4512U

R1 → A145U ; R2 → A'2367U

R1 → A156U ; R2 → A'4237U

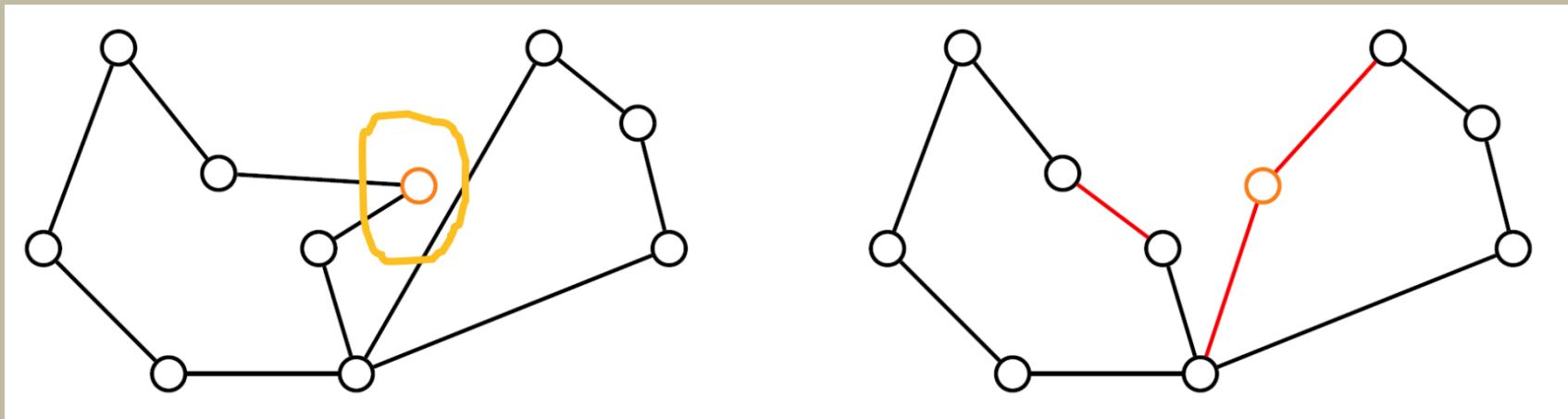
R1 → A167U ; R2 → A'4523U





# String relocation (move)

Una stringa formata da al massimo k vertici viene spostata da una route ad un'altra, di solito k=1 oppure k=2.





# String relocation

```

def string_relocation(rt.Route r1, rt.Route r2):
    routes = []
    if(len(r1.get_nodi())==6 and len(r2.get_nodi())==6):
        r=[]
        r.insert(0, r1)
        r.insert(1, r2)
        routes.append(r)
        return routes

    if(len(r1.get_nodi())==6 or len(r2.get_nodi())==6):
        if len(r1.get_nodi())==6:
            routes=string_relo(r1, r2, True)
        else:
            routes=string_relo(r2, r1, False)
    else:
        copy_r1 = rt.Route()
        copy_r2=rt.Route()
        for i in range(0, len(r1.get_nodi())):
            copy_r1.aggiungi_nodo(r1.get_nodi()[i])
        for i in range(0, len(r2.get_nodi())):
            copy_r2.aggiungi_nodo(r2.get_nodi()[i])
        routes=string_relo(r1, r2, True)
        routes.extend(string_relo(copy_r2, copy_r1, False))
    return routes

```

Se entrambe le route hanno dimensione 6 non faccio alcuna modifica.

Se solo una delle due ha lunghezza 6 allora chiamo la string\_relo con primo argomento quella da 6.

Altrimenti chiamo la string\_relo due volte: una con prima r1 e poi r2 e una con prima r2 poi r1..

```

def string_relo(rt.Route r1, rt.Route r2, flag):
    routes = []
    for i in range(1, len(r1.get_nodi())-1):
        for j in range(1, len(r2.get_nodi())):
            routes.append(pos(r1, r2, i, j, flag))
    return routes

```

Per ogni nodo i di r1 e ogni posizione j di r2 chiamo la pos.

```

def pos(rt.Route r1, rt.Route r2, i, j, flag):
    r=[]
    if(flag):
        r.insert(0, remove_node(r1, i))
        r.insert(1, add_node(r2, j, r1.get_nodi()[i]))
    else:
        r.insert(0, add_node(r2, j, r1.get_nodi()[i]))
        r.insert(1, remove_node(r1, i))
    return r

```

Rimuovo il nodo i-esimo dalla r1 e lo aggiungo nella j-esima posizione della r2 (facendo scorrere quindi i nodi da j compreso in poi).

```

def remove_node(route, i):
    r = rt.Route()
    for j in range(0, i):
        r.aggiungi_nodo(route.get_nodi()[j])
    for j in range(i+1, len(route.get_nodi())):
        r.aggiungi_nodo(route.get_nodi()[j])
    return r

```

Rimuovo il nodo i-esimo dalla route inserendo tutti i nodi prima di i (escluso) e tutti i nodi dopo i (escluso).

```

def add_node(route, i, node):
    r = rt.Route()
    for j in range(0, i):
        r.aggiungi_nodo(route.get_nodi()[j])
    r.aggiungi_nodo(node)
    for j in range(i, len(route.get_nodi())):
        r.aggiungi_nodo(route.get_nodi()[j])
    return r

```

Aggiungo alla route il nodo node nella i-esima posizione (facendo scorrere gli elementi dopo).

Esempio:

R1 → Autista,1,2,Unife

R2 → Autista',3,4,5,Unife

parto spostando i nodi di R1 e mettendoli in R2:

R1 → A2U ; R2 → A'1345U

R1 → A2U ; R2 → A'3145U

R1 → A2U ; R2 → A'3415U

R1 → A2U ; R2 → A'3451U

R1 → A1U ; R2 → A'2345U

R1 → A1U ; R2 → A'3245U

R1 → A1U ; R2 → A'3425U

R1 → A1U ; R2 → A'3452U

passo a spostare i nodi di R2 e mettendoli in R1:

R1 → A312U ; R2 → A'45U

R1 → A132U ; R2 → A'45U

R1 → A123U ; R2 → A'45U

# String relocation



R1 → A412U ; R2 → A'35U

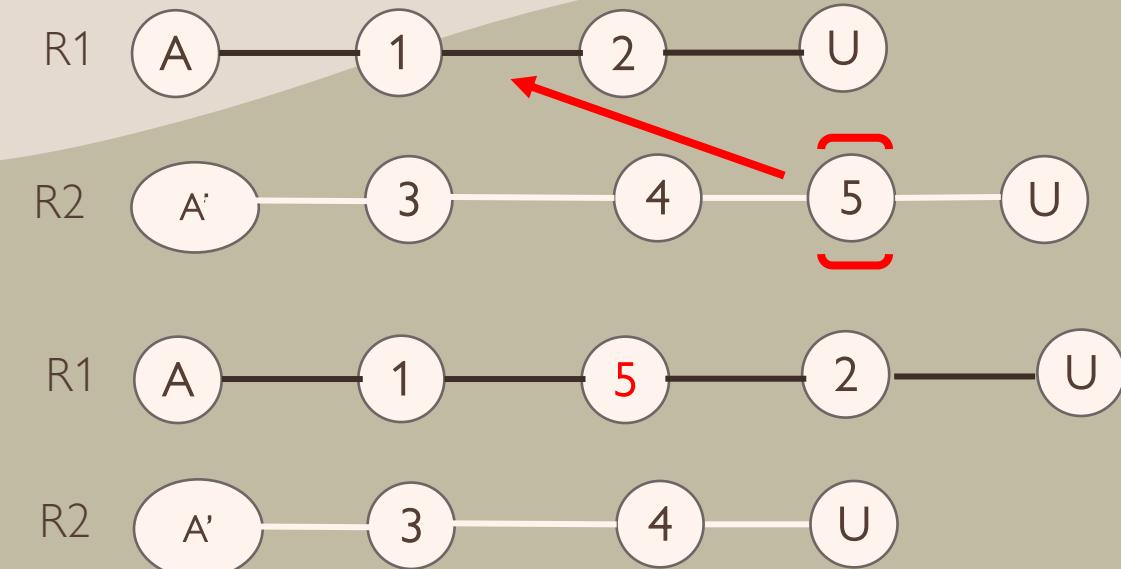
R1 → A142U ; R2 → A'35U

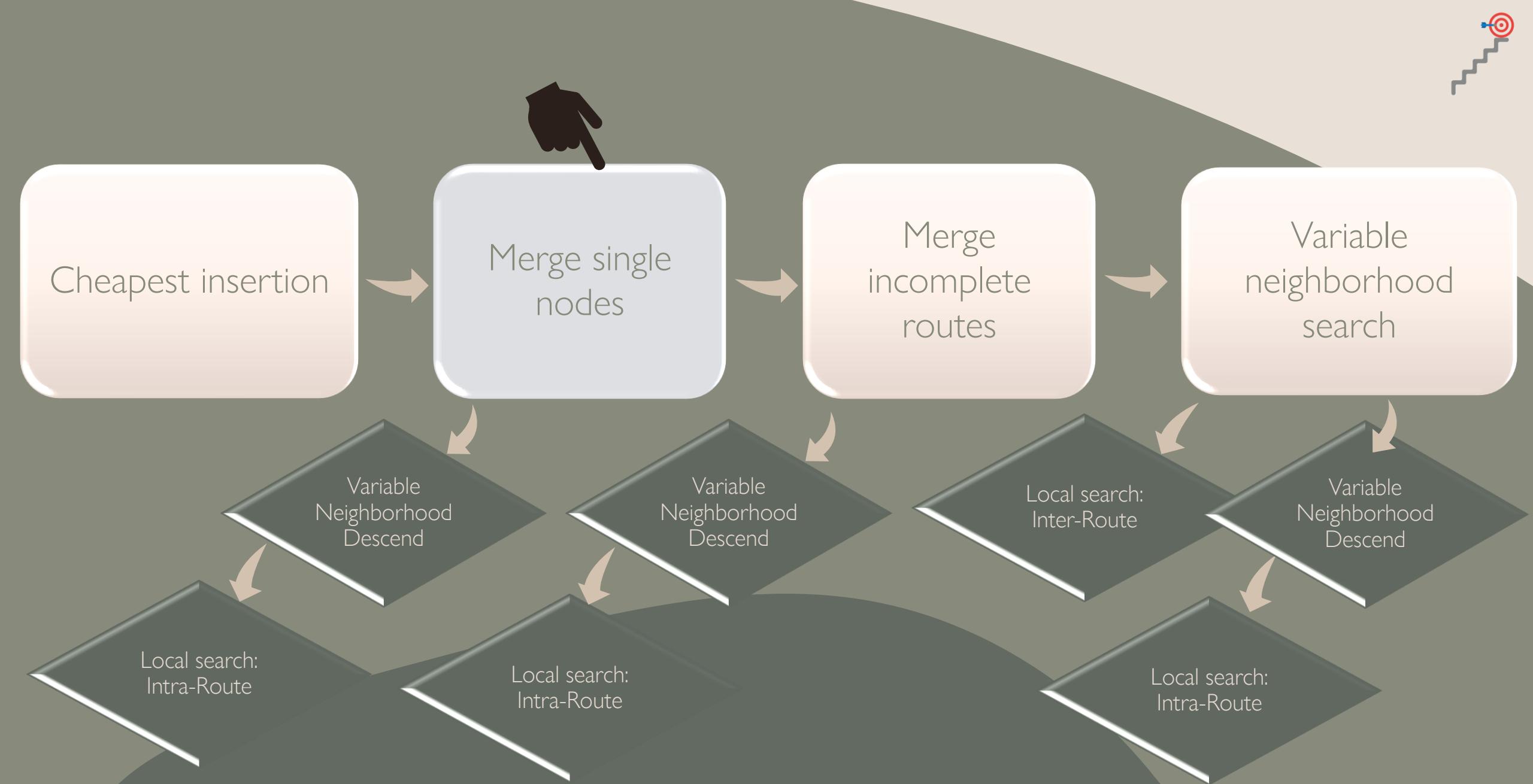
R1 → A124U ; R2 → A'35U

R1 → A512U ; R2 → A'34U

R1 → A152U ; R2 → A'34U

R1 → A125U ; R2 → A'34U





# Merge single nodes

```
def merge_single_nodes(current_solution, solo_route_list):
    soluzioni=[]
    soluzioni.append(current_solution)
    new_current_solution=ut.copy_routes(current_solution)
    incomplete_route_list=find_incomplete_route(current_solution)
    while(len(solo_route_list)!=0 and len(incomplete_route_list)!=0):
        for route in incomplete_route_list:
            soluzioni.append(merging_single_node(new_current_solution, solo_route_list[0], route))

    solo_route_list.pop(0)
    index=0
    bestkm=ut.kmtot(soluzioni[0])
    for j in range(1, len(soluzioni)):
        if ut.kmtot(soluzioni[j]) < bestkm:
            index=j
            bestkm=ut.kmtot(soluzioni[j])

    new_current_solution=ut.copy_routes(soluzioni[index])

    incomplete_route_list.clear()
    incomplete_route_list=find_incomplete_route(new_current_solution)

    soluzioni.clear()
    soluzioni.append(new_current_solution)

return new_current_solution
```

Prende in ingresso la lista delle soluzioni correnti e la lista dei passeggeri che vanno ad UniFe da soli

Seleziono tra tutte le possibili route date dal passeggero combinato con le varie route incomplete quella che minimizza i km

```
def merging_single_node(current_solution, solo, incomplete):
    new_route = rt.Route()
    new_solution=ut.copy_routes(current_solution)
    new_route.aggiungi_nodo(incomplete.get_nodi()[0])
    new_route.aggiungi_nodo(solo.get_nodi()[0])
    for j in range(1, len(incomplete.get_nodi())):
        new_route.aggiungi_nodo(incomplete.get_nodi()[j])
    new_route_vnd=meta.variable_neighborhood_descend(new_route)
    for el in new_solution:
        if solo.get_nodi()[0] in el.get_nodi():
            new_solution.remove(el)
    for el in new_solution:
        if incomplete.get_nodi()[0] in el.get_nodi():
            new_solution.remove(el)
    new_solution.append(new_route_vnd)
    return new_solution
```

```
def find_incomplete_route(current_solution):
    incomplete=[]
    for el in current_solution:
        if len(el.get_nodi())==2 and el.get_nodi()[0].get_c()==5:
            incomplete.append(el)
        else:
            if len(el.get_nodi()) >2 and len(el.get_nodi())<6:
                incomplete.append(el)

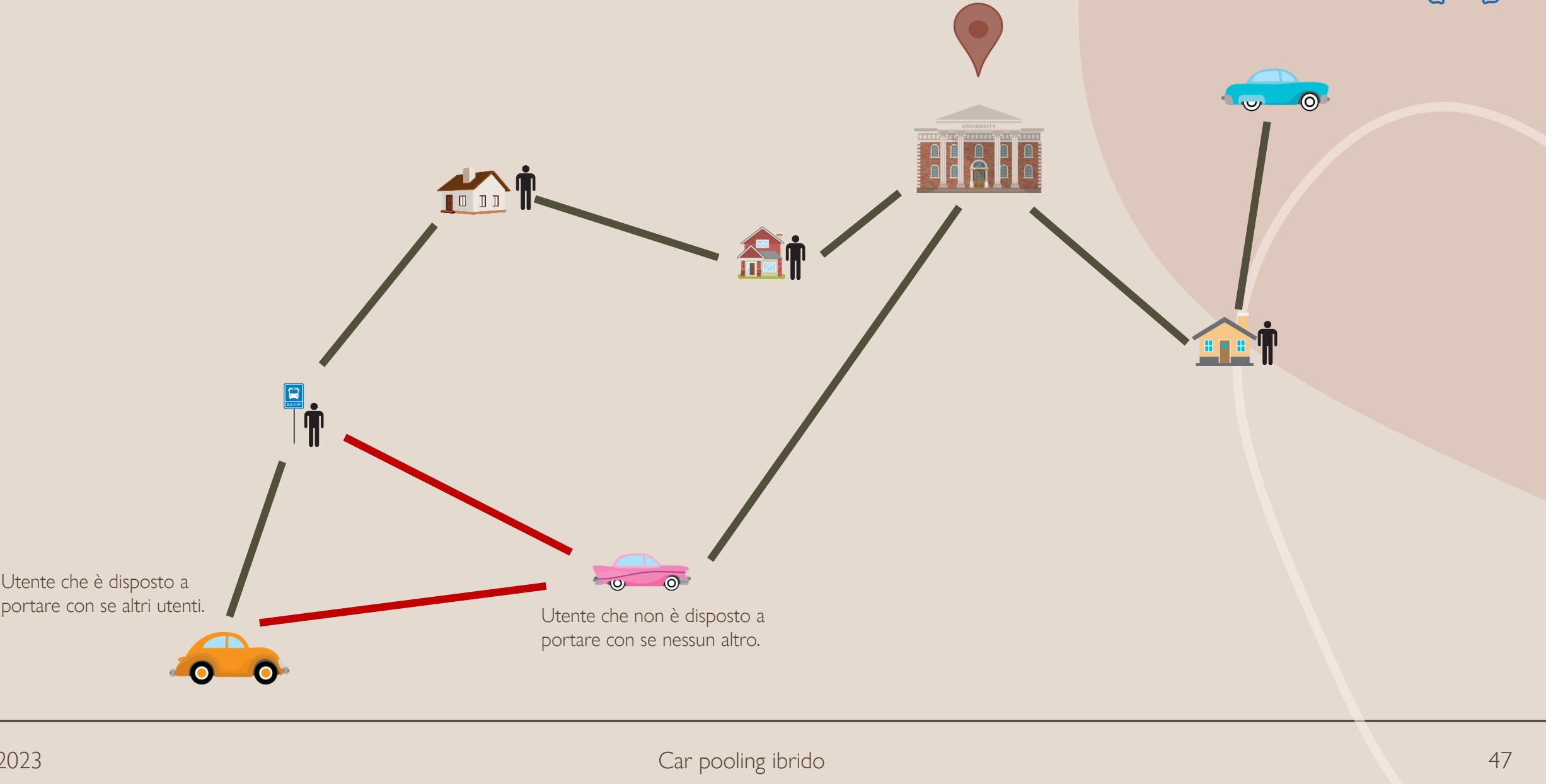
    return incomplete
```

A partire dalla soluzione corrente, una route formata da un passeggero che va da solo ad UniFe e da una route incompleta, mi fonde queste due route in una unica. Elimina la route del passeggero e quella incompleta dalle mie soluzioni correnti e ci aggiunge quella nuova.

Ottimizzo le soluzioni con la VND

A partire dalla soluzione corrente restituisco l'insieme delle route formate o da un autista che va ad UniFe senza prelevare nessuno oppure le route che non raggiungono la capacità massima.

# Merge single nodes



# Visualizziamo i risultati



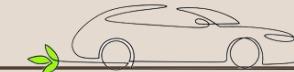
```
routes_merge1=mer.merge_single_nodes(routes2, solo_routes)  
for el in routes_merge1:  
    print(el)
```

```
gf.grafico(lista1, lista2, routes_merge1, 'figura2.png')
```

```
routes_merge2= mer.merge_incomplete_routes(routes_merge1)  
for el in routes_merge2:  
    print(el)
```

```
gf.grafico(lista1, lista2, routes_merge2, 'figura3.png')
```

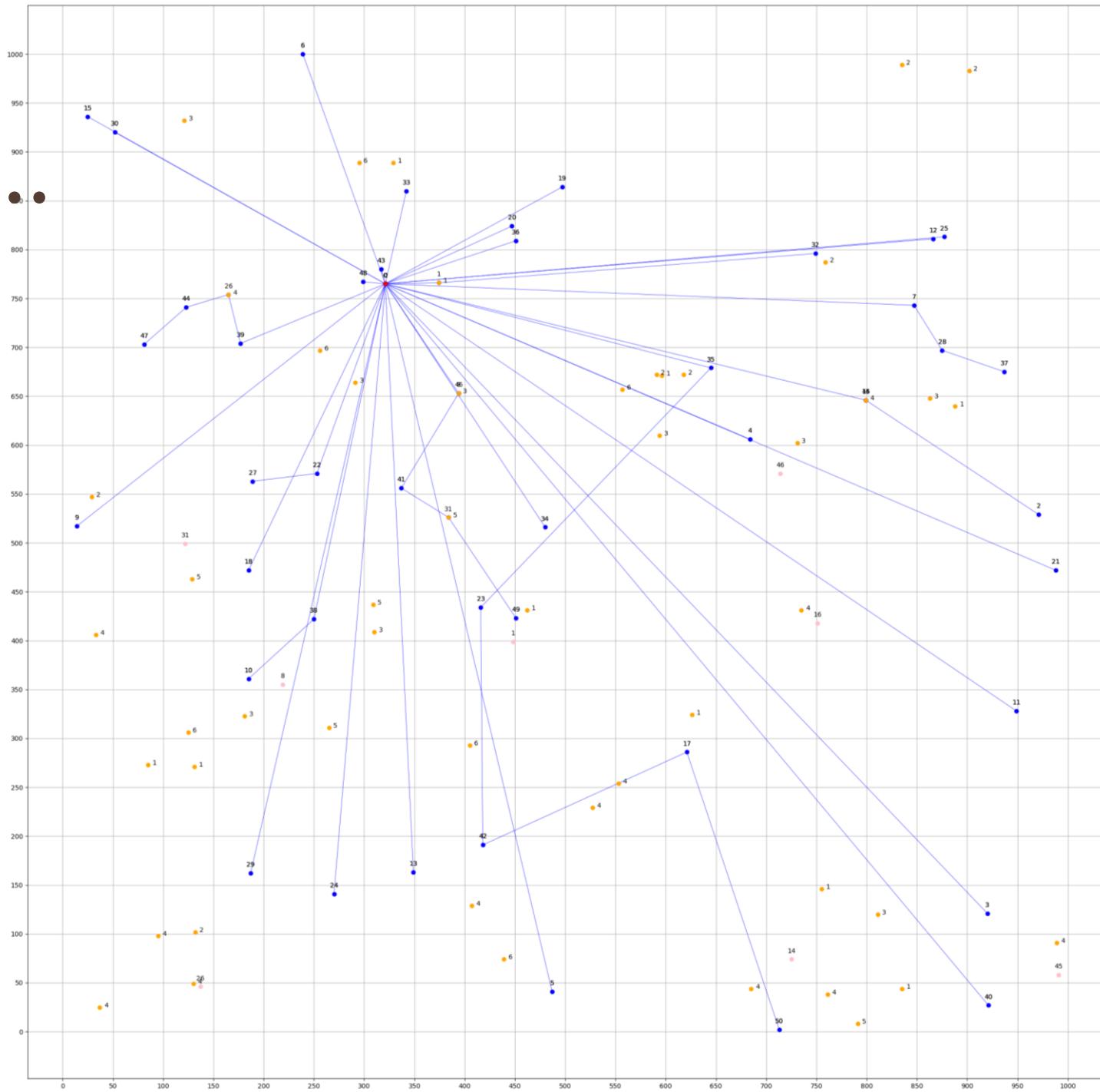
Grafichiamo i risultati della merge  
single nodes



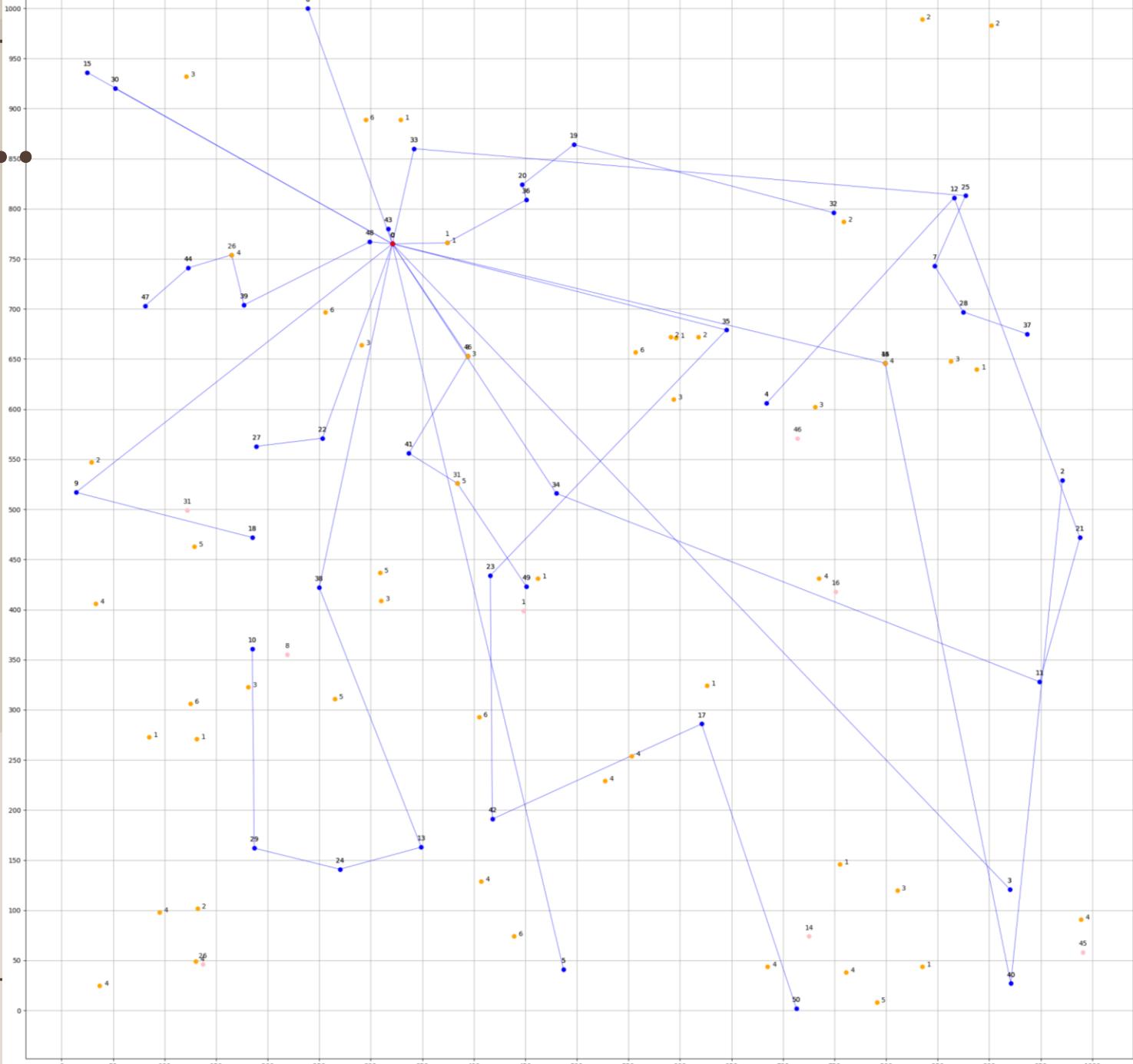
# Prima..

Cheapest insertion

totale km  
cheapest insertion:  
**14597.187234266958**



# Dopo.

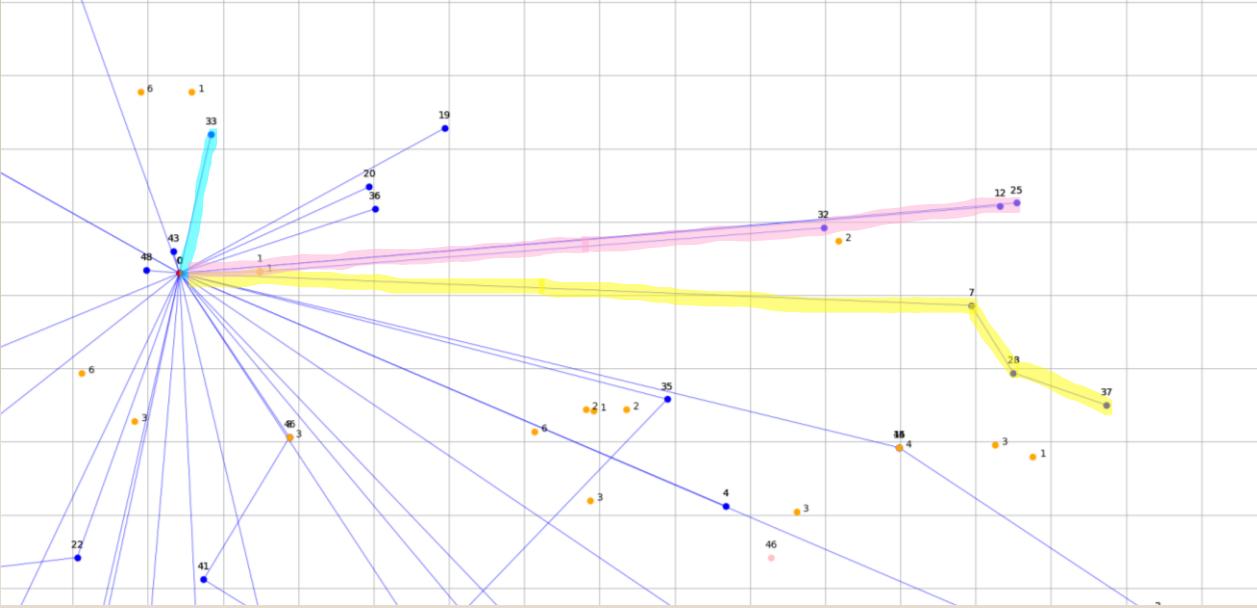


## Merge single nodes

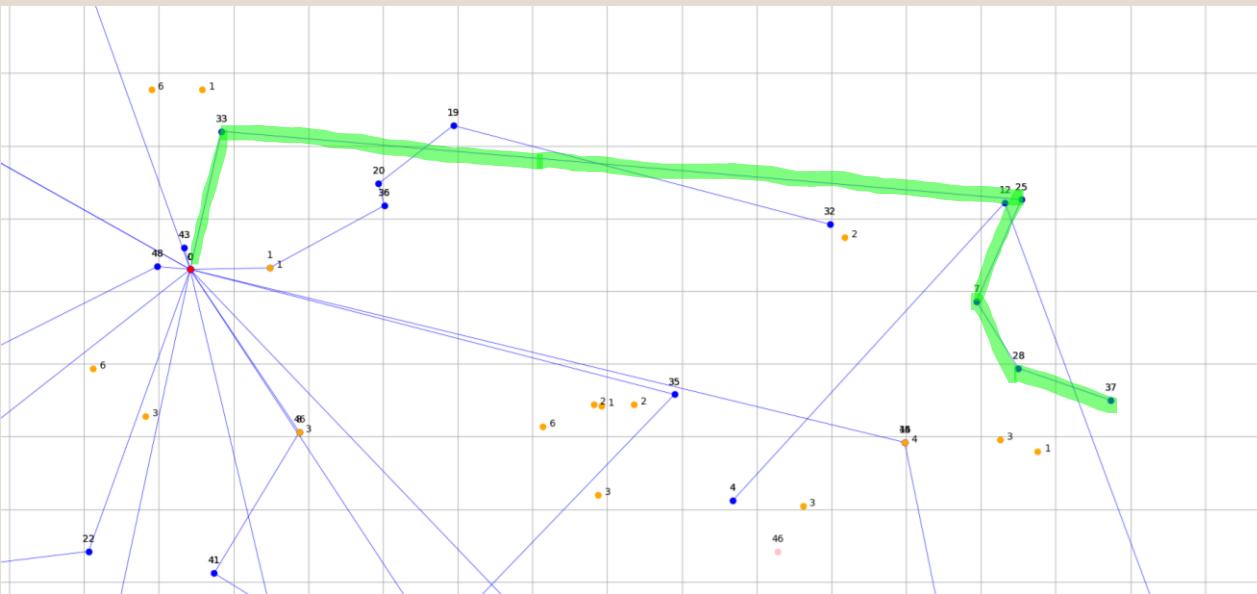
totale km  
merge single nodes:  
**11071.467509599799**

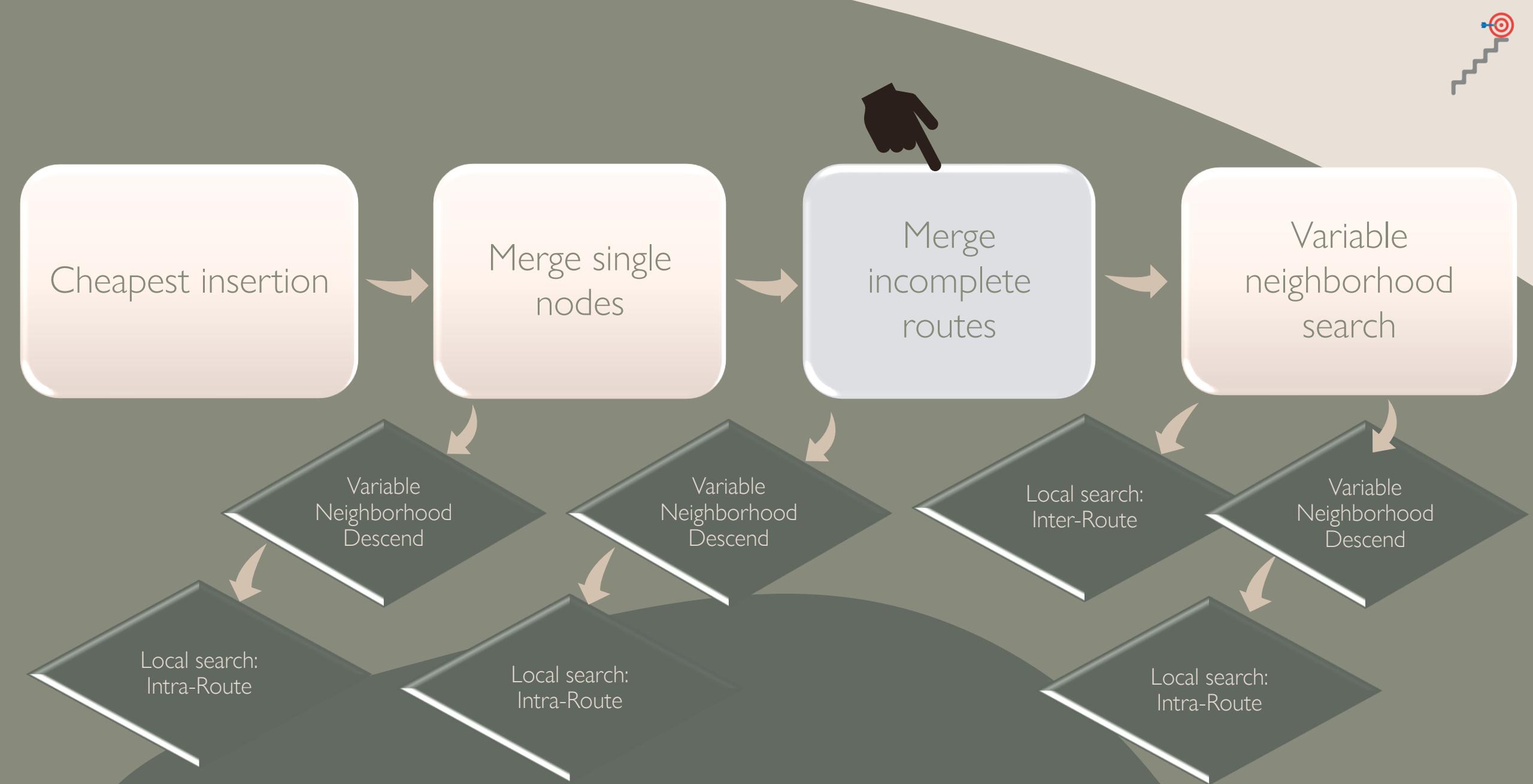


Cheapest insertion



Merge single nodes





# Merge incomplete routes ↑



```

def merge_incomplete_routes(current_solution):
    possibili_soluzioni=[]
    possibili_soluzioni.append(current_solution)
    new_current_solution=ut.copy_routes(current_solution)
    incomplete_route_list=find_incomplete_route(current_solution)
    not_consider=[]

    while len(incomplete_route_list) >1:
        for i in range(1, len(incomplete_route_list)):
            if (len(incomplete_route_list[0].get_nodi()) + len(incomplete_route_list[i].get_nodi())) <=7:
                possibili_soluzioni.append(merging_incomplete_routes(new_current_solution, incomplete_route_list[0], incomplete_route_list[i]))

        index=0
        best_km=ut.kmtot(possibili_soluzioni[0])
        for j in range(1, len(possibili_soluzioni)):
            if ut.kmtot(possibili_soluzioni[j])<best_km:
                index=j
                best_km=ut.kmtot(possibili_soluzioni[j])

        if index ==0:
            not_consider.append(incomplete_route_list[0])

        new_current_solution=ut.copy_routes(possibili_soluzioni[index])
        incomplete_route_list.clear()
        incomplete_route_list=find_incomplete_route(new_current_solution)

        for el in not_consider:
            for el2 in incomplete_route_list:
                if el.get_nodi()[0] in el2.get_nodi():
                    incomplete_route_list.remove(el2)

        possibili_soluzioni.clear()
        possibili_soluzioni.append(new_current_solution)

    return new_current_solution

```

Per ogni route incompleta scorro tutte le altre route incomplete e per ogni combinazione chiamo la merging\_incomplete\_routes

Scelgo la route, formata dalla combinazione della route che sto esaminando e tutte le altre, che mi minimizza i km.

```

def merging_incomplete_routes(current_solution, r1, r2):
    routes=[]
    new_routes=[]
    new_solution=ut.copy_routes(current_solution)
    autisti = []

    for i in range(0, len(r1.get_nodi())):
        if r1.get_nodi()[i].get_c()==5:
            autisti.append(r1.get_nodi()[i])

    for i in range(0, len(r2.get_nodi())):
        if r2.get_nodi()[i].get_c()==5:
            autisti.append(r2.get_nodi()[i])

    for i in range(0, len(autisti)):
        routes.append(make_route(autisti[i], r1, r2))

    for i in range(0, len(routes)):
        new_routes.append(meta.variable_neighborhood_descend(routes[i]))

    for el in new_solution:
        if r1.get_nodi()[0] in el.get_nodi():
            new_solution.remove(el)
    for el in new_solution:
        if r2.get_nodi()[0] in el.get_nodi():
            new_solution.remove(el)

    best_km=new_routes[0].get_km()
    index=0
    for i in range(1, len(new_routes)):
        if new_routes[i].get_km() < best_km:
            index=i
            best_km = new_routes[i].get_km()

    new_solution.append(new_routes[index])
    return new_solution

```

Creo una lista composta dai due autisti delle due route.

Per ogni autista chiamo la make\_route insieme alle due route.

Ottimizzo le soluzioni con la VND

A partire dalla soluzione corrente restituisco l'insieme delle route formate o da un autista che va ad UniFe senza prelevare nessuno oppure le route che non raggiungono la capacità massima.

```

def find_incomplete_route(current_solution):
    incomplete=[]
    for el in current_solution:
        if len(el.get_nodi())==2 and el.get_nodi()[0].get_c()==5:
            incomplete.append(el)
        else:
            if len(el.get_nodi()) >2 and len(el.get_nodi())<6:
                incomplete.append(el)

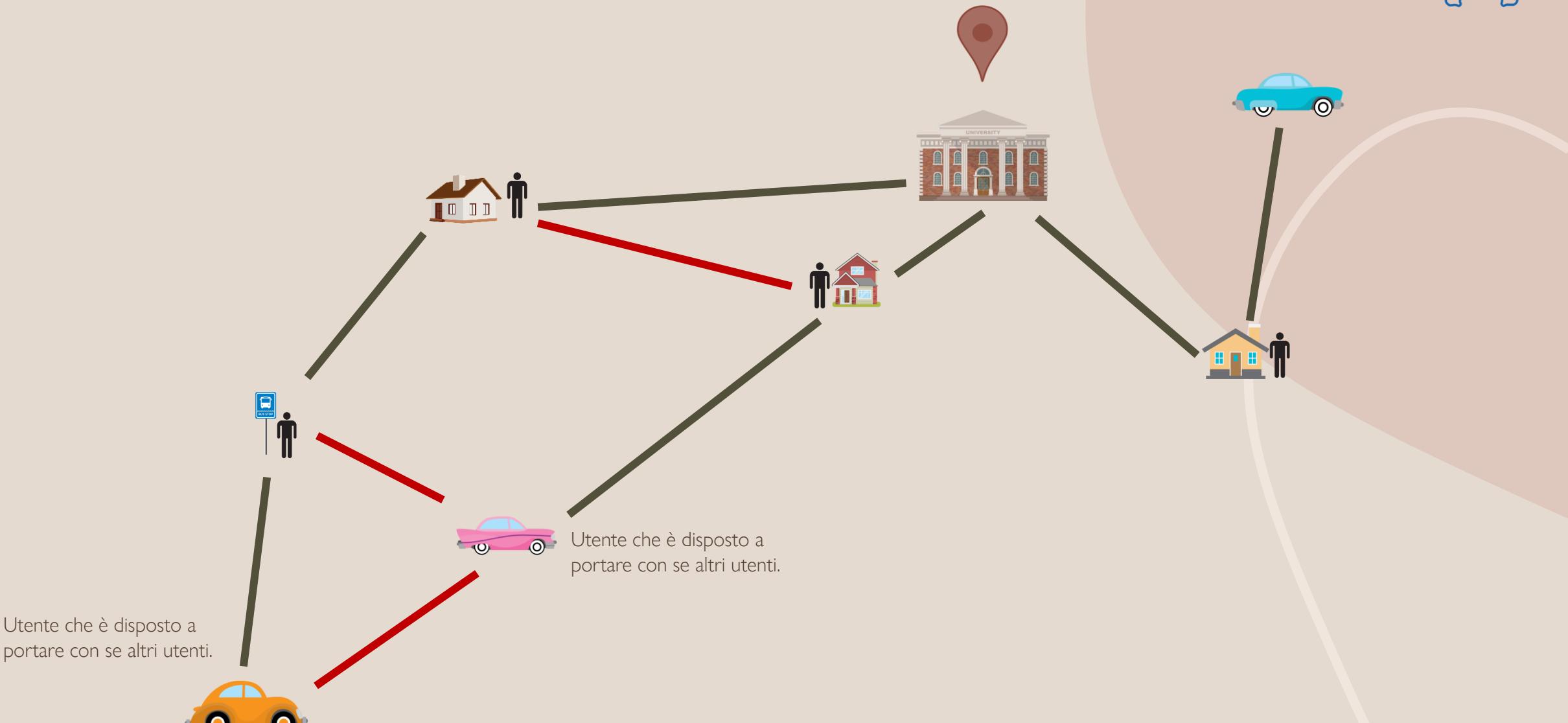
    return incomplete

def make_route(autista, r1, r2):
    route = rt.Route()
    route.aggungi_nodo(autista)
    if ((len(r1.get_nodi())-1) + len(r2.get_nodi())) <=6:
        for i in range(0, len(r1.get_nodi())-1):
            if r1.get_nodi()[i].get_id() != autista.get_id():
                route.aggungi_nodo(r1.get_nodi()[i])
        for i in range(0, len(r2.get_nodi())):
            if r2.get_nodi()[i].get_id() != autista.get_id():
                route.aggungi_nodo(r2.get_nodi()[i])
    route.set_km()
    return route

```

Crea una route con l'autista in prima posizione e poi tutti gli utenti delle due route + UniFe..

# Merge incomplete routes



# Visualizziamo i risultati



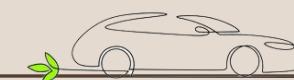
```
routes_merge1=mer.merge_single_nodes(routes2, solo_routes)  
for el in routes_merge1:  
    print(el)
```

```
gf.grafico(lista1, lista2, routes_merge1, 'figura2.png')
```

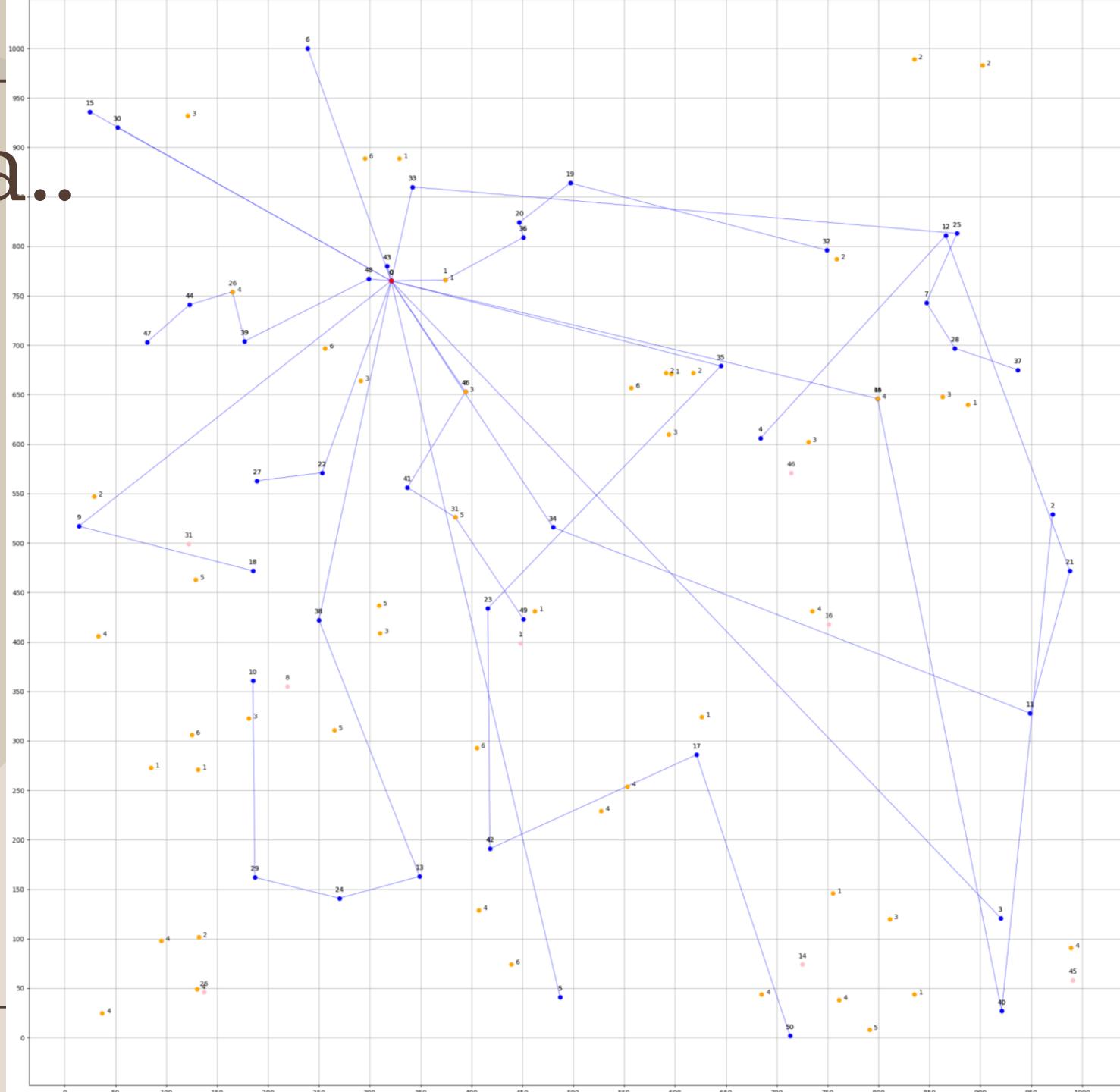
```
routes_merge2= mer.merge_incomplete_routes(routes_merge1)  
for el in routes_merge2:  
    print(el)
```

```
gf.grafico(lista1, lista2, routes_merge2, 'figura3.png')
```

Grafichiamo i risultati della merge  
incomplete routes



# Prima..

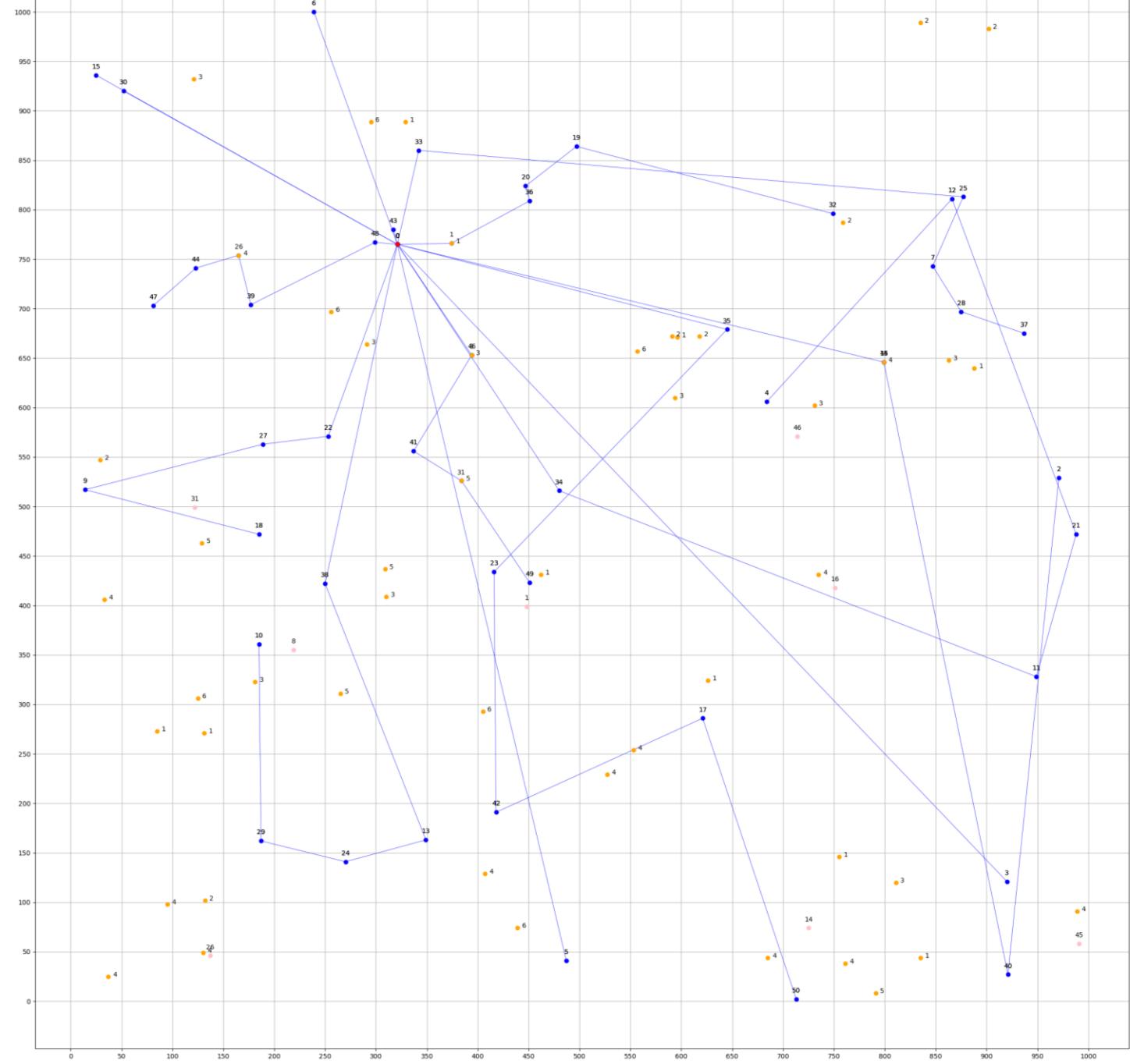


Merge single nodes

totale km  
merge single nodes:  
11071.467509599799



# Dopo..

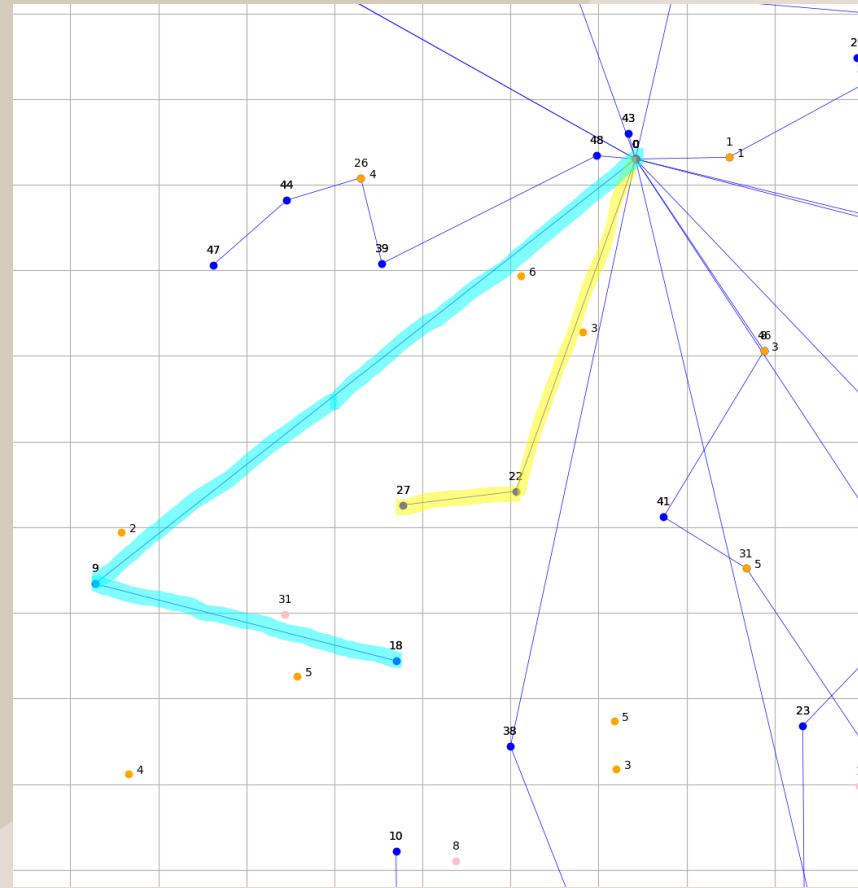


Merge incomplete routes

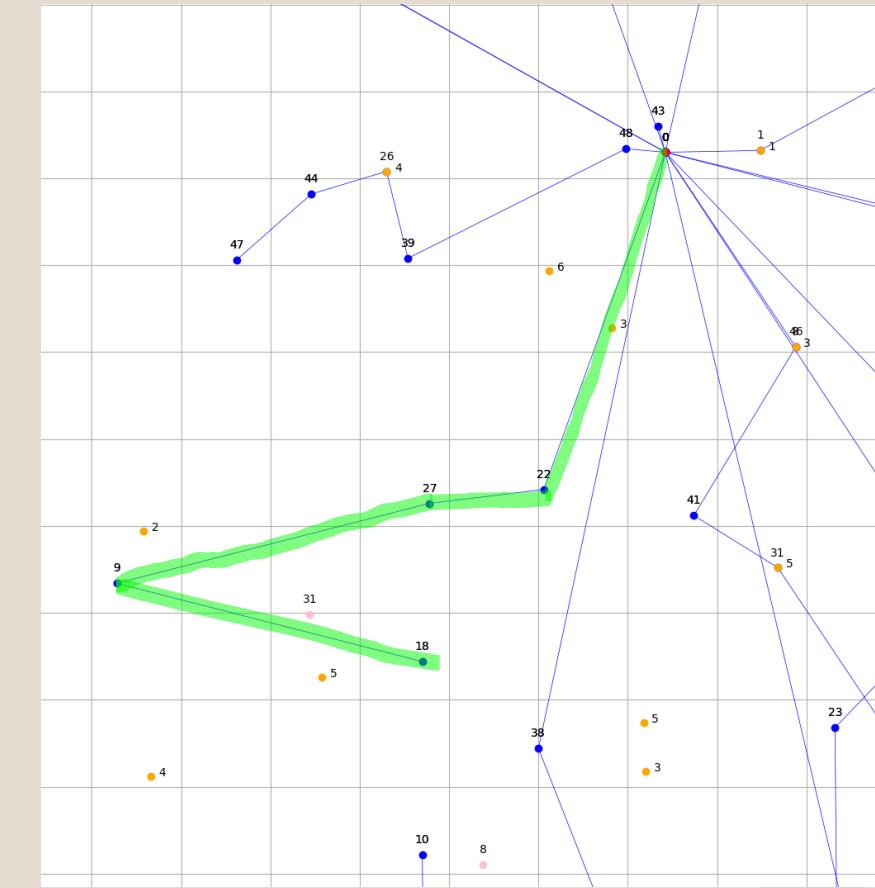
totale km  
incomplete routes:  
**10857.75670653187**



Merge single nodes



Merge incomplete routes





Cheapest insertion

Merge single nodes

Merge incomplete routes

Variable neighborhood search

Local search:  
Intra-Route

Local search:  
Intra-Route

Local search:  
Inter-Route

Local search:  
Intra-Route



Variable  
Neighborhood  
Descend

Variable  
Neighborhood  
Descend

Variable  
Neighborhood  
Descend

# Variable Neighborhood Descend

La metaeuristica opera in 3 intorni di ricerca N1, N2 ed N3 associati a tre diverse “mosse” applicabili. Nel nostro caso N1=2-OPT, N2=Node Swap ed N3=Chain Relocation.

- L'euristica compie un'esplorazione e se trova una mossa che migliora la soluzione corrente si sposta, altrimenti passa all'intorno successivo.
- Ogni volta che viene trovata una mossa “migliorante” si riparte dall'esplorazione dell'intorno N1.
- La procedura termina quando non vengono più trovate mosse “miglioranti”. Dunque termina su un ottimo locale di tutti gli intorni usati.



```
Data una soluzione iniziale  
Begin  
do  
{cerca una mossa migliorante in N1  
if mossa trovata  
    then applica la mossa in N1  
else  
    begin  
    { cerca una mossa migliorante in N2  
    if mossa trovata  
        then applica la mossa in N2  
    else  
        begin  
        { cerca una mossa migliorante in N3  
        if mossa trovata  
            then applica la mossa in N3  
        end }  
    end }  
} }  
while una mossa viene eseguita  
end
```

# Variable Neighborhood Descend



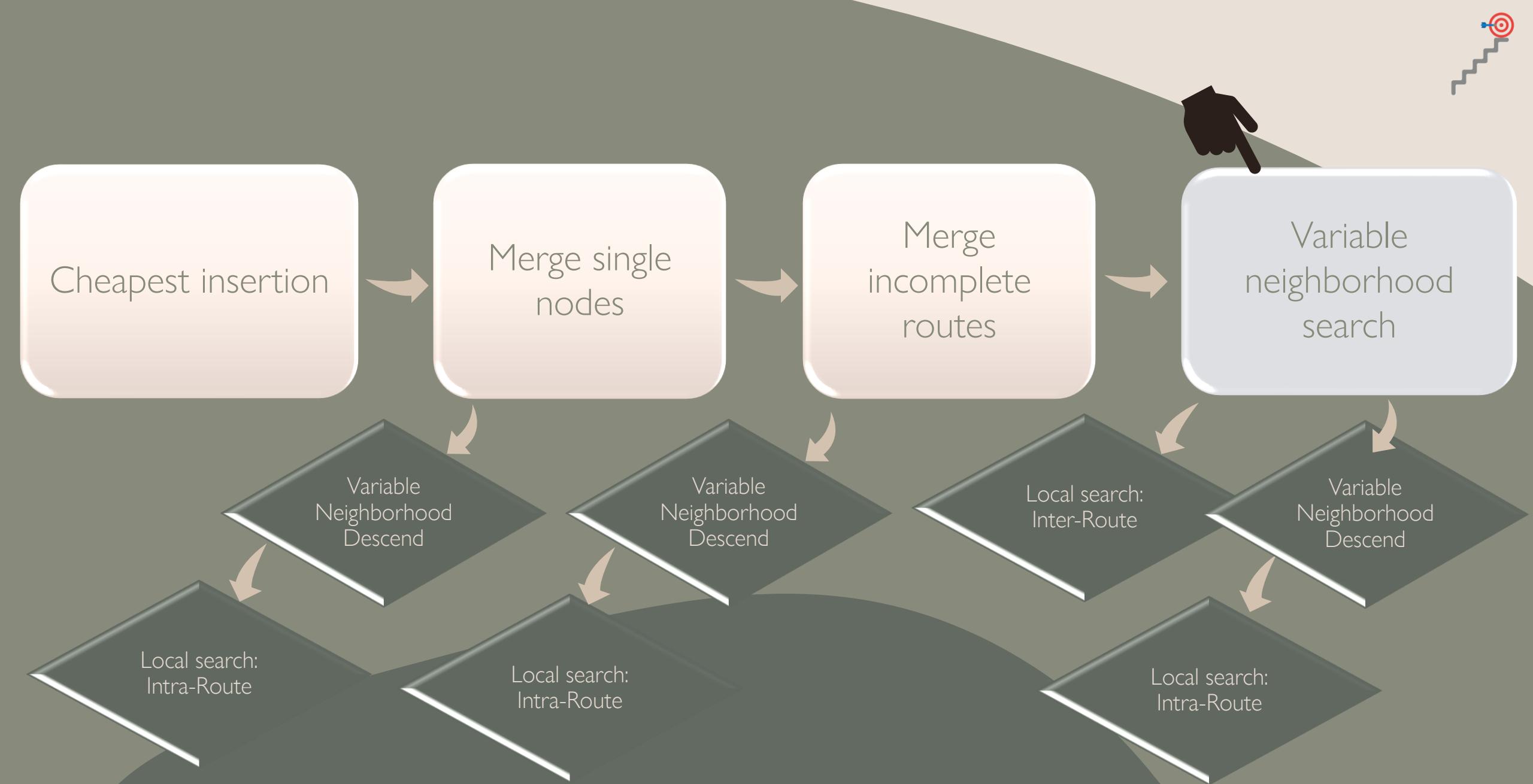
```
def variable_neighborhood_descend(init_sol):  
    intorno = []  
    current_solution = rt.Route()  
    for j in range(0, len(init_sol.get_nodi())):  
        current_solution.aggiungi_nodo(init_sol.get_nodi()[j])  
    current_solution.set_km()  
    number_of_neighbor=0
```

```
while number_of_neighbor < len(conf.intra_route):  
    intorno=conf.intra_route[number_of_neighbor](current_solution)  
    for j in range(0, len(intorno)):  
        intorno[j].set_km()  
    intorno.sort(key=attrgetter('km'))  
    if intorno[0].get_km() >= current_solution.get_km():  
        number_of_neighbor = number_of_neighbor+1  
    else:  
        current_solution.get_nodi_rif().clear()  
        for j in range(0, len(intorno[0].get_nodi())):  
            current_solution.aggiungi_nodo(intorno[0].get_nodi()[j])  
        current_solution.set_km()  
        number_of_neighbor=0  
return current_solution
```

Partendo da una route iniziale applico una delle euristiche intra-route. Ottengo così tante soluzioni (route) possibili.

Prendendo in considerazione la soluzione che mi minimizza i km, se questa è migliore della soluzione corrente aggiorno la soluzione corrente. Altrimenti passo ad applicare l'euristica intra-route successiva.

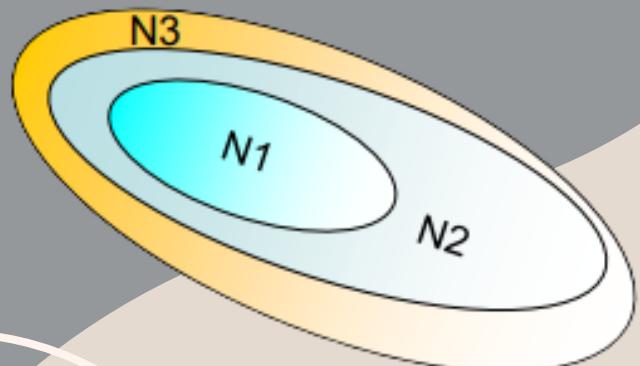




# Variable Neighborhood Search



1. **Shaking:** Si genera un punto  $x'$  a caso nel  $k$ -th intorno (con  $k$  indice di una funzione inter-route) di  $x$  (cioè  $x' \in N_k(x)$ ).
2. **Local search:** si applica una local search (nota: può essere l'algoritmo di VND) a partire da  $x'$  ottenendo così un ottimo locale  $x''$ .
3. **Move or not:** se tale ottimo locale  $x''$  è meglio della soluzione candidata corrente  $x$ , ci si sposta, cioè,  $x := x''$ , e il procedimento ricomincia riassegnando  $k := 1$ . Altrimenti si passa all'intorno successivo impostando  $k := k+1$



Ingredienti:

- Kmax, che è il numero di funzioni inter-route che decidiamo di applicare. Nel nostro caso  $kmax=2$  in quanto le nostre strutture sono la String Exchange e la String Relocation.
- una soluzione iniziale ammissibile  $x$
- una condizione di stop (quando ho raggiunto un numero massimo di iterazioni).

While not condizione\_di\_stop do  
 $k:=1$ ,

Until  $k=kmax$  repeat

(Shaking)  $x' :=$  random in  $N_k(x_k)$

(Optimize Locally)  $x'' :=$  Local Search ( $x', N_k$ )

(Move or not) if ( $c_{x''} < c_x$ ) then  $x := x''$ ,  $k := 1$

else  $k := k+1$



# Variable Neighborhood Search



```
def vnd(r1, r2):
    routes=[]
    routes.append(variable_neighborhood_descend(r1))
    routes.append(variable_neighborhood_descend(r2))
    return routes
```

Ottimizzo le route tramite la VND.

Partendo da due route iniziali applico una delle euristiche inter-route. Ottengo così tante soluzioni (route) possibili.



```
def variable_neighborhood_search(r1, r2):
    iter=0
    max_iterations=5
    routes=[]
    r=[]
    r.append(r1)
    r.append(r2)
    routes_optimized=[]
    routes_optimized.append(r)
    dim_sol_trovata=0
    route1=r1
    route2=r2
    solution=[]

    while True:
        number_of_neighbor=0
        while number_of_neighbor < len(conf.inter_route):
            funzione=conf.inter_route[number_of_neighbor]
            routes=funzione(route1, route2)
            if len(routes)==0:
                break
            if(len(routes)==1):
                route1primo=routes[0][0]
                route2primo=routes[0][1]
            else:
                shake= ra.randint(0, len(routes)-1)
                route1primo=routes[shake][0]
                route2primo=routes[shake][1]
            routes.clear()

            Riapplico la stessa euristica inter-route usata prima alle due route ottenute.

            Per ogni coppia di route ottenuta chiamo la vnd per ottizzarle.
```

```
routes=funzione(route1primo, route2primo)
for i in range(0, len(routes)):
    routes_optimized.append(vnd(routes[i][0], routes[i][1]))
index=0
best_km=ut.kmtot(routes_optimized[0])
for j in range(1, len(routes_optimized)):
    if ut.kmtot(routes_optimized[j])<best_km:
        index=j
        best_km=ut.kmtot(routes_optimized[j])
if index==0:
    number_of_neighbor=number_of_neighbor+1
else:
    number_of_neighbor=0

Se non ho trovato una soluzione migliore rispetto alle due route r1 e r2 iniziali allora provo con una altra euristica inter-route.

route1(routes_optimized[index][0])
route2(routes_optimized[index][1])
routes_optimized.clear()
r.clear()
r.append(route1)
r.append(route2)
routes_optimized.append(r)

Altrimenti ricomincio applicando tutte le euristiche a partire dalla prima, sulla soluzione migliorata.

iter=iter+1

if iter >= max_iterations:
    break

solution.append(route1)
solution.append(route2)
return solution
```

# Visualizziamo i risultati

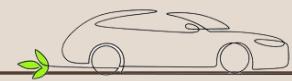


```
for i in range(0, len(lista_finale)):  
    for k in range(i+1, len(lista_finale)):  
        temp=meta.variable_neighborhood_search(lista_finale[i], lista_finale[k])  
        if(ut.kmtot(temp)<(lista_finale[i].get_km()+lista_finale[k].get_km())):  
            lista_finale[i]=temp[0]  
            lista_finale[k]=temp[1]  
  
for el in lista_finale:  
    print(str(el))  
  
gf.grafico(lista1, lista2, lista_finale, 'figura4.png')
```

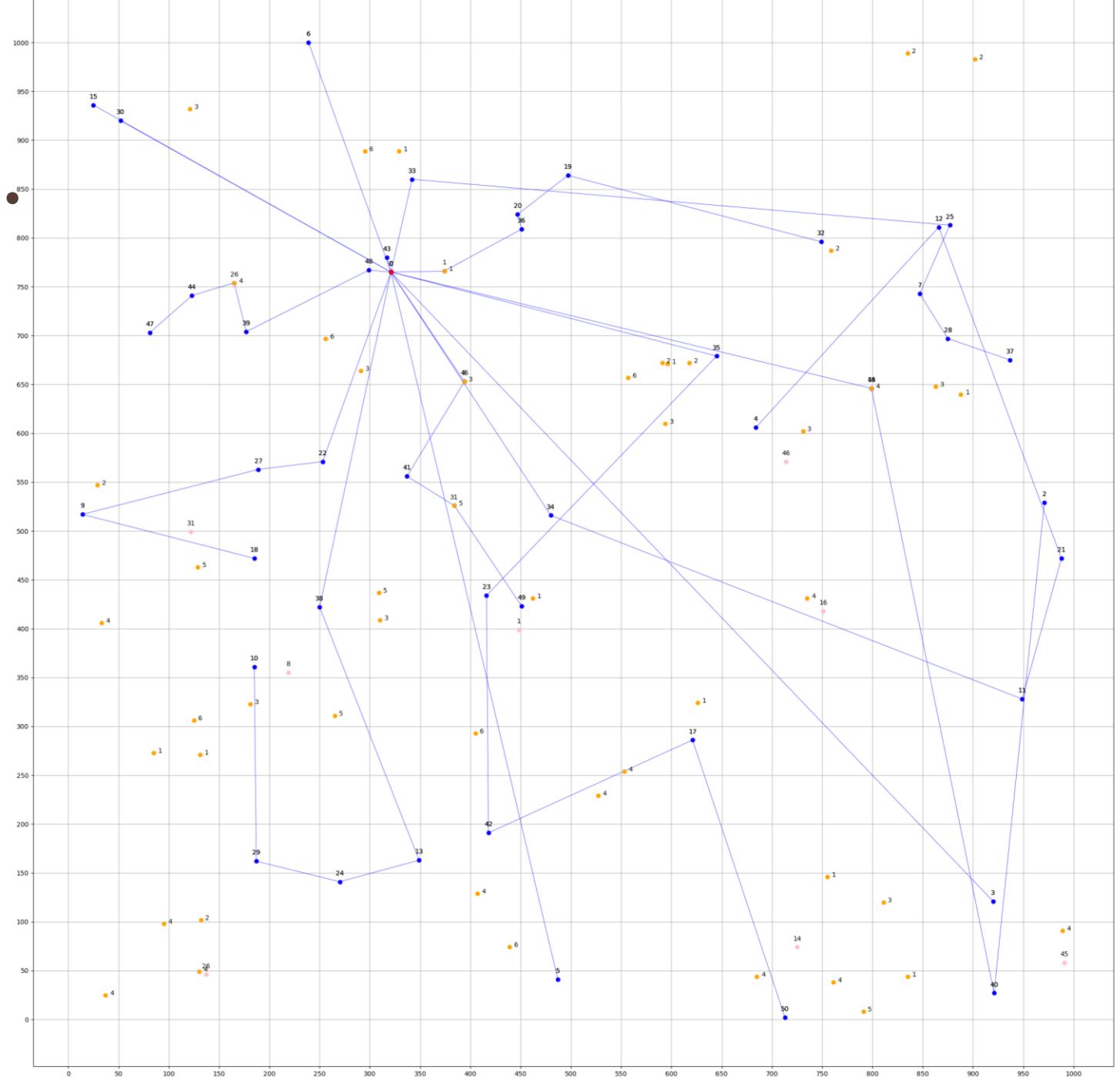
Per ogni route i e ogni sua route  
successiva k chiamo la VNS

Se i km delle due route trovate dalla VNS  
hanno km minori dei km delle route i e j  
sommati allora sostituisco le route i e j con  
quelle ottenute dalla VNS

Grafichiamo i risultati della VNS



# Prima..

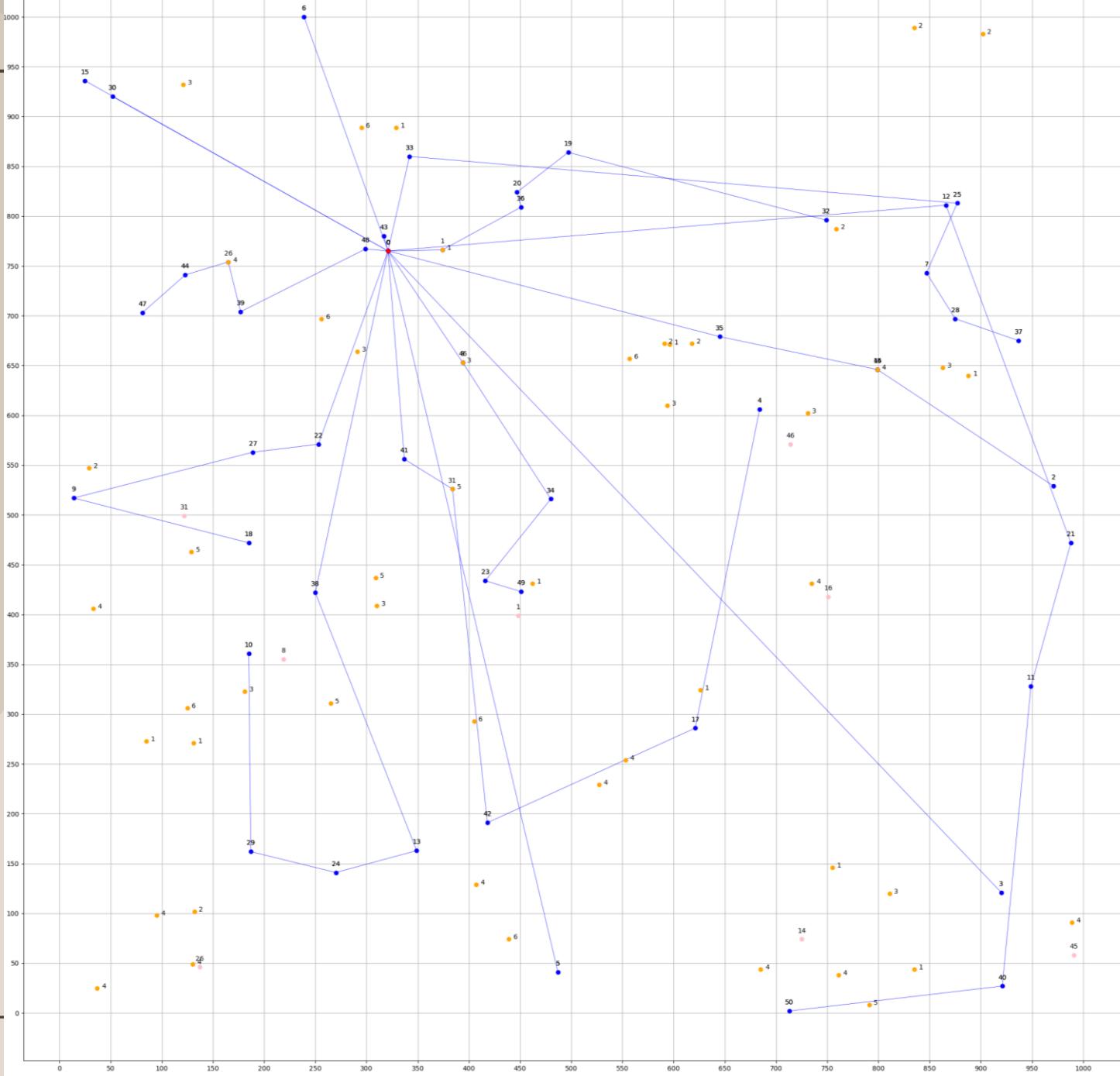


Merge incomplete routes

totale km  
incomplete routes:  
10857.75670653187



# Dopo..



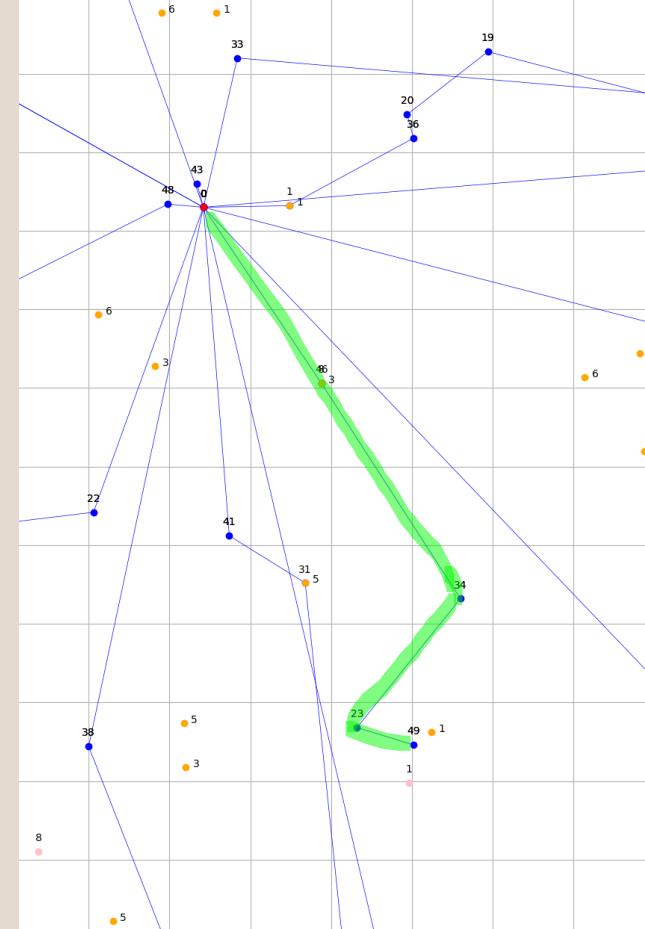
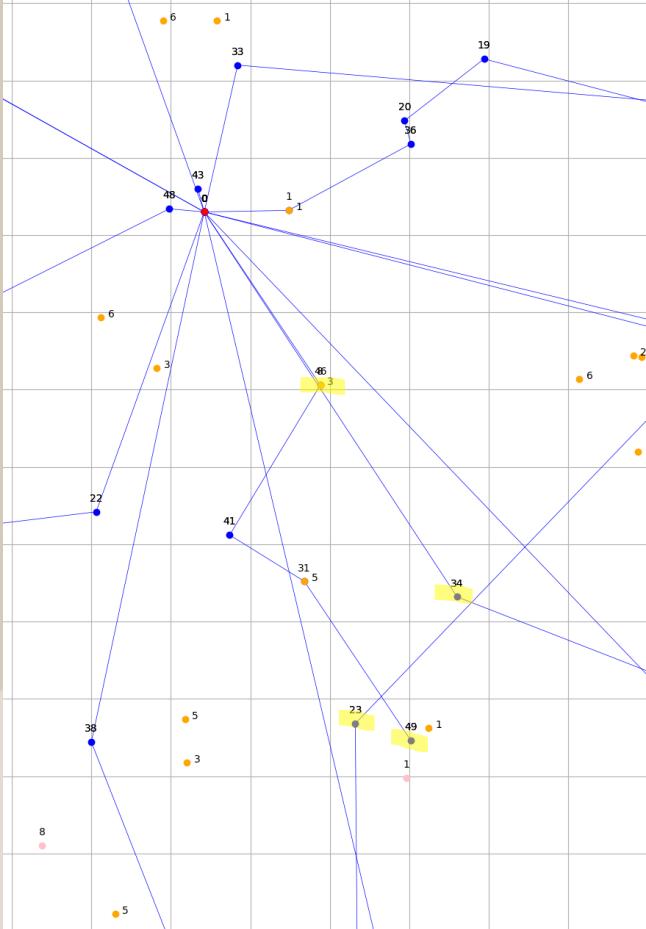
Variable  
Neighborhood  
Search

totale km variable  
neighborhood search:  
**9640.447200380258**

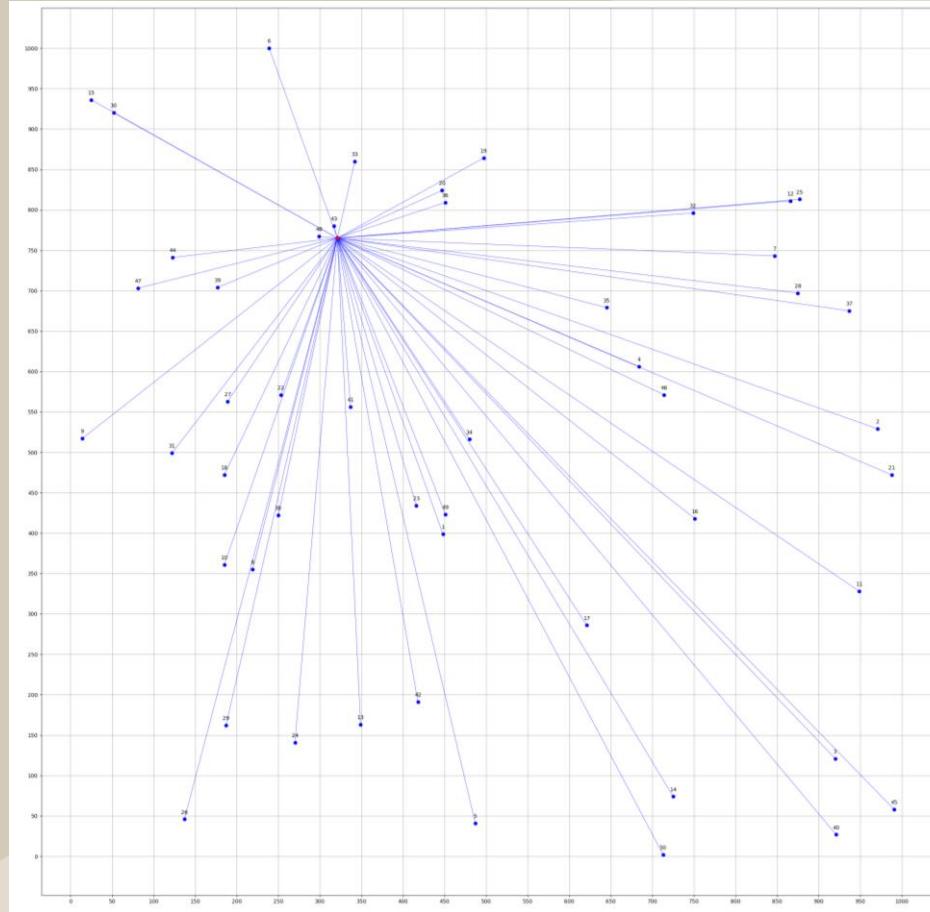


# Variable Neighborhood Search

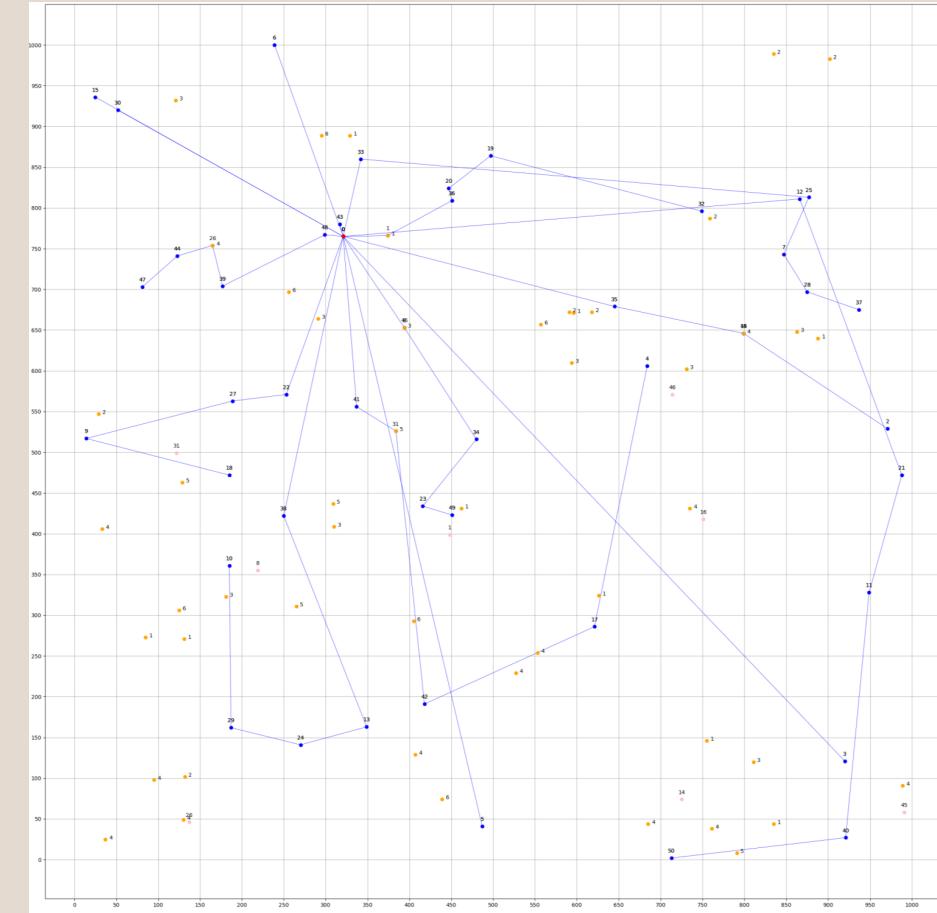
Merge incomplete routes



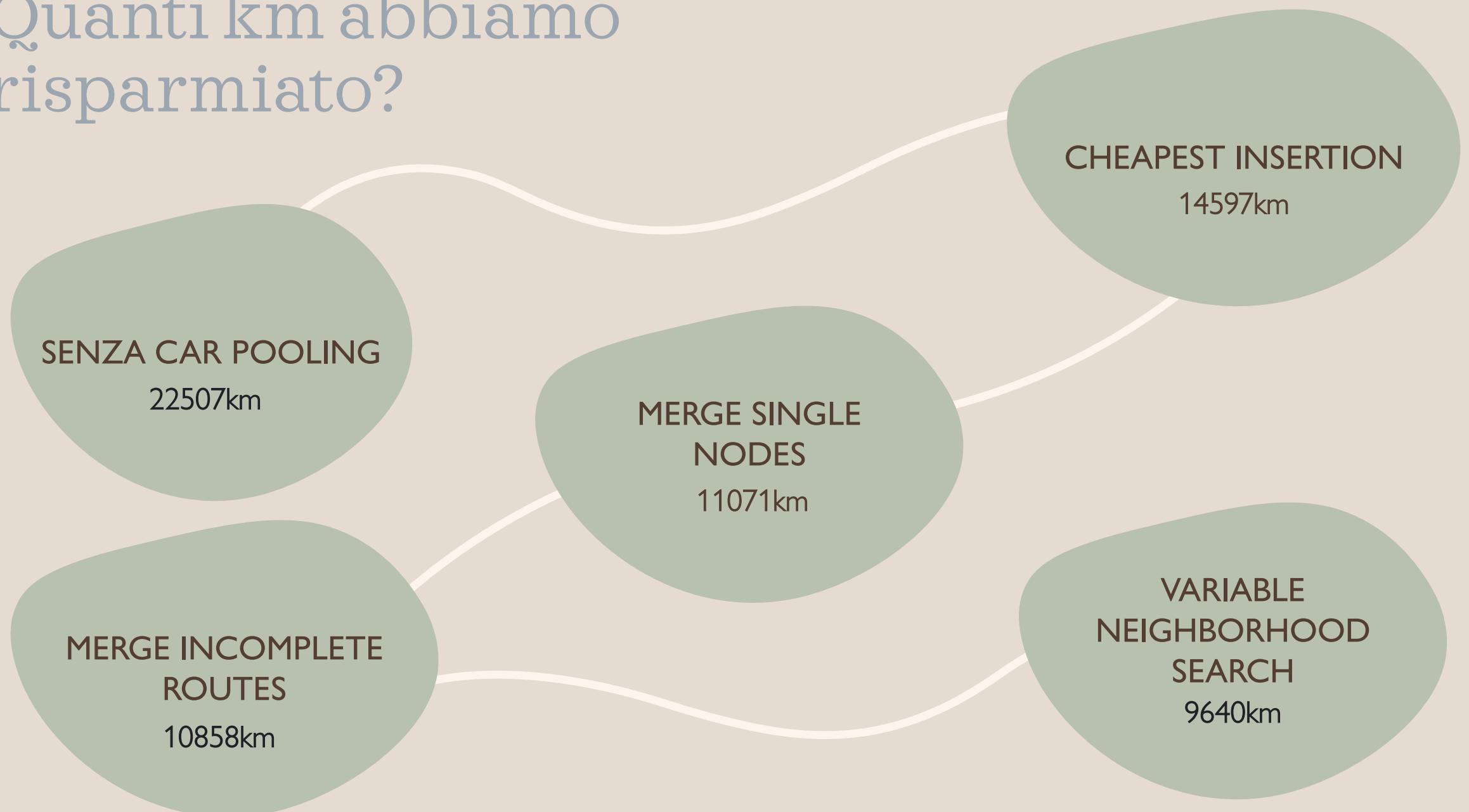
## Senza Car Pooling



## Soluzione finale



# Quanti km abbiamo risparmiato?



# Il car pooling in Italia:

Carpooling aziendale in Italia:  
635.000 euro risparmiati e 413  
tonnellate di CO<sub>2</sub> evitate nel  
2022





# Grazie per l'attenzione!

Margherita Rizzieri, Matteo Brina, Rachele Rubini