UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Vehicle Routing Problem with Time Windows

Elaborato di Algoritmi di Ottimizzazione Combinatoria e su Rete

Davide Mancinelli matr. M63001674

Angelo Caliolo matr. M63001623

Indice

Capitolo 1

Introduzione

Il Vehicle Routing with Time Windows (VRPTW) è una variante del Vehicle Routing Problem (VRP), in cui ogni flotta di veicoli con capacità limitata deve prelevare o consegnare merci (o persone) in differenti punti, ognuno dei quali accessibile solo in determinati intervalli di tempo. Quindi, a differenza del VRP standard, bisogna rispettare anche vincoli temporali per le consegne.

1.1 Assunzioni del problema

Prima di formalizzare il problema, è necessario elencare le seguenti assunzioni:

- E' presente un nodo deposito, da cui la flotta di veicoli inizia e termina il suo percorso
- Sono presenti dei nodi (chiamati punti vendita) in cui almeno un veicolo della flotta deve passare
- Ogni punto vendita deve essere transitato esclusivamente una volta
- Ogni punto vendita presenta un orario di apertura (prima del quale non è possibile ricevere consegne) e un orario di chiusura (oltre il quale non è possibile ricevere consegne)

- Nel caso in cui un veicolo arrivasse in un punto vendita prima dell'orario di apertura, tale veicolo deve attendere l'apertura del punto vendita
- Tutti i punti vendita devono essere serviti entro l'orario di chiusura
- L'obiettivo è minimizzare i costi di trasporto

1.2 Approcci per la risoluzione

Ci sono differenti approcci per la risoluzione del VRPTW; di seguito sono presentati quelli affrontati in questo elaborato:

- Algoritmi esatto: [CONTINUA]
- Algotimo euristico: [CONTINUA]
- Algoritmo TabuSearch: [CONTINUA]

Capitolo 2

Definizione del problema

Il VRPWT può essere definito da:

- Flotta **V** di veicoli; con m=|V|
- Insieme **N** di nodi; tale insieme è costituito a sua volta da:
  - Insieme **P={1,...,p}** di punti vendita
  - Deposito **D={s,t}**; rappresentato da 2 nodi distinti: s come nodo di partenza, t come nodo di destinazione

  E' possibile quindi definire n=|N|=|P|+2, con N={s,1,...,p,t}
- Insieme **A** di archi orientati (i,j), dove $i \neq j$
- Grafo **G(V,N)** orientato e pieno

Altri elementi da prendere in considerazione sono:

- Il costo c_{ij} associato a ogni arco (i,j)
- Il tempo t_{ij} di percorrenza associato a ogni arco (i,j)
- La portata massima Q_k di ogni ogni veicolo k
- La richiesta d_i di merce da soddisfare per ogni punto vendita i
- La finestra di tempo $f_i = [a_i, b_i]$ in cui ogni punto vendita è aperto

- L'insieme di cluster $C = \{C_1, ..., C_m\}$; è possibile dividere l'insieme N in sottoinsiemi cluster C_k, ognuno associato al veicolo k

- Il tempo di servizio s_{ik} sul nodo i; in altre parole, è il tempo da attendere affinchè il nodo i sia totalmente servito. Ipotizziamo $s_{sk} = 0, s_{tk} = 0 \quad \forall k \in V$

2.1 Funzione obiettivo

L'obiettivo è quello di minimizzare i costi di ogni veicolo per servire tutti i punti vendita negli orari adeguati. Possiamo scrivere la funzione obiettivo come segue

$$min \sum_{k \in V} \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ijk}$$

2.2 Variabili di decisione

Definiamo quindi le seguenti variabili di decisione:

- $x_{ijk} = 1$ se l'arco (i,j) è percorso dal veicolo k, 0 altrimenti

- Il tempo di arrivo l_{ik} del veicolo k al nodo i. Ipotizziamo $l_{sk} = 0 \quad \forall k \in V$, con $s \in D$

2.3 Vincoli

Definiamo i seguenti vincoli:

- Vincolo di capacità; la somma di tutte le richieste soddisfatte non deve superare la capacità massima del singolo veicolo

$$\sum_{i \in N} \sum_{j \in N} d_i x_{ijk} \leq Q_k \qquad \forall k \in V$$

- Vincoli di routing
  - Vincolo di unicità dei punti vendità; ogni punto vendita i deve essere visitato una sola volta

$$\sum_{k \in V} \sum_{j \in N} x_{ijk} = 1 \qquad \forall i \in P$$

  - Vincolo sulla partenza del singolo veicolo; controlla che il veicolo sia partito dal deposito

$$\sum_{j \in P} x_{sjk} = 1 \quad \forall k \in V$$

    In alternativa, nel caso in cui decidessimo che non è necessario che partano tutti i veicoli, basta imporre tale sommatoria ≤ 1
  - Vincolo sull'arrivo del singolo veicolo; controlla che il veicolo sia arrivato dal deposito

$$\sum_{i \in P} x_{itk} = 1 \quad \forall k \in V$$

    In alternativa, nel caso in cui decidessimo che non è necessario che partano tutti i veicoli, basta imporre tale sommatoria ≤ 1
  - Vincolo di coerenza partenza-arrivo; tutti i veicoli che sono partiti, devono essere tornati

$$\sum_{j \in P} x_{sjk} = \sum_{i \in P} x_{itk} \quad \forall k \in V$$

  - Vincolo sul flusso dei punti vendita; una volta che un veicolo è entrato in un punto vendita, deve anche uscirne. In

alternativa, possiamo dire che la somma dei nodi entrati in un punto vendita deve essere uguale a quella dei nodi uscenti

$$\sum_{i \in N} x_{ijk} = \sum_{i \in N} x_{jik} \quad \forall j \in P,\ \forall k \in V$$

Bisogna tenere conto del fatto che $j \in P$ e non N, poichè i nodi s e t (compatibili con il deposito) possono anche presentare solo nodi entranti o solo nodi uscenti

- Vincolo sui sottogiri; si vogliono evitare cicli, facendo in modo che, per ogni sottoinsieme di P, la somma degli archi con origine e destinazione in questo sottoinsieme non può superare la cardinalità di S-1, altrimenti si avrebbero cicli

$$\sum_{i,j \in S} x_{ijk} \leq |S| - 1 \quad S \subset P \quad |S| \geq 2 \quad \forall k \in V$$

Un esempio applicativo di questo vincolo è il seguente. Sia $P = 2, 3, 4, 5$ ed $S = 2, 3, 4$; il vincolo imporrebbe $\sum_{i,j \in S} x_{ijk} \leq 3 - 1 = 2$. Di conseguenza, se il percorso del veicolo fosse $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$, la somma degli archi sarebbe pari a 3, violando il vincolo

- Vincoli temporali
  - Vincolo di inizio servizio; il tempo di inizio servizio l_{jk} deve essere almeno pari all'istante l_{ik} in cui si è attivati nel generico punti i, più il tempo per servire quel nodo, più il tempo per spostarsi lungo l'arco (i,j)

    $$l_{jk} \geq (l_{ik} + s_{ik} + t_{ij}) - (1 - x_{ijk})M \quad \forall i, j \in N \quad \forall k \in V$$

    con M costante sufficientemente grande. Tale costante viene aggiunta affinchè siano esclusi tutti gli archi non transitati
  - Vincolo sulla finestra temporale

    $$a_i \leq l_i \leq b_i$$

- Vincoli di dominio

- $x_{ijk} \in \{0, 1\}$
- $i, j \in N$
- $k \in V$
- $l_{ik} \in R^+ \cup \{0\}$
- $s_{ik} \in R^+ \cup \{0\}$

Capitolo 3

Risoluzione tramite algoritmo esatto

Un problema VRPTW può essere risolto tramite un algoritmo esatto. Nel nostro caso è stato utilizzato **PuLP**, un modellatore scritto in Python

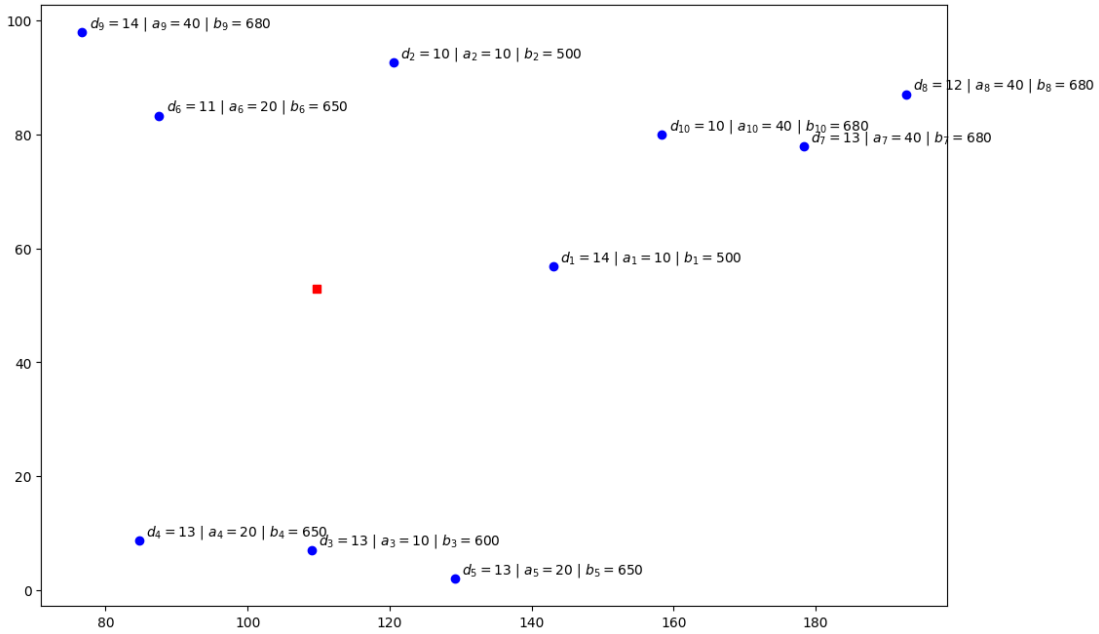
3.1 Creazione dei dati

I dati sono stati scritti secondo la seguente logica:

- Il numero dei punti vendita è stato scelto arbitrariamente da noi
- Le coordinate x e y di ogni punto vendita sono state scelte casualmente

```
n = 10
xc = rnd.rand(n+1)*200
xc = np.append(xc, xc[0])
yc = rnd.rand(n+1)*100
yc = np.append(yc, yc[0])
```

- L'insieme P dei punti vendita è un vettore che va da 1 a n
- L'insieme N dei nodi è costituito da P, preceduto da 0 e succeduto da n+1



- L'insieme A degli archi è un vettore dato dalle combinazione dei vari nodi
- I costi associati a ogni arco sono stati scelti casualmente
- Il tempo di percorrenza di ogni arco è pari alla distanza tra i nodi associati a ogni arco
- La richiesta di Ogni nodo è scelta casualmente
- Le finestre temporali di ogni nodo sono state scelte arbitrariamente da noi
- L'insieme Q della capacità è stato scelto arbitrariamente da noi
- I tempi di servizio di ogni nodo sono stati scelti casualmente

```
P = [il for il in range(1,n+1)]
N = [0] + P + [n+1]

A = [(i,j) for i in N for j in N if i!=j and
    (i,j)!=(n+1,0) and (i,j)!=(0,n+1)]

c = {(i,j): np.random.randint(2,15) for i,j
   in A}
t = {(i,j): int(np.hypot(xc[i]-xc[j], yc[i]-
   yc[j])) for i,j in A}
```

```
np.random.seed(0)
d = {i: np.random.randint(10,15) for i in P}
d[0] = 0
d[n+1] = 0

a = {0:0, 1:10, 2:10, 3:10, 4:20, 5:20,
   6:20, 7:40, 8:40, 9:40, 10:40, 11:0}
   # Orario di apertura
b = {0:200, 1:500, 2:500, 3:600, 4:650,
   5:650, 6:650, 7:680, 8:680, 9:680, 10:680,
    11:700} # Orario di chiusura

V = [1,2,3,4]
Q = {1: 50, 2:50, 3:25, 4:25}

s = {(i,k): np.random.randint(3,5) if i!=0
   and i!=n+1 else 0 for i in N for k in V}
```

3.2 Utilizzo di PuLP

Per prima cosa creiamo il modello tramite la libreria PuLP come segue:

```
model = pulp.LpProblem("VRPTW", pulp.LpMinimize)
```

3.2.1 Variabili di decisione

Si aggiungono le variabili di decisione:

```
arco_var = [(i,j,k) for i in N for j in N for k in V
   if i!=j and (i,j)!=(n+1,0) and (i,j)!=(0,n+1)]
arco_arrivo = [(i,k)for i in N for k in V]
x = pulp.LpVariable.dicts("x", arco_var, cat="Binary"
   )
l = pulp.LpVariable.dicts("x", arco_arrivo, lowBound
   = 0, cat="Continuous")
```

3.2.2 Funzione obiettivo

Si aggiuge la funzione obiettivo:

```
model += pulp.lpSum(c[i,j]*x[i,j,k] for (i,j,k) in
    arco_var)
```

In generale, è possibile anche cambiare la funzione obiettivo, ponendo per esempio all'interno della sommatoria il tempo di percorrenza invece dei costi

```
model += pulp.lpSum(t[i,j]*x[i,j,k] for (i,j,k) in
    arco_var)
```

3.2.3 Vincoli

Si aggiungono i vincoli precedentemente definiti:

```
# Vincolo di capacita'
for k in V:
  model += pulp.lpSum(d[i]*x[i,j,k] for i in P for j
     in N if i!=j) <= Q[k]

# Vincolo di unicita' dei punti di vendita
for i in P:
  model += pulp.lpSum(x[i,j,k] for j in N for k in V
     if (i,j) if i!=j) == 1

# Vincolo sulla partenza e arrivo del singolo veicolo
for k in V:
  model += pulp.lpSum(x[0,j,k] for j in P) == 1
  model += pulp.lpSum(x[i,n+1,k] for i in P) == 1

# Vincolo di coerenze partenza-arrivo
for k in V:
  model += pulp.lpSum(x[0,j,k] for j in P) == pulp.
     lpSum(x[i,n+1,k] for i in P)

# Vincolo sul flusso dei punti vendita
for j in P:
  for k in V:
    model += pulp.lpSum(x[i,j,k] for i in N if i!=j)
       == pulp.lpSum(x[j,i,k] for i in N if i!=j)

# Vincolo sui sottogiri
```

```
    for r in range(2, len(P)+1):
      for S in itertools.combinations(P,r):
        for k in V:
          model += pulp.lpSum(x[i,j,k] for i in S for j
             in S if i!=j and (i,j)!=(n+1,0) and (i,j)
             !=(0,n+1)) <= len(S)-1

    # Vincolo di inizio servizio
    M = 10000000
    for k in V:
      for j in N:
        for i in N:
          if i!=j and (i,j)!=(n+1,0) and (i,j)!=(0,n+1):
            model += l[j,k] >= (l[i,k]+s[i,k]+t[i,j]) - M
               *(1-x[i,j,k])


    # Vincolo sulla finestra temporale
    for (i, k) in arco_arrivo:
        model += l[i,k] >= a[i]
        model += l[i,k] <= b[i]

    # Inizializzazione del deposito
    for k in V:
      l[0,k] = a[0]
```

3.2.4 Esecuzione del modello

Possiamo quindi eseguire il modello:

```
    model.solve()
    print("Costo: ", pulp.value(model.objective))
```

Capitolo 4

Risoluzione tramite algoritmo euristico

Un altro approccio che può essere utilizzato per la risoluzione del VRPTW può essere l'utilizzo di un algoritmo di tipo euristico (precisamente greedy): **l'algoritmo di Clarke and Wright**. Per quanto riguarda la creazione dei dati, abbiamo utilizzato lo stesso codice presente nell'approccio tramite algoritmo esatto.

4.1 Breve descrizione dell'algoritmo

L'algoritmo di Clarke and Wright si basa sul concetto di saving, traducibile con risparmio.
Tale algoritmo definisce di volta in volta delle rotte (chiamate Tour). Inizialmente ogni rotta è costituita da un unico punto vendita, successivamente si prova a unire ogni coppia di rotte in presenza di un eventuale risparmio. La soluzione finale sarà costituita da 4 rotte, una per ogni veicolo. Ovviamente, devono essere anche fatti controlli per quanto riguarda il rispetto di vincoli, tra i quali quello di capacità, quelli temporali e del numero di giri per ogni veicolo.

4.2 Spiegazione del codice

Prima di spiegare i passi principali dell'algoritmo, bisogna fare una breve introduzione a ogni funzione utilizzata.

4.2.1 Verifica della capacità

Questa funzione ha il compito di verificare che sia rispettato il vincolo di capacità di ogni veicolo associato a una determinata rotta. Viene calcolata la domanda totale dei punti vendita appartenenti a una potenziale rotta, per poi essere confrontata con la capacità del veicolo:

```
def verifica_capacita (rotta, d, capacita):
    domanda_totale = 0
    for i in rotta:
        domanda_totale += d[i]
    return domanda_totale <= capacita
```

4.2.2 Verifica della finestra temporale

In questa funzione si vuole controllare che siano rispettato gli orari di apertura e di chiusura dei punti vendita.
Per ogni punto vendita della rotta calcoliamo il tempo corrente come somma del tempo impiegato fino a quel momento e il tempo di percorrenza dell'ultimo arco.
Controlliamo se tale tempo corrente sia minore dell'orario di apertura del punto vendita, in tal caso attendiamo l'orario di apertura.
Controlliamo quindi che il tempo corrente non sia superiore all'orario di chiusura, nel caso escludiamo il nodo.
Infine controlliamo che il veicolo sia tornato al deposito.

```
def verifica_finestra (rotta, t, s, a, b, veicolo_id):
    tempo_corrente = 0
    nodo_precedente = 0                    # Deposito di
        partenza

    for i in rotta:
        tempo_corrente += t[nodo_precedente, i]

        # Nel caso in cui arrivo prima dell'apertura,
           aspetto
```

```
        if tempo_corrente < a[i]:
            tempo_corrente = a[i]

        # Nel caso in cui arrivo dopo la chiusura, n[U+FFFD]
            rispettato il vincolo
        if tempo_corrente > b[i]:
            return False

        # Aggiunta del tempo di servizio
        print(s[i, veicolo_id])
        tempo_corrente += s[i, veicolo_id]
        nodo_precedente = i

    # Verifico se e' ritornato al deposito t
    if (nodo_precedente, n+1) in t:
        tempo_corrente += t[nodo_precedente, n+1]
        if tempo_corrente > b[n+1]:
            return False

    return True
```

4.2.3 Verifica di un nodo estremo

In questa funzione si vuole trovare la rotta che contiene un nodo estremo (iniziale o finale) e il suo eventuale indice. Scorrendo tra i punti vendita di ogni rotta attiva, si controlla se tale punto vendita sia un estremo. Nel caso in cui lo sia, viene restituita la rotta e il suo indice

```
    def trova_rotta_contenente (i, rotte_attive):
        # Scorro tra i nodi delle rotte attive
        for r_attiva in rotte_attive:
            if i in r_attiva:
                indice = r_attiva.index(i)
                # Si controlla che sia un estremo
                if indice==0 or indice==len(r_attiva)-1:
                    return r_attiva, indice
        return None, -1
```

4.2.4 Unione delle rotte

Questa funzione ha il compito di unire due rotte. Per fare ciò, si fanno vari controlli per validare un'eventuale unione tra rotte. In generale,

un'unione è validata se i punti vendita considerati sono estremi di tali rotte

```
def unisci_rotte (rotta_i, rotta_j, indice_i, indice_j):

    # Sono effettuati vari controlli

    # i e' l'ultimo di rotta_i, mentre j e' il primo di
       rotta_j
    if indice_i==len(rotta_i)-1 and indice_j==0:
        return rotta_i + rotta_j
    # j e' l'ultimo di rotta_j, mentre i e' il primo di
       rotta_i
    elif indice_j==len(rotta_j)-1 and indice_i==0:
        return rotta_j + rotta_i
    # i e' l'ultimo di rotta_i, mentre j e' l'ultimo di
       rotta_j
    elif indice_i==len(rotta_i)-1 and indice_j==len(
       rotta_j)-1:
        return rotta_i + rotta_j[::-1]
    # i e' il primo di rotta_i, mentre j e' il primo di
       rotta_j
    elif indice_i==0 and indice_j==0:
        return rotta_i[::-1] + rotta_j

    return None
```

4.2.5 Conteggio delle rotte

Questa funzione deve contare le rotte per ognuno dei veicoli.

```
def conta_rotte_per_veicolo(assegnazione, V):
    conteggio = {k: 0 for k in V}
    for rotta, veicolo in assegnazione.items():
        conteggio[veicolo] += 1
    return conteggio
```

4.2.6 Flusso principale

Il flusso principale del codice può essere diviso in differnti step:

1. Inizializziamo ogni punto vendita come rotta separata, successivamente distribuiamo i nodi tra i veicoli disponibili secondo una

logica Round-Robin, in modo tale da bilanciare i punti vendita assegnati a ogni veicolo

```
    # Inizializzazione di ogni punto vendita come
       rotta separata
    rotte_attive = [[i] for i in P]

    assegnazione = {}          # Lista di
       associazione di ogni rotta al veicolo

    # Distribuzione dei nodi tra i veicoli
       disponibili
    indice_veicolo = 0
    for r_attiva in rotte_attive:
        # Trova un veicolo con capacita sufficiente,
           usando un round-robin per bilanciare
        assegnato = False
        for offset in range(len(V)):
            k = V[(indice_veicolo + offset)%len(V)]
            if verifica_capacita(r_attiva, d, Q[k]):
                assegnazione[tuple(r_attiva)] = k
                indice_veicolo = (indice_veicolo +
                   1) % len(V)  # Si passa al
                   prossimo veicolo
                assegnato = True
                break
```

2. Calcoliamo il saving secondo questa formula:

$$sav_{ij} = c_{0,i} + c_{j,n+1} - c_{ij}$$

Devo però considerare il fatto che precedentemente non sono stati calcolati i costi $c_{0,0}, c_{0,n+1}, c_{n+1,0}, c_{n+1,n+1}$.

```
    savings = []
    for i in range(len(N)):
        for j in range(i+1, len(N)):
            if (i,j) in A:
                # Il costo c precedentemente non era
                   stato calcolato per i nodi
                   # (0,0), (0,11), (11,0) e
                      (11,11). Di conseguenza e'
                      stato necessario escludere
                   # tali nodi
                if i==0 and j==0:
                    sav = 0
                elif i==0:
```

```
                sav = c[j,n+1] - c[i,j]
            elif j==0:
                sav = c[0,i] - c[i,j]
            elif i==n+1 and j==n+1:
                sav = 0
            elif i==n+1:
                sav = c[j,n+1] - c[i,j]
            elif j==n+1:
                sav = c[0,i] - c[i,j]
            else:
                sav =c[0,i] + c[j,n+1] - c[i,j]
            savings.append(((N[i], N[j]), sav))
```

Successivamente i saving sono ordinati in ordine decrescente

```
savings.sort(key=lambda x: x[1], reverse=True)
```

3. Per ognuno degli archi si controlla che il proprio saving sia maggiore di 0; successivamente si trovano le rotte contenti il nodo i e il nodo j come estremi, e i veicoli assegnati a ognuno di queste rotte.
   Si provano a unire le due rotte, e si assegna tale rotta a un veicolo: se i veicoli diversi, viene escelto quello con meno rotte o con più capacità se hanno lo stesso numero di rotte.

```
for (i,j),sav_ij in savings:
    if sav_ij <= 0:             # I savings <= 0
        possono essere esclusi
        break

    iterazione += 1

    # Si trovano le rotte contenenti i e j come
      estremi
    rotta_i, indice_i = trova_rotta_contenente(i
      , rotte_attive)
    rotta_j, indice_j = trova_rotta_contenente(j
      , rotte_attive)

    # Se non sono stati trovati, si saltano
    if rotta_i is None or rotta_j is None or
      rotta_i==rotta_j:
        continue

    # Ottieni i veicoli assegnati
    veicolo_i = assegnazione.get(tuple(rotta_i))
```

```
        veicolo_j = assegnazione.get(tuple(rotta_j))

        if veicolo_i is None or veicolo_j is None:
            continue

        # Conta le rotte per veicolo
        conteggio_rotte = conta_rotte_per_veicolo(
           assegnazione, V)

        # Tentativo di unione
        nuova_rotta = unisci_rotte(rotta_i, rotta_j,
            indice_i, indice_j)


        if veicolo_i == veicolo_j:
            veicolo_scelto = veicolo_i
        else:
            # Se veicoli diversi, viene scelto
               quello con meno rotte
            # o con piu' capacita se hanno lo stesso
                numero di rotte
            if conteggio_rotte[veicolo_i] <
               conteggio_rotte[veicolo_j]:
                veicolo_scelto = veicolo_i
            elif conteggio_rotte[veicolo_i] >
               conteggio_rotte[veicolo_j]:
                veicolo_scelto = veicolo_j
            else:
                # Scelta del veicolo con maggiore
                   capacita
                if Q[veicolo_i] >= Q[veicolo_j]:
                    veicolo_scelto = veicolo_i
                else:
                    veicolo_scelto = veicolo_j
```

4. Si verifica che siano i rispettati i vincoli. In caso positivo, sono rimosse le vecchie rotte ed è aggiunta la nuova rotta.

```
    if (verifica_capacita(nuova_rotta, d, Q[
       veicolo_scelto])) and (verifica_finestra(
       nuova_rotta, t, s, a, b, veicolo_scelto)):
        # Rimozione delle vecchie rotte
        rotte_attive.remove(rotta_i)
        rotte_attive.remove(rotta_j)
        del assegnazione[tuple(rotta_i)]
        del assegnazione[tuple(rotta_j)]
```

```
        # Aggiunta della nuova rotta
        rotte_attive.append(nuova_rotta)
        assegnazione[tuple(nuova_rotta)] =
           veicolo_scelto
```

5. Controlliamo posteriormente il numero di rotte per ogni veicolo. Nonostante le prevenzioni attuate precedentemente (al momento dell'assegnazione delle rotte ai veicoli), è capitato che alcuni veicoli abbiano più di una rotta, mentre altri veicoli 0 rotte, non rispettando i vincoli precedentemente fissati. Di conseguenza, per risolvere questo problema, abbiamo deciso di effettuare un ulteriore controllo, seguendo le seguenti logiche:

   (a) Se uno dei veicoli ha più di una rotta e un veicolo ha 0 rotte, assegno una delle 2 rotte al veicolo con 0 rotte

   (b) Se uno dei veicoli ha più di una rotta e gli altri veicoli hanno tutti almeno una rotta, unisco le rotte di quel veicolo

```
    # Punto 1
    # Controllo quali veicoli hanno piu' di una
       rotta e quali hanno 0 rotte
    veicoli_senza_rotte = [v for v, r in
       rotte_per_veicolo.items() if len(r) == 0]
    veicoli_con_piu_rotte = [v for v, r in
       rotte_per_veicolo.items() if len(r) > 1]

    # Se esistono veicoli con le caratteristiche
       sopra specificate, eseguiamo il punto 1
    if veicoli_senza_rotte and veicoli_con_piu_rotte
       :
        veicolo_vuoto = veicoli_senza_rotte[0]
        veicolo_con_rotte = veicoli_con_piu_rotte[0]
        # Spostamento della rotta
        rotta_da_spostare = rotte_per_veicolo[
           veicolo_con_rotte].pop()
        rotte_per_veicolo[veicolo_vuoto].append(
           rotta_da_spostare)
        assegnazione[tuple(rotta_da_spostare)] =
           veicolo_vuoto


    # Punto 2
    # Eventuale unione delle rotte
    for veicolo, rotte in rotte_per_veicolo.items():
```

```
        # Controllo la lunghezza delle rotte
        if len(rotte) > 1:
            for i in range(len(rotte) - 1):
                rotta_attuale = rotte[i]
                rotta_successiva = rotte[i+1]

                # Unione delle rotte
                nuova_rotta = rotta_attuale +
                   rotta_successiva

                # Controllo dei vincoli
                if (verifica_capacita(nuova_rotta, d
                   , Q[veicolo]) and
                   verifica_finestra(nuova_rotta, t,
                   s, a, b, veicolo)):

                    # Rimozione vecchie rotte
                    rotte.remove(rotta_attuale)
                    rotte.remove(rotta_successiva)
                    del assegnazione[tuple(
                       rotta_attuale)]
                    del assegnazione[tuple(
                       rotta_successiva)]

                    # Aggiunta nuova rotta
                    rotte.append(nuova_rotta)
                    assegnazione[tuple(nuova_rotta)]
                        = veicolo
```

Capitolo 5

Risoluzione tramite Algoritmo TabuSearch

Un approccio più avanzato per la risoluzione del VRPTW è l'utilizzo di una meta-euristica come la **Tabu Search**. Questo algoritmo esplora lo spazio delle soluzioni in modo più intelligente rispetto a un approccio puramente greedy, cercando di evitare di rimanere bloccato in minimi locali. Per quanto riguarda la generazione dei dati di base, abbiamo utilizzato la stessa configurazione vista nei capitoli precedenti.

5.1 Breve descrizione dell'algoritmo

La Tabu Search è una tecnica di ottimizzazione che esplora lo spazio delle soluzioni muovendosi da una soluzione a una ad essa "vicina".
La caratteristica fondamentale è l'uso di una **lista Tabu**, una memoria a breve termine che registra le mosse recenti (o le soluzioni visitate) e le "proibisce" per un certo numero di iterazioni (il *tenure*). Questo meccanismo costringe l'algoritmo a esplorare nuove aree dello spazio delle soluzioni, diversificando la ricerca.
L'algoritmo include anche un **criterio di aspirazione**, che consente di ignorare il divieto imposto dalla lista Tabu se una mossa proibita porta a una soluzione significativamente migliore di quella migliore trovata finora.
Il processo si articola nei seguenti passi:

1. Si parte da una soluzione iniziale ammissibile, detta Pre-soluzione. Tale pre-soluzione è una famiglia di sottoinsiemi $\mathcal{C} = \{C_1, C_2, ..., C_t\}$, uno per ogni veicolo, con la proprietà che ogni punto vendita sia assegnato ad un solo sottoinsieme C_i. Una pre-soluzione per essere ammissibile deve avere la somma delle domande dei punti vendita inferiore alla capacità del veicolo e deve rispettare i vincoli di finestra temporale

2. Si genera un "vicinato" di soluzioni alternative partendo dalla soluzione corrente, tramite una mossa di swap tra due nodi

3. Si sceglie la migliore soluzione nel vicinato che non sia nella lista Tabu (a meno che non soddisfi il criterio di aspirazione)

4. Si aggiorna la soluzione corrente con quella scelta e si aggiunge la mossa alla lista Tabu. La lista Tabu è una memoria a breve termine, poichè a seguito di un certo numero di iterazioni (chiamate tenure), la mossa viene rilasciata ed è rieseguibile

5. Il processo viene ripetuto per un numero prefissato di iterazioni

5.2 Spiegazione del codice

L'algoritmo che andremo a utilizzare per effettuare la TabuSearch in relazione al VRPTW è quello di **Hertz, Gendreau e Laporte**.
Di seguito vengono illustrate le funzioni principali. Le funzioni ausiliarie sono quella di verifica della capacità e quella di verifica della finestra temporale, che sono identiche a quelle utilizzate nell'algoritmo euristico del capitolo precedente e non verranno ripetute.

5.2.1 Funzione di Fitness

Questa funzione calcola il costo totale di una data soluzione. Il costo è definito come la somma dei costi di tutti gli archi percorsi dai veicoli, inclusi i tragitti dal deposito di partenza al primo cliente e dall'ultimo cliente al deposito di arrivo.

```
    def fitness(solution, c, depot_s, depot_t):
        total_cost = 0
        # La 'solution' e' un dizionario {veicolo_id: [
           cliente1, cliente2, ...]}}
        for veiculo_id, route in solution.items():
            if route:
                # 1. Costo del deposito in partenza 's'
                   -> primo cliente
                total_cost += c.get((depot_s, route[0]),
                   MAX_COST)

                # 2. Costo tra i clienti nella rotta.
                for i in range(len(route) - 1):
                    total_cost += c.get((route[i], route[
                       i+1]), MAX_COST)

                # 3. Costo dall'ultimo cliente ->
                   deposito in arrivo 't'.
                total_cost += c.get((route[-1], depot_t),
                    MAX_COST)
        return total_cost
```

5.2.2 Generazione della Soluzione Iniziale

Per avviare la ricerca, è necessaria una soluzione di partenza ammissibile. Questa funzione implementa una strategia *Greedy* di inserimento: i punti vendita vengono ordinati per orario di chiusura e inseriti uno alla volta nella posizione e nella rotta che minimizzano l'aumento del costo totale, garantendo sempre il rispetto dei vincoli.

```
    def initialize_solution(P, V, Q, d, t, s, a, b, c,
       depot_s, depot_t):
        remaining_clients = list(P)
        # Ordina i clienti in base all'orario di chiusura
            (Euristica per rispettare i Time Window)
        remaining_clients.sort(key=lambda c_id: b[c_id])
        initial_solution = {v: [] for v in V}

        max_attempts = len(P) * len(V) * 20 # Limite per
           evitare loop infiniti
        attempt_count = 0

        while remaining_clients:
```

```
            client_to_assign = remaining_clients.pop(0) #
                Prende il prossimo cliente da assegnare
            best_insertion = None
            min_route_cost = MAX_COST

            # Cerca la migliore posizione e il miglior
               veicolo per l'inserimento
            for veicolo_id in V:
                current_route = initial_solution[
                   veicolo_id]
                for pos in range(len(current_route) + 1):
                    # Crea la rotta temporanea con l'
                       inserimento
                    temp_route = current_route[:pos] + [
                       client_to_assign] + current_route[
                       pos:]

                    # Verifica che la rotta modificata
                       sia ancora ammissibile (vincoli
                       VRPTW)
                    if is_feasible({veicolo_id:
                       temp_route}, d, Q, t, s, a, b):
                        temp_solution = deepcopy(
                           initial_solution)
                        temp_solution[veicolo_id] =
                           temp_route
                        # Calcola il costo totale della
                           soluzione dopo l'inserimento
                        new_cost = fitness(temp_solution,
                            c, depot_s, depot_t)

                        # Trovato un inserimento migliore
                            (Greedy)
                        if new_cost < min_route_cost:
                            min_route_cost = new_cost
                            best_insertion = (veicolo_id,
                                temp_route)

            # $U+FFFD$ata trovata una posizione
               ammissibile
            if best_insertion:
                veicolo_id, final_route = best_insertion
                initial_solution[veicolo_id] =
                   final_route # Aggiorna la soluzione con
                    il miglior inserimento
            else:
                # Se nessun veicolo/posizion[U+FFFD]
```

```
                    ammissibile, rimette il cliente in coda
                     e riprova
                remaining_clients.append(client_to_assign
                    )

            attempt_count += 1
            if attempt_count > max_attempts:
                # Lancia un'eccezione se non riesce a
                    trovare una soluzione ammissibile dopo
                    troppi tentativi
                raise Exception("Impossibile generare una
                     soluzione iniziale ammissibile.
                    Controlla i vincoli.")

        initial_cost = fitness(initial_solution, c,
            depot_s, depot_t)
        # Restituisce la soluzione iniziale trovata e il
            suo costo
        return initial_solution, initial_cost
```

5.2.3 Generazione del Vicinato (Swap)

Il "vicinato" è l'insieme delle soluzioni raggiungibili dalla soluzione corrente tramite una mossa. In questo caso, la mossa utilizzata è lo **swap** (scambio) di due clienti. Vengono esplorati due tipi di swap:

- **Intra-rotta**: scambio di due clienti all'interno della stessa rotta
- **Inter-rotta**: scambio di due clienti appartenenti a rotte diverse

Per ogni possibile scambio, si verifica se la mossa è Tabu e se la soluzione risultante è ammissibile.

```
    def find_neighborhood(current_solution, TABU, d, Q, t
        , s, a, b):
        neighborhood = [] # Lista per le soluzioni vicine
            ammissibili
        moves = []        # Lista per le mosse
            corrispondenti
        current = current_solution
        vehicle_ids = list(current.keys())

        # Itera su tutti i nodi (clienti) in tutte le
            rotte
```

```
        for v1_id in vehicle_ids:
            route1 = current[v1_id]
            for idx1 in range(len(route1)):
                node1 = route1[idx1]

                # 1. Swap Intra-Rotta (scambio all'
                   interno della stessa rotta)
                for idx2 in range(idx1 + 1, len(route1)):
                    node2 = route1[idx2]
                    move = (node1, node2); inverse_move =
                        (node2, node1)

                    # Controllo Tabu: Ignora le mosse
                       proibite
                    if move not in TABU and inverse_move
                       not in TABU:
                        new_solution = deepcopy(current)
                        new_route1 = new_solution[v1_id]
                        # Esegue lo swap dei nodi
                        new_route1[idx1], new_route1[idx2
                           ] = new_route1[idx2],
                           new_route1[idx1]

                        # Verifica l'ammissibilit[U+FFFD]della
                            rotta modificata
                        if is_feasible({v1_id: new_route1
                           }, d, Q, t, s, a, b):
                            neighborhood.append(
                               new_solution)
                            moves.append(move)

                # 2. Swap Inter-Rotta (scambio tra due
                   rotte diverse)
                for v2_id in vehicle_ids:
                    if v1_id != v2_id:
                        route2 = current[v2_id]
                        for idx2 in range(len(route2)):
                            node2 = route2[idx2]
                            move = (node1, node2);
                               inverse_move = (node2,
                               node1)

                            # Controllo Tabu: Ignora le
                               mosse proibite
                            if move not in TABU and
                               inverse_move not in TABU:
                                new_solution = deepcopy(
```

```
                                    current)
                                new_route1 = new_solution
                                    [v1_id]
                                new_route2 = new_solution
                                    [v2_id]

                                # Esegue lo swap inter-
                                    rotta
                                new_route1[idx1],
                                    new_route2[idx2] =
                                    new_route2[idx2],
                                    new_route1[idx1]

                                # Verifica l'
                                    ammissibilit[U+FFFD]di
                                    entrambe le rotte
                                    modificate
                                if is_feasible({v1_id:
                                    new_route1, v2_id:
                                    new_route2}, d, Q, t, s
                                    , a, b):
                                    neighborhood.append(
                                        new_solution)
                                    moves.append(move)

        return neighborhood, moves
```

5.2.4 Aggiornamento della Lista Tabu

Dopo aver eseguito una mossa, questa viene aggiunta alla lista Tabu per un numero di iterazioni pari al *tenure*. Ad ogni iterazione, il *tenure* di tutte le mosse nella lista viene decrementato di 1. Quando il *tenure* di una mossa raggiunge lo zero, essa viene rimossa dalla lista, tornando ad essere una mossa permessa.

```
    # Aggiorna la lista Tabu con la nuova mossa e
        decrementa il tenure delle mosse esistenti
    def update_tabu_list(TABU, best_move, tabu_size):
        if best_move is not None:
            node1, node2 = best_move
            inverse_move = (node2, node1)

            # Aggiunge la mossa e la sua inversa alla
                lista Tabu con il tenure specificato
```

```
            TABU[best_move] = tabu_size
            TABU[inverse_move] = tabu_size

        # Decremento del tenure
        moves_to_delete = []
        # Itera su tutte le mosse nella lista Tabu
        for move in list(TABU.keys()):
            TABU[move] -= 1 # Decrementa il tenure
            if TABU[move] <= 0:
                moves_to_delete.append(move) # Segna per
                   la cancellazione se il tenure scade

        # Rimuove le mosse con tenure scaduto
        for move in moves_to_delete:
            del TABU[move]
```

5.2.5 Flusso Principale della Tabu Search

Questa è la funzione che orchestra l'intero processo:

1. **Inizializzazione:** Viene generata una soluzione iniziale e impostata come soluzione corrente e migliore soluzione globale.
2. **Ciclo principale:** Per un numero prefissato di iterazioni:
   - Viene generato il vicinato della soluzione corrente
   - Viene selezionato il miglior "candidato" (la migliore soluzione vicina) che non sia Tabu. Viene applicato il criterio di aspirazione: una mossa Tabu è ammessa se porta a una soluzione migliore di quella migliore mai trovata
   - La soluzione corrente viene aggiornata con il candidato scelto.
   - Se il candidato è migliore della soluzione globale, quest'ultima viene aggiornata
   - La lista Tabu viene aggiornata con la mossa appena effettuata
3. **Terminazione:** L'algoritmo restituisce la migliore soluzione globale trovata.

```
    def tabu_search(P, c, t, d, a, b, V, Q, s, n_iters
        =1000, tabu_size=50):

        # Parametri del Deposito
        depot_s = 0              # Deposito in partenza '
            s' (indice 0)
        depot_t = len(P) + 1 # Deposito in arrivo 't' (
            indice N+1)

        # Stato della ricerca (popola la variabile
            globale STATE)
        global STATE
        # Dizionario che mantiene lo stato corrente e il
            migliore stato globale
        STATE = {
            'best_solution': None,               # La
                migliore soluzione *globale* mai trovata
            'best_cost': MAX_COST,               # Il
                costo della migliore soluzione globale
            'current_solution': None,    # La soluzione
                corrente su cui si basa la ricerca (
                potrebbe essere peggiore della '
                best_solution')
            'current_cost': MAX_COST,    # Il costo della
                soluzione corrente
            'TABU': {},                                  #
                 Dizionario che contiene le mosse proibite
                e il loro tenure rimanente
            'tabu_size': tabu_size,              # Durata
                di proibizione per una mossa (tenure)
            'n_iters': n_iters,                  # Numero
                massimo di iterazioni
            # Memorizza tutti i parametri del problema
                per l'accesso dalle funzioni ausiliarie
            'depot_s': depot_s, 'depot_t': depot_t,
            'c': c, 't': t, 'd': d, 'a': a, 'b': b, 'V':
                V, 'Q': Q, 's': s
        }

        try:
            # 1. Fase di Inizializzazione
            initial_solution, initial_cost =
                initialize_solution(P, V, Q, d, t, s, a, b,
                 c, depot_s, depot_t)

            # Imposta lo stato iniziale: soluzione
```

```
            corrente = soluzione iniziale
        STATE['current_solution'] = initial_solution
        STATE['current_cost'] = initial_cost

        # Imposta la migliore soluzione globale =
            soluzione iniziale (usiamo deepcopy per non
             alterarla)
        STATE['best_solution'] = deepcopy(
            initial_solution)
        STATE['best_cost'] = initial_cost
    except Exception as e:
        print(f"Errore: {e}")
        return None, None

    print(f"Inizio Ricerca Tabu. Costo iniziale: {
        STATE['best_cost']:.2f}")

    # 2. Ciclo Principale della Ricerca Tabu
    for iter_count in range(STATE['n_iters']):

        # Genera il vicinato (insieme di soluzioni
            raggiungibili con una mossa singola)
        # Vengono scartate immediatamente le mosse
            esplicitamente proibite (Tabu) O che
            portano a soluzioni non ammissibili (
            vincoli VRPTW violati)
        neighborhood, moves = find_neighborhood(
            STATE['current_solution'], STATE['TABU'],
                STATE['d'], STATE['Q'], STATE['t'],
            STATE['s'], STATE['a'], STATE['b']
        )

        if not neighborhood:
            # Condizione di stallo: se non ci sono
                vicini ammissibili e non Tabu, la
                ricerca si ferma
            print("Nessun vicino ammissibile trovato.
                Termine ricerca.")
            break

        best_candidate = None
        best_candidate_cost = MAX_COST
        best_move = None # Mossa che genera il
            miglior candidato

        # 3. Selezione del Miglior Vicino (anche se
            Tabu, con Aspirazione)
```

```
            for nb, move in zip(neighborhood, moves):
                nb_cost = fitness(nb, STATE['c'], STATE['
                   depot_s'], STATE['depot_t'])
                is_tabu = move in STATE['TABU'] #
                   Controlla se la mos$U+FFFD}rrentemente
                   proibita

                # Criterio di Aspirazione:
                # Accetta il candidato se:
                # a) N$U+FFFD}abu.
                # b$U+FFFD}abu, MA il suo costo (nb_cost)
                [U+FFFD]TTAMENTE MIGLIORE del miglior
                   costo globale ('best_cost').
                if not is_tabu or (is_tabu and nb_cost <
                   STATE['best_cost']):
                    # $U+FFFD}l candidato migliore finora
                       trovato nel vicinato
                    if nb_cost < best_candidate_cost:
                        best_candidate_cost = nb_cost
                        best_candidate = nb
                        best_move = move

            # 4. Aggiornamento dello Stato
            if best_candidate is not None:
                # Imposta il miglior candidato trovato
                   come la nuova soluzione corrente
                STATE['current_solution'] =
                   best_candidate
                STATE['current_cost'] =
                   best_candidate_cost

                # Verifica e aggiorna la migliore
                   soluzione globale
                if STATE['current_cost'] < STATE['
                   best_cost']:
                    STATE['best_cost'] = STATE['
                       current_cost']
                    # Salva la soluzione migliore (
                       Deepcopy essenziale!)
                    STATE['best_solution'] = deepcopy(
                       STATE['current_solution'])

                # Aggiornamento Lista Tabu: Proibisce la
                   mossa appena eseguita per un certo
                   tenure
                update_tabu_list(STATE['TABU'], best_move
                   , STATE['tabu_size'])
```

```
            # Stampa l'avanzamento ogni 100 iterazioni
            if (iter_count + 1) % 100 == 0:
                print(f"Iterazione {iter_count+1}/{STATE
                    ['n_iters']} - Costo corrente: {STATE['
                    current_cost']:.2f} | Costo migliore: {
                    STATE['best_cost']:.2f}")

        print(f"\nRicerca Tabu completata. Costo Finale:
            {STATE['best_cost']:.2f}")
        return STATE['best_solution'], STATE['best_cost']
```

Capitolo 6

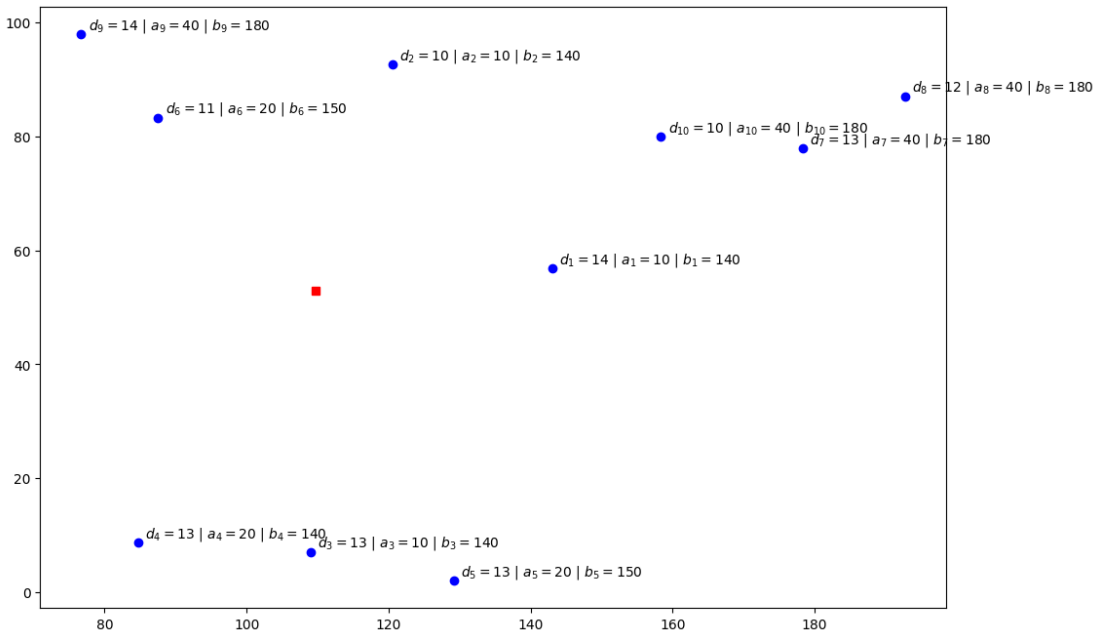
Casi d'uso

Di seguito saranno mostrati diversi casi d'uso con ambedue gli approcci

6.1 Caso con 10 punti vendita

In questo caso d'uso, prendiamo in considerazione 10 punti vendita, con le seguenti caratteristiche:

- Le coordinate sono le seguenti:



- Come specificato precedentemente, i costi degli archi sono scelti randomicamente, e non sono simmetrici, così come la richiesta di ogni punto vendita e i tempi di servizio; mentre i tempi di percorrenza sono dati dalla distanza dei punti vendita
- Le finestre di apertura e chiusura dei punti vendita sono le seguenti:

```
a = {0:0, 1:10, 2:10, 3:10, 4:20, 5:20, 6:20,
   7:40, 8:40, 9:40, 10:40, 11:0}      # Orario
   di apertura
b = {0:100, 1:140, 2:140, 3:140, 4:140, 5:150,
   6:150, 7:180, 8:180, 9:180, 10:180, 11:200} #
   Orario di chiusura
```

- I veicoli sono 4
- Le capacità dei veicoli sono i seguenti:

```
Q = {1: 50, 2:50, 3:25, 4:25}
```

- La funzione obiettivo è quella di minimizzare i costi

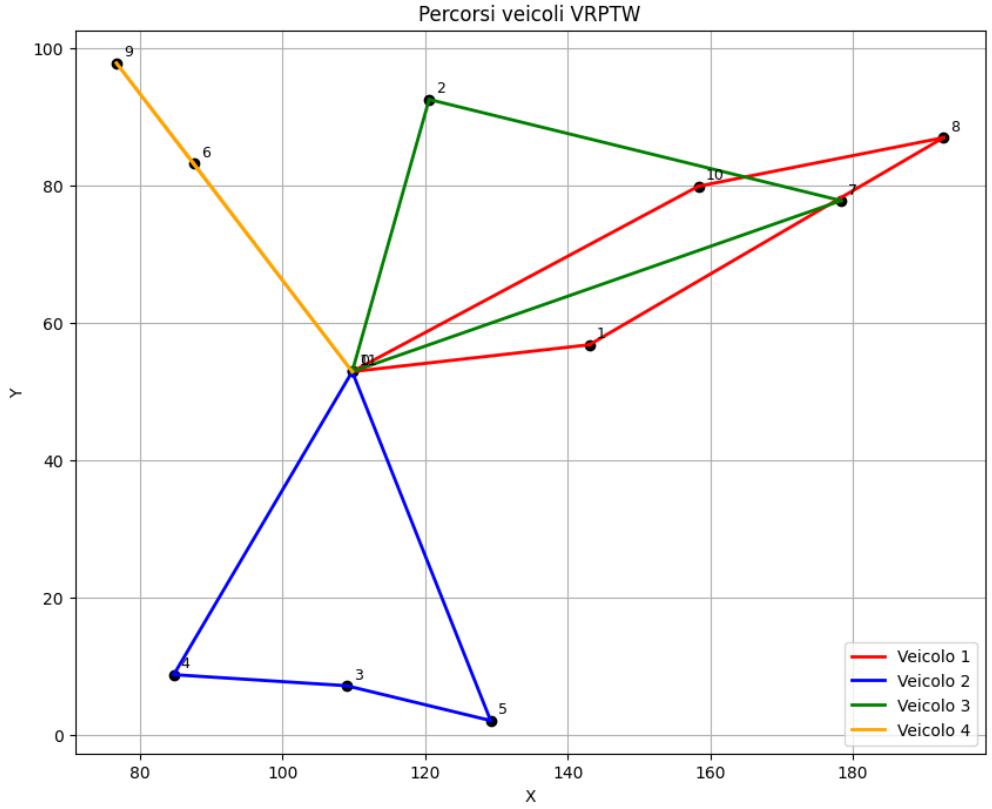
6.1.1 Algoritmo esatto

L'algoritmo esatto fornisce una soluzione ottima con il seguente costo e i seguenti percorsi:

```
Percorso veicolo 1:
  0 → 1
  1 → 8
  8 → 10
  10 → 11

Percorso veicolo 2:
  0 → 5
  3 → 4
  4 → 11
  5 → 3

Percorso veicolo 3:
  0 → 2
  2 → 7
  7 → 11

Percorso veicolo 4:
  0 → 6
  6 → 9
  9 → 11
```

```
        l[0,k] = a[0]

    model.solve()

    print("Costo: ", pulp.value(model.objective))

11]  ✓ 4m 2.3s

··  Costo:  65.0
```

Ottenendo quindi questo grafico:



Possiamo notare dalle immagini che il tempo di esecuzione tramite PuLP è pari a 4 minuti e 2 secondi.

6.1.2 Algoritmo euristico

Nel caso dell'algoritmo euristico, la situazione è più complessa e ha presentato varie problematiche descritte nel capitolo precedente. Per questo motivo dobbiamo fare una panoramica sulla soluzione prima e dopo il controllo del numero di rotte per ogni veicolo. Inizialmente otteniamo i seguenti percorsi:

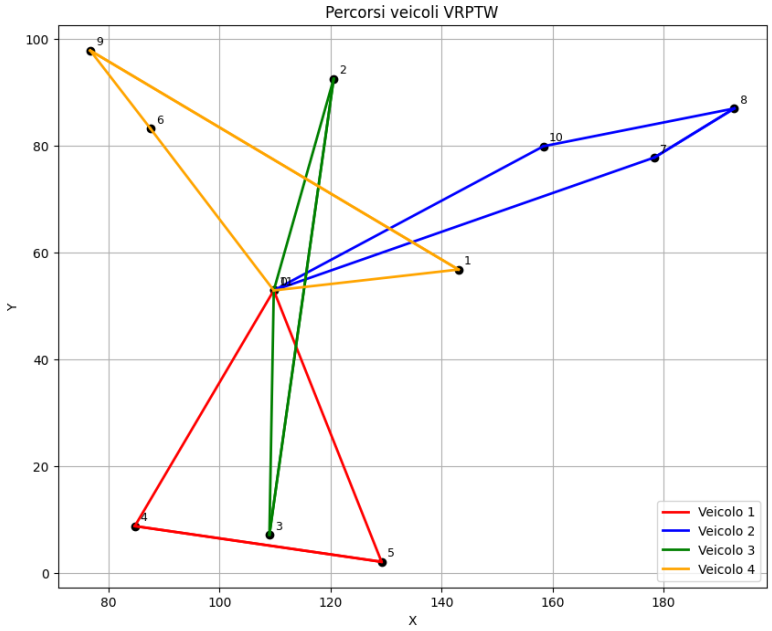
```
rotte_per_veicolo
✓ 0.0s  Open 'rotte_per_veicolo' in Data Wrangler  Python
{1: [[4, 5]], 2: [[7, 8, 10], [1, 9, 6]], 3: [[2, 3]], 4: []}
```

Possiamo notare la violazione del vincolo di partenza dei veicoli (cioè, tutti i veicoli devono partire una sola volta); infatti, il veicolo 2 presenta 2 rotte, mentre il veicolo 4 presenta 0 rotte.
Applicando il controllo sulle rotte dei veicoli, la situazione cambia, rispettando così il vincolo:

```
{1: [[4, 5]], 2: [[7, 8, 10]], 3: [[2, 3]], 4: [[1, 9, 6]]}
```

Ottenendo un costo pari a:

```
Rotta: [4, 5]
Veicolo 1: 0, 4 (costo: 9)
Veicolo 1: 4, 5 (costo: 11)
Veicolo 1: 5,11 (costo: 7)
Rotta: [7, 8, 10]
Veicolo 2: 0, 7 (costo: 11)
Veicolo 2: 7, 8 (costo: 9)
Veicolo 2: 8, 10 (costo: 5)
Veicolo 2: 10,11 (costo: 2)
Rotta: [2, 3]
Veicolo 3: 0, 2 (costo: 5)
Veicolo 3: 2, 3 (costo: 8)
Veicolo 3: 3,11 (costo: 13)
Rotta: [1, 9, 6]
Veicolo 4: 0, 1 (costo: 5)
Veicolo 4: 1, 9 (costo: 2)
Veicolo 4: 9, 6 (costo: 13)
Veicolo 4: 6,11 (costo: 4)
Costo totale: 104
```



Possiamo affermare quindi che:

- Il tempo di esecuzione è notevolemente inferiore rispetto al metodo esatto, essendo addirittura pari a 0 secondi (come mostrato nell'immagine)
- Il risultato risulta però essere peggiore rispetto a quello ottenuto dal metodo esatto, avendo il seguente gap

$$gap = |\frac{UB(I) - EUR(I)}{UB(I)}| \cdot 100 = |\frac{63 - 104}{63}| \cdot 100 \approx 58.7\%$$

6.1.3 Algoritmo TabuSearch

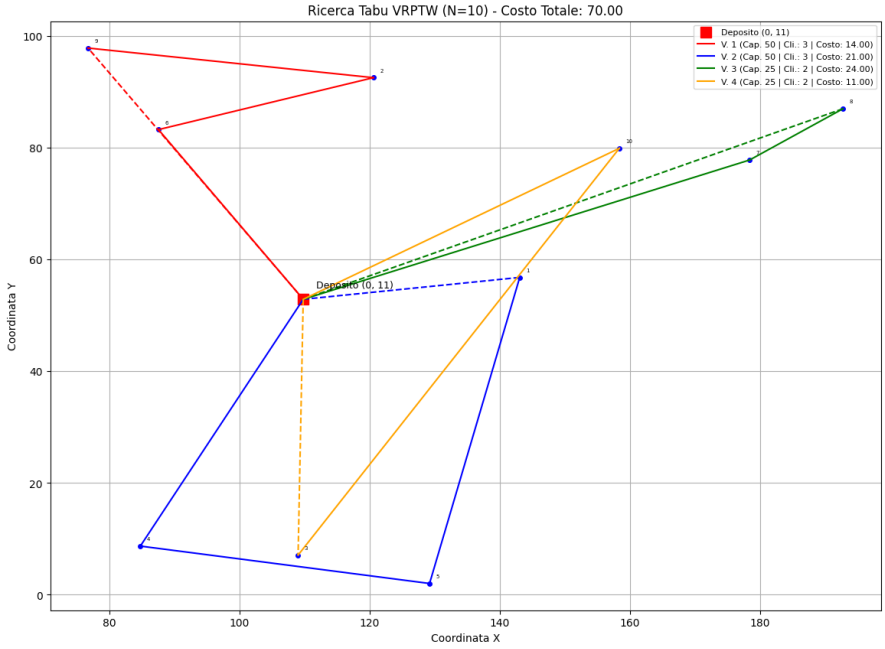
Nel caso dell'algoritmo di TabuSearch, possiamo notare come la soluzione risulti essere molto più soddisfacente rispetto all'algoritmo euristico precedentemente applicato:

```
Ricerca Tabu completata. Costo Finale: 70.00

--- Risultati Finali ---
Costo Totale Finale: 70.00
Soluzione (Rotta per Veicolo, solo Nodi Cliente):
Veicolo 1 (Cap. 50): Rotta di 3 clienti: [9, 2, 6]
Veicolo 2 (Cap. 50): Rotta di 3 clienti: [1, 5, 4]
Veicolo 3 (Cap. 25): Rotta di 2 clienti: [8, 7]
Veicolo 4 (Cap. 25): Rotta di 2 clienti: [3, 10]
```

```
# Visualizzazione della Soluzione Ottima (Output grafico)
plot_solution(final_solution, final_cost, xc, yc, n, Q, c, P)
```

✓ 0.2s

Ottenendo questo grafico:



Possiamo affermare quindi che:

- Anche in questo caso, il tempo di esecuzione è notevolmente inferiore rispetto al metodo esatto, ma maggiore rispetto all'algoritmo euristico (anche se la differenza di 0.2 s è talmente minima da essere impercettibile)

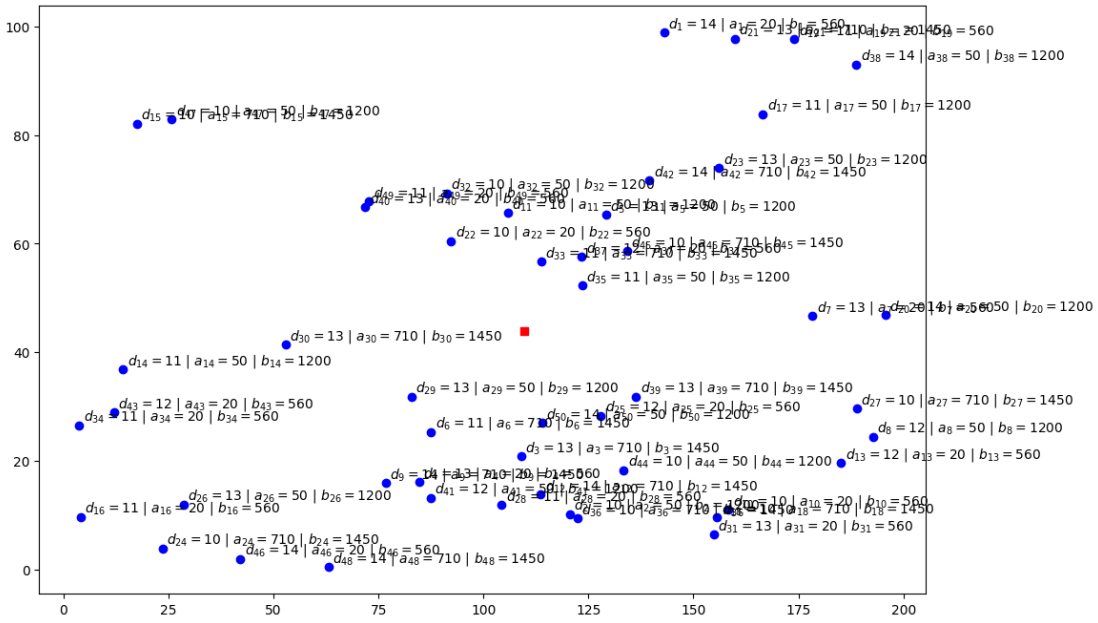
- Il costo minimo ottenuto è molto vicino alla soluzione esatta, e molto più basso rispetto alla soluzione euristica

$$gap = |\frac{UB(I) - EUR(I)}{UB(I)}| \cdot 100 = |\frac{63 - 70}{63}| \cdot 100 \approx 11.1\%$$

6.2 Caso con 50 punti vendita

In questo caso d'uso, prendiamo in considerazione 50 punti vendita, con le seguenti caratteristiche:

- Le coordinate sono le seguenti:



- Come specificato precedentemente, i costi degli archi sono scelti randomicamente, e non sono simmetrici, così come la richiesta di ogni punto vendita e i tempi di servizio; mentre i tempi di percorrenza sono dati dalla distanza dei punti vendita

- Le finestre di apertura e chiusura dei punti vendita sono le seguenti:

```
    # Orari depositi
    a[0], b[0] = 0, 1500
    a[n+1], b[n+1] = 0, 1500
    # Divisione in 3 fasce orario: {[20,480],
       [50,900], [710, 1450]}
    for i in P:
        r = i % 3
        if r == 1:
            a[i], b[i] = 20, 560
        elif r == 2:
            a[i], b[i] = 50, 1200
        else:
            a[i], b[i] = 710, 1450
```

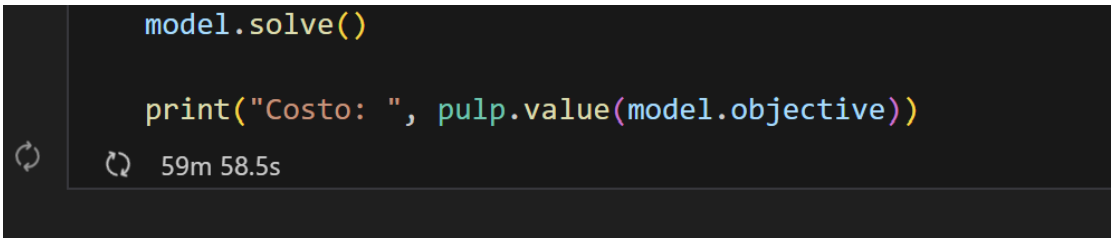
- I veicoli sono 10
- Le capacità dei veicoli sono i seguenti:

```
    Q = {1:90, 2:90, 3:80, 4:80, 5:90, 6:90, 7:80,
       8:80, 9:90, 10:100}
```

- La funzione obiettivo è quella di minimizzare i costi

6.2.1 Algoritmo esatto

PuLP non è stato in grado di trovare una soluzione a questo problema entro i 60 minuti, come mostrato in figura:



6.2.2 Algoritmo euristico

La situazione è invece differente per quanto riguarda il metodo euristico. In questo caso, è stato possibile trovare una soluzione in maniera repentina (0 secondi), ed è stato nuovamente necessario controllare e gestire i veicoli con più rotte:

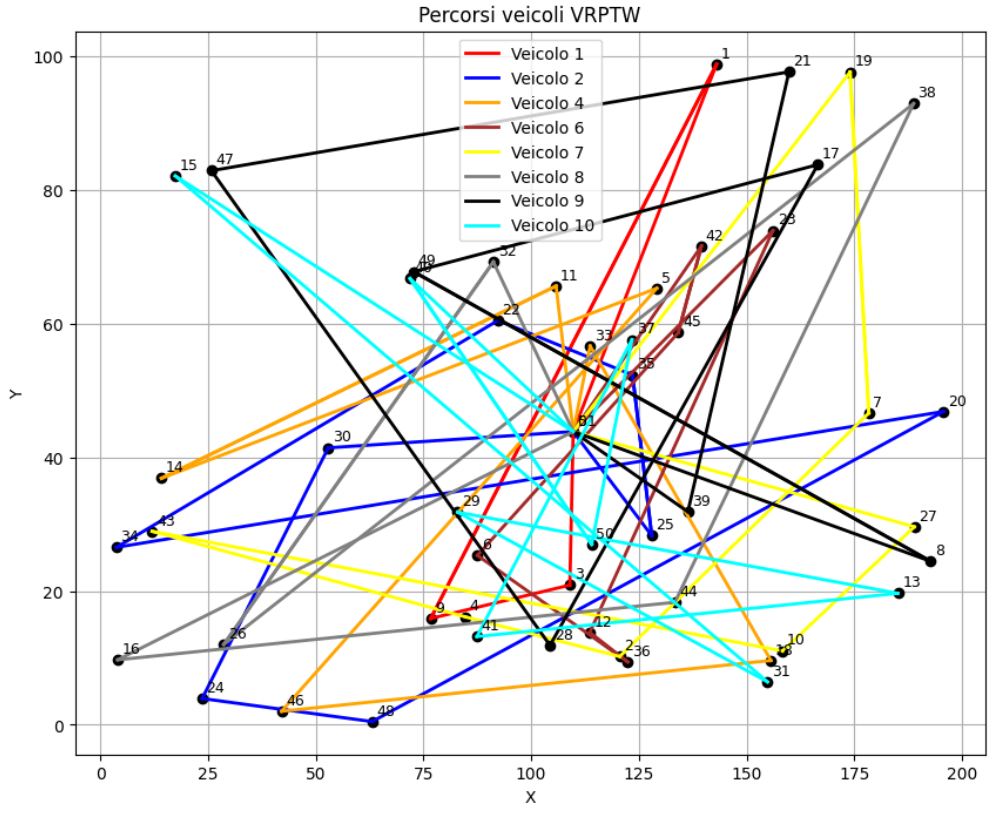
```
{1: [[1, 9, 3]],
 2: [],
 3: [],
 4: [[11, 14, 5, 46, 18, 33]],
 5: [],
 6: [[42, 45, 6, 36, 12, 23]],
 7: [[19, 7, 2, 4, 43, 10, 27]],
 8: [[32, 26, 38, 44, 16]],
 9: [[8, 49, 17, 28, 47, 21, 39]],
 10: [[40, 50, 37, 41, 13, 29, 31, 15], [25, 35, 22, 34, 20, 48, 24, 30]]}
```

```
{1: [[1, 9, 3]],
 2: [[25, 35, 22, 34, 20, 48, 24, 30]],
 3: [],
 4: [[11, 14, 5, 46, 18, 33]],
 5: [],
 6: [[42, 45, 6, 36, 12, 23]],
 7: [[19, 7, 2, 4, 43, 10, 27]],
 8: [[32, 26, 38, 44, 16]],
 9: [[8, 49, 17, 28, 47, 21, 39]],
 10: [[40, 50, 37, 41, 13, 29, 31, 15]]}
```

Il costo totale è pari a 342:

```
Costo totale: 342
```

Ottenendo così il seguente grafico:



Possiamo quindi concludere dicendo che:

- Il metodo euristico presenta una soluzione abbastanza costosa
- Il metodo euristico di nuovo riesce a ottenere una soluzione in brevi istanti
- Di contro, viene violato il vincolo imposto precedentemente (anche se facoltativo) per cui ogni veicolo deve partire

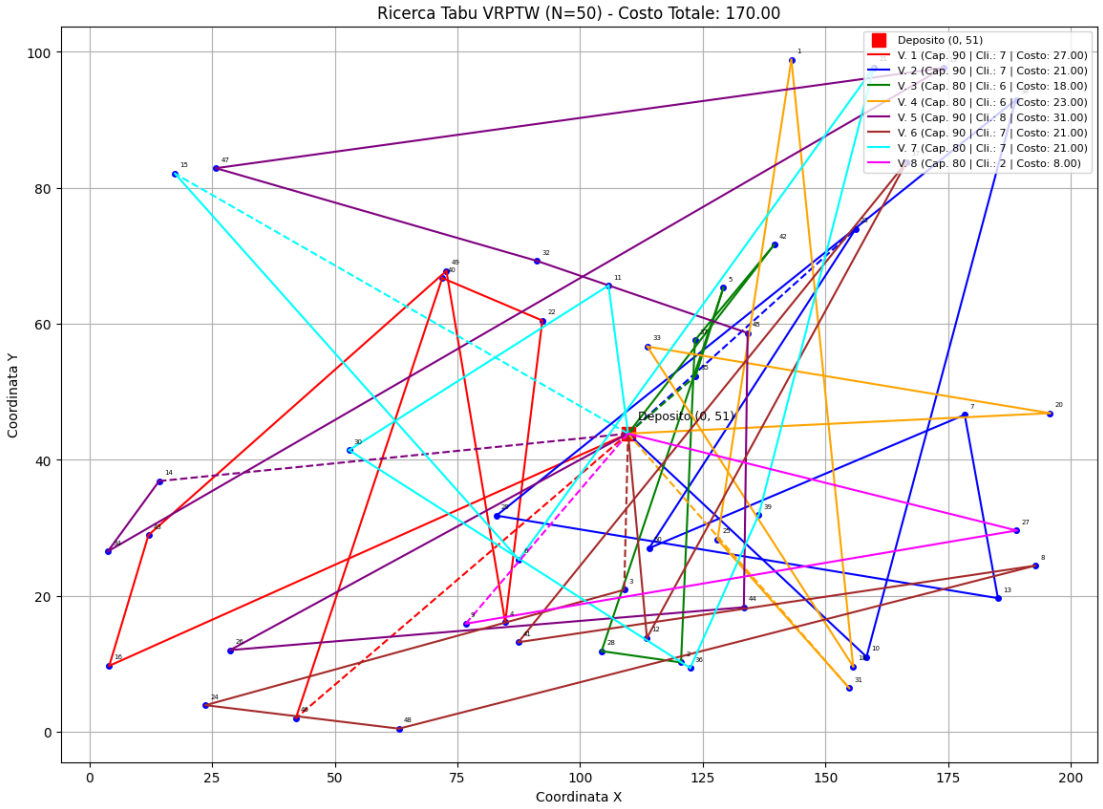
6.2.3 Algoritmo TabuSearch

Nel caso dell'algoritmo di TabuSearch, possiamo notare come anche in questo caso la soluzione risulti essere molto più soddisfacente rispetto all'algoritmo euristico precedentemente applicato:

```
--- Risultati Finali ---
Costo Totale Finale: 170.00
Soluzione (Rotta per Veicolo, solo Nodi Cliente):
Veicolo 1 (Cap. 90): Rotta di 7 clienti: [46, 40, 22, 4, 49, 43, 16]
Veicolo 2 (Cap. 90): Rotta di 7 clienti: [23, 50, 7, 13, 29, 38, 10]
Veicolo 3 (Cap. 80): Rotta di 6 clienti: [35, 5, 28, 2, 37, 42]
Veicolo 4 (Cap. 80): Rotta di 6 clienti: [31, 25, 1, 18, 33, 20]
Veicolo 5 (Cap. 90): Rotta di 8 clienti: [14, 34, 19, 47, 32, 45, 44, 26]
Veicolo 6 (Cap. 90): Rotta di 7 clienti: [3, 24, 48, 8, 41, 17, 12]
Veicolo 7 (Cap. 80): Rotta di 7 clienti: [15, 6, 21, 39, 36, 30, 11]
Veicolo 8 (Cap. 80): Rotta di 2 clienti: [9, 27]
```

```
# Visualizzazione della Soluzione Ottima (Output grafico)
plot_solution(final_solution, final_cost, xc, yc, n, Q, c, P)
```
✓ 1m 26.7s

Ottenendo questo grafico:



Possiamo quindi concludere dicendo che:

- Il metodo TabuSearch presenta una soluzione poco costosa
- Il metodo TabuSearch riesce a ottenere una soluzione rapida (1 minuto e 26 secondi)
- Di contro, viene violato il vincolo imposto precedentemente (anche se facoltativo) per cui ogni veicolo deve partire

Capitolo 7

Conclusioni

Per ovvie ragioni, la soluzione migliore risulta essere quella ottenuta tramite algoritmo esatto, che però risulta molto onerosa all'aumentare dei punti vendita; infatti, nel caso di 50 punti vendita dopo un'ora non si è ancora giunti a una soluzione. Di conseguenza non conviene in situazioni in cui la soluzione deve essere trovata in tempi ragionevoli. Il vantaggio principale dell'algoritmo euristico è la velocità. La soluzione anche nel caso di 50 punti vendita è istantanea, a discapito però della qualità della soluzione: il costo infatti risulta essere troppo elevato. Un buon compromesso è quindi l'algoritmo TabuSearch, che trova una soluzione vicina all'ottimo e in breve tempo.