Solving CVRP Problem with Classic Heuristics

Valerio Firmano (s302620)
Margherita Soro (s304667)
Cecilia Cardellino (s302675)

July 2022

Contents

CVRP

0.1 Presentazione del problema

In un CVRP, *Capacitated Veichule Routing Problem*, una flotta di veicoli di consegna con capacità uniforme deve servire dei clienti, ognuno dei quali necessita di una quantità nota di merce. I veicoli, in numero fissato, iniziano e terminano i loro percorsi in un deposito comune. Ogni cliente può essere servito da un solo veicolo.

L'obiettivo è assegnare una sequenza di clienti a ciascun veicolo della flotta, minimizzando la distanza totale percorsa, in modo che tutti i clienti siano serviti e che la domanda totale trasportata da ciascun veicolo non superi la sua capacità.

I **dati** del problema sono il numero di veicoli e la loro capacità, le coordinante dei clienti e le loro domande, le coordinate del deposito. Il **risultato** è, per ognuno dei veicoli, la sequenza ordinata di clienti da servire e il costo totale in termini di distanza.

Il TSP, *Travelling Salesperson Problem*, é il problema che prende in considerazione un singolo veicolo che visita più sedi di clienti prima di tornare al deposito con l'obiettivo di minimizzare la distanza totale percorsa. Per risolvere un problema di instradamento di un **numero fissato** di veicoli con **capacità limitate** (CVRP), dopo aver studiato varie opzioni, abbiamo pensato di suddividere i clienti in un numero di cluster pari al numero di veicoli disponibili -rispettando le capacità di ogni veicolo- e successivamente risolvere per ognuno di essi un problema TSP simmetrico.

Metodi risolutivi per il problema del commesso viaggiatore simmetrico

Possiamo classificare i metodi di risoluzione in 2 principali categorie:

1. I metodi **costruttivi** creano direttamente una soluzione a partire da un percorso parziale espanso attraverso un criterio ragionevole.

2. I metodi **iterativi** partono da una soluzione data e cercano di migliorare il percorso iniziale generando una sequenza di soluzioni alternative. Essi sono computazionalmente più onerosi e necessitano di un metodo costruttivo per la route iniziale ma permettono di ottenere dei risultati migliori.

Dopo aver considerato diverse possibilità, la nostra scelta è stata quella di implementare un metodo costruttivo basato sull'inserimento di nuovi archi, detto **extra miliage**, alla cui soluzione applichiamo un metodo iterativo detto **2-opt**,

2-opt è un algoritmo di ricerca locale la cui idea principale è quella di eliminare dal percorso, ad ogni iterazione, due archi e riconnettere i nodi con nuove combinazioni, cercando una combinazione che diminuisca il costo rispetto a quella di partenza.

La soluzione iniziale data in input all'algoritmo **2-opt** è generata con un approccio costruttivo basato sull'inserimento, a partire da cammino formato da un solo arco, di tutti i successivi punti appartenenti al cluster.

I metodi implementati, di cui sopra è descritta l'idea alla base, vengono esposti in modo dettagliato nella sezione successiva.

0.2 Funzioni

0.2.1 Initialize

Questa è la prima funzione che useremo per estrarre i dati utili e ottenere le prime informazioni che ci serviranno per le succesive valutazioni. In input, attraverso la cartella *Data* processiamo un file in formato "*csv*" estraiamo informazioni tipo:

- Coordinate (x, y) dei punti
- La loro relativa domanda
- Coordinate (x, y) del magazzino
- Numero di mezzi disponibili e la loro capacità

Inoltre definisco tre vettori:

- "Distance": calcolo della distanza euclidea tra tutti i punti del Dataset. (Moltiplico questa distanza per 10 evitando così di lavorare con numeri decimali).
- "Cost": calcolo del vettore dei Costi di tipo *Extra Mileage* che valuta la differenza tra la distanza *magazzino-punto j* e quella che include anche il punto i, cioè *magazzino-punto i-punto j*. In formule:

$$c_{i,k} = c_{0,1} + c_{i,\sigma_k} - c_{0,\sigma_k}$$

- "v": distanza tra il magazzino ed il punto j per due (costo per fare avanti indietro):

$$v_j = 2 * distance_{(0,j)}$$

0.2.2 Clustering-Introduzione

Abbiamo scritto due algoritmi per il clustering dei punti, entrambi considerabili come problemi di assegnamento generalizzato. Il primo, più complesso computazionalmente, risolve in contemporanea il problema dell'individuazione dei semi dei cluster e l'assegnamento dei punti ai differenti cluster individuati dai semi (Algoritmo di *Bramel Simchi-Levi)*). Il Secondo (Algoritmo di *Fisher Jaikumar*) invece, risolve solo il problema di assegnamento dei punti ai cluster, avendo fissato a priori le posizioni dei semi secondo un algoritmo che verrà spiegato in seguito. Questo Algoritmo è computazionalmente più veloce ed efficiente anche se i risultati potrebbero essere più miopi.

Il numero dei semi sarà pari al numero dei veicoli passato in input e la loro posizione viene scelta cercando tra i punti che hanno *"Cost"* relativo maggiore. In pratica si risolve, per trovare il seme σ_{k+1}:

$$\max_i \min\{c_{i0},\ c_{i\sigma_1}, c_{i\sigma_2},\ \dots, c_{i\sigma_k}\} \tag{1}$$

Dove $c_{i\sigma_k}$ è la distanza che abbiamo calcolato in *Initialize* del punto i dal seme k.

Schema **Algoritmo individuazione semi**:

1 Il primo seme sarà il punto più distante dal deposito.

2 Per ogni punto si calcola la distanza minima tra di essi e gli altri semi (ed il magazzino).

3 Si sceglie come nuovo seme quello con la distanza calcolata più lontana, cioè quello più lontano dagli altri semi e dal deposito (risolvo (1))

4 Ripeto dal "2" finchè non finisco il numero di veicoli.

0.2.3 Processo di Clustering - Algoritmo di *Fisher, Jaikumar*

Ogni cluster è caratterizzato in partenza da ogni seme trovato in precedenza. Questa funzione risolve un problema di ottimizzazione per distribuire tutti i punti del sistema all'interno dei vari cluster, tenendo conto della *Domanda* e della *Capacità* dei veicoli.
La funzione "Clustering 2" si serve della funzione **"optimproblem"** già implementata da Matlab, che risolve un problema di ottimizzazione vincolato:

VARIABILI DECISIONALI

- $y_{i,k} = 0, 1 \rightarrow$ Valore binario che indica se il cliente i è assegnato al mezzo k

VARIABILI DEL PROBLEMA

- $c_{i,k} = c_{0,1} + c_{i,\sigma_k} - c_{0,\sigma_k} \rightarrow$ Costo di tipo "Extra Mileage" di inserire i nel cammino k
- $d_i \rightarrow$ Domanda del cliente i

FUNZIONE OBIETTIVO

$$\sum_{k=1}^{m} \sum_{i=1}^{n} c_{ik} y_{ik} \tag{2}$$

VINCOLI

$$\begin{aligned} &\sum_{k=1}^{m} y_{ik} = 1 \quad \forall i = 1, ..., n \\ &\sum_{i=1}^{n} d_i y_{ik} \leq R \quad \forall k = 1, ..., m \\ &y_{ik} \in \{0, 1\} \end{aligned} \tag{3}$$

```
1 [sol_FJ,GAPprob_FJ]=clustering_2(numCostumers,numMacchine
      ,capacity,d,seed_pos,distance)
```

In input avremo:

- *numCostumers*: numero di punti totale, numero di clienti totale da servire
- *numMacchine*: numero di veicoli disponibili.
- *capacity*: capacità massima dei veicoli.
- *d*: vettore delle domande
- *seed pos*: coordinate dei punti che rappresentano i semi
- *distance*: la distanza relativa dei punti dai semi

In output otteniamo:

- *sol FJ*: la lista ottenuta con la funzione **solve** del problema primale proposto.
- *GAPprob FJ*: Il problema primale.

0.2.4 Processo di Clustering- Algoritmo di *Bramel Simchi-Levi*

Questa funzione non richiede di conoscere già i semi, anzi li calcola all'interno del processo stesso di definizione dei cluster.

Questo algoritmo è sicuramente più pensante ed addirittura inutilizzabile in alcuni casi, quando il problema si fa più complesso ed il numero di nodi cresce. Anche in questo caso utilizziamo la funzione già implementata da Matlab **optimproblem**.
Il problema in questo caso richiede un numero di vincoli maggiore perchè tra le variabili decisionali, oltre a quella già vista $y_{ik} \in \{0,1\}$ che indicano se il punto i fa parte del cluster k, dobbiamo definire anche $z_j \in \{0,1\}$ che indica se il punto $j-esimo$ è un seme di un cluster.
Il problema si formula nel seguente modo:

VARIABILI DECISIONALI

- $y_{i,j} = 0,1 \rightarrow$ Valore binario che indica se il cliente i è assegnato al cluster di cui j è seme
- $z_j = 0,1 \rightarrow$ Valore binario che indica se il cliente j è selezionato come seme del cluster

VARIABILI DEL PROBLEMA

- $g_{i,j} = c_{0,i} + c_{i,j} - c_{0,j} \rightarrow$ Costo di tipo "Extra Mileage"
- $v_j = 2c_{0,j} \rightarrow$ Costo della selezione di j come seme
- $d_i \rightarrow$ Domanda del cliente i

FUNZIONE OBIETTIVO

$$\sum_{k=1}^{m}\sum_{i=1}^{n} g_{ik}y_{ik} + \sum_{j=1}^{n} v_j z_j \tag{4}$$

VINCOLI

$$\begin{aligned}
&\sum_{j=1}^{m} y_{ij} = 1 \quad \forall i = 1,...,n \\
&\sum_{i=1}^{n} d_i y_{ij} \leq R \quad \forall k = 1,...,m \\
&\sum_{i=1}^{n} z_j = m \\
&y_{ij} \leq z_j \quad \forall i,j = 1,...,n \\
&y_{ij}, z_j \in \{0,1\}
\end{aligned} \tag{5}$$

L'algoritmo che stiamo implementando prende il nome di **Bramel Simchi-Levi** e funziona bene solo nei casi in cui il *numCostumers* non è moto elevato (max 75) e, soprattutto, ci sono pochi cluster (max 5/6).

```
[sol_BS, GAPprob_BS]=clustering(numCostumers, numMacchine,
    capacity, d, cost, v)
```

I parametri di input e output sono praticamente indentici a quelli di *"clustering 2"* tranne che per la posizione dei semi che viene rimpiazzata dal vettore dei costi *"cost"* e dal vettore *"v"* definiti nella funzione *itialize*.

0.2.5 TSP costructer

Funzione che si occupa della parte costruttiva del processo di definizione del percorso di un singolo mezzo all'interno di un singolo cluster.
Ad ogni passo dell'algoritmo si dividono i nodi in due insiemi, l'insieme V contiene i punti ancora da inserire nel percorso, mentre l'insieme T contiene i punti già assegnati in ordine.
Ad ogni passo bisogna scegliere un punto $k \in V$ da inserire in un arco aperto (i, j) con $i, j \in T$ aggiornando la sottosequenza in (i, k, j).
Esplicitamente l'algoritmo è:

1 Scelgo il primo nodo da collegare al magazzino come quello più lontano da esso

2 *Scelta di* k: il prossimo nodo che andrò ad aggiungere al percorso tra tutti i $k \in V$ sarà quello più vicino ad un $i \in T$.

3 *Scelta dell'arco* (i, j): l'arco scelto sarà quello che minimizza il Costo di tipo "Extra Mileage" $c_{ik} + c_{kj} - cij$.

4 Ora trasferisco il punto k da V a T.

5 Ripeto da "2" finchè non esaurisco tutti gli "elementi di V

Analizziamo Input ed Output:

```
function [T, distance_cluster]=TSP_constuctive(x_cluster,
    y_cluster)
```

In Input passiamo solo le corordinate dei punti del cluster di riferimento, mentre in output abbiamo in T già un ordinamento di punti che fa riferimento all'ordine di consegna del mezzo, mentre *distance cluster* è la solita distanza euclidea (moltiplicata per 10) utilizzata nella funzione e calcolata solo tra gli elementi del cluster.

0.2.6 2-Opt

Data la sequenza iniziale ricavata con la funzione *TSP costructive* implemento un metodo **iterativo** per migliorare il percorso e per cercare la soluzione ottima all'interno del cluster.
Questo algoritmo parte dalla sequenza iniziale a valuta tutte le permutazioni di

due elementi della sequenza, per ogni permutazione ne calcola il costo e infine sceglie quella con il costo più vantaggioso. Continua questo processo finchè lo scarto tra il vecchio costo di percorso e il nuovo è maggiore di una certa tolleranza passata in input.

```
1 function [T_1,ncycles,minimum]=opt2 (T,tolerance,
      distance_cluster)
```

OUTPUT

- *T 1* percorso aggiornato e definitivo
- *ncycles* numero di cicli necessari per arrivare alla tolleranza fissata
- *minimum* salvo il costo per servire tutti i punti all'interno del cluster di riferimento

0.2.7 Solve

All'interno di questa funzione semplicemente iteriamo sul numero di cluster in modo che ci restituisca direttamente i risultati degli algoritmi di risoluzione di tipo **TSP** per ogni singolo cluster.
Si occupa di sommare i costi minimi e quindi di trovare il costo complessivo del procedimento. Inoltre non si limita a salvare le sequenze definitive di consegna nei cluster ma ne salva anche le coordinate in modo che il *"plot"* sia più semplice da eseguire.
Questa funzione sfrutta fortemente le proprietà delle matrici per minimizzare il numero di passi.

0.3 Risultati

Mettiamo a confronto i Risultati ottenuti con i due differenti metodi di Clustering, che rappresentano il bottle neck del programma per quanto riguarda i tempi di esecuzione, dal momento che la risoluzione del TSP nei vari cluster richiede tempi di esecuzione bassi con l'euristica costruttiva e successivo 2-opt.

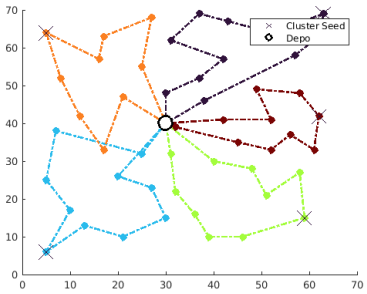
In particolare abbiamo testato il codice su setting di tipo E nelle librerie CVRP, ovvero insieme di punti generati secondo una distribuzione uniforme random in un raggio attorno al deposito e su settings più particolari e moderni per osservare il comportamento degli algoritmi.

I test sono condotti concentrandosi su: "fattibilità" dell'esecuzione dell'algoritmo di clustering, eventuali tempi di esecuzione, costo totale in confronto con il costo migliore presente in letteratura, plot delle soluzioni. I risultati sono riportati in tabella (1).

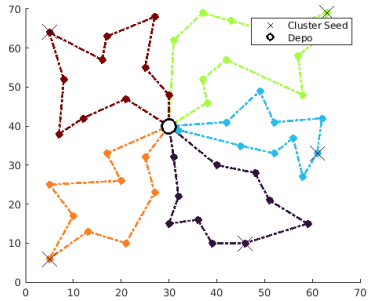
Setting	Fisher Jaikumar			Bramel Simchi-Levi			
	Time	Cost	Err_percento	Time	Cost	Err_percento	Best Value
E-n51-k5	1.4	558	7.00%	17	544	4.00%	521
E-n76-k10	2	885	6.60%	-	-	-	830
E-n101-k8	2.4	902	10.40%	-	-	-	817
B-n41-k6	1.58	838	1.00%	240.4	846	2.00%	829
P-n76-k5	1.6	699	11.00%	82.9	698	11.00%	627

Table 1: *Possiamo osservare in questa tabella il confronto di prestazioni tra l'algoritmo basato sul clustering di* ***Fisher Jaikumar*** *e quello basato sul clustering di* ***Bramel Simchi****. Sono riportati i tempi di esecuzione, il costo totale e l'errore percentuale rispetto alla migliore soluzione trovata fino ad ora in letteratura*

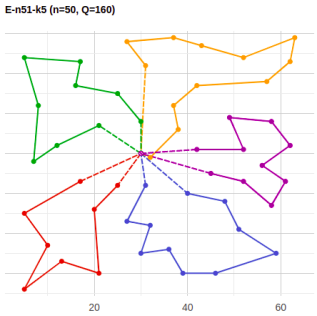
- **E-n51-k5**



(a) Algoritmo **Fisher Jaikumar**



(b) Algoritmo **Bramel Simchi-Levi**

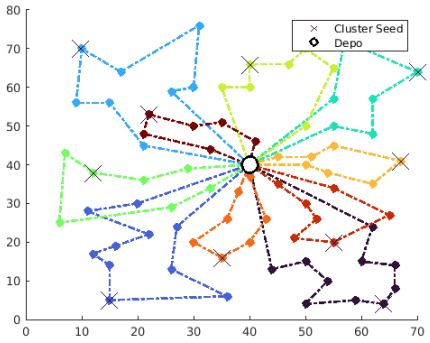


(c) Miglior soluzione in letteratura

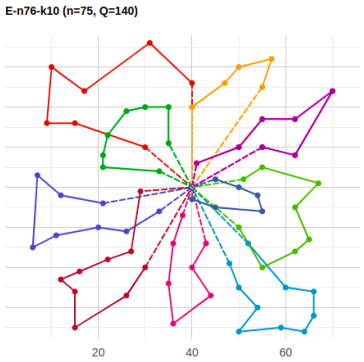
In questo caso abbastanza semplice entrambi i metodi di clustering funzionano velocemente e abbiamo una soluzione che migliora del 3% se utilizziamo l'algoritmo B.S. con un costo in termini di tempo di +15.5 secondi. La soluzione che troviamo ha comunque un errore relativo abbastanza

basso rispetto alla migliore presente in letteratura.

- **E-n76-k10**



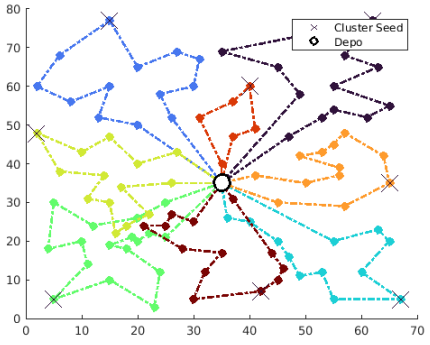
(a) Algoritmo **Fisher Jaikumar**



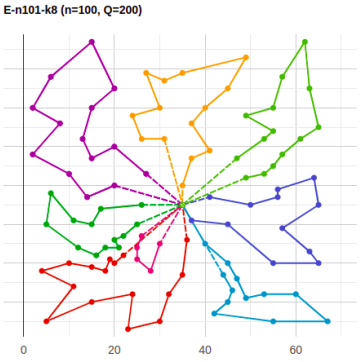
(b) Miglior soluzione in letteratura

In questo caso, aumentando il numero di punti e raddoppiando il numero di cluster, l'algoritmo *B.S.* non riesce a raggiungere una soluzione in tempi ragionevoli[1], e il risultato che troviamo con *F.J.* ha un errore relativo del 6% rispetto alla migliore soluzione, che comunque è un buon risultato.

- **E-n101-k8**



(a) Algoritmo **Fisher Jaikumar**



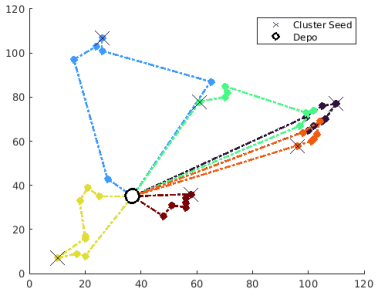
(b) Miglior soluzione in letteratura

Come nel caso precedente, l'algoritmo *B.S.* non riesce a raggiungere una soluzione in tempi ragionevoli dato l'alto numero di punti e cluster. Il risultato che troviamo con *F.J.* ha un errore relativo del 10% rispetto alla migliore soluzione. Probabilmente quello che influisce maggiormente su

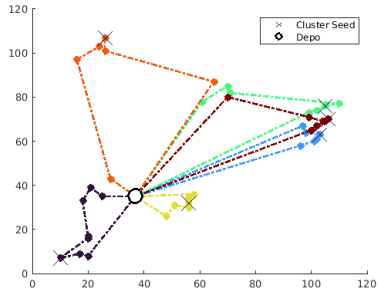
[1] Abbiamo provato a migliorare le prestazioni dell'algoritmo settando, ad esempio, la soluzione del *F.J.* come starting point del problema o provando a cambiare i parametri del solver *intlinprog* ma non siamo comunque riusciti ad arrivare ad un risultato

queste differenze è che con il nostro algoritmo posizioniamo a priori i semi dei cluster, facendo così una approssimazione non da poco conto sebbene i semi siano posizionato secondo un criterio ragionato.

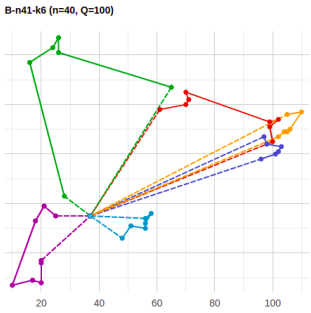
- **B-n41-k6**



(a) Algoritmo **Fisher Jaikumar**



(b) Algoritmo **Bramel Simchi-Levi**

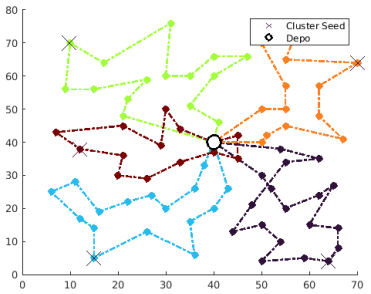


(c) Miglior soluzione in letteratura

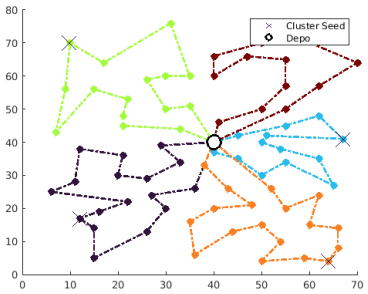
Come possiamo vedere dai Plot questo è un caso un po' più particolare rispetto ai precedenti, nel senso che i punti non sono distribuiti uniformemente nello spazio ma sono raggruppati in piccoli gruppi distanti tra di loro e dal centro. Per questo motivo probabilmente il posizionamento dei semi a priori del *F.J.* funziona particolarmente bene tanto da trovare un risultato con un errore relativo dell'1%. Lo stesso non si può dire di *B.S.* che invece ha tempi di esecuzione decisamente più lunghi e restituisce un risultato addirittura peggiore.

- **P-n76-k5**

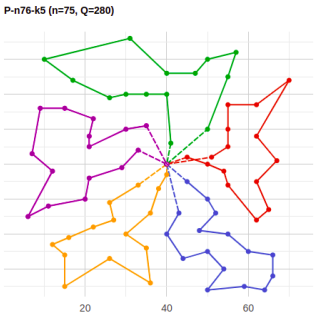
Anche in questo Set più "moderno" in cui i dati non sono distribuiti uniformemente i due algoritmi restituiscono una soluzione non particolarmente brillante, nel senso che (pur trovando due soluzioni parecchio diverse) entrambi ottengono un costo quasi identico, lontano dell'11% circa



(a) Algoritmo **Fisher Jaikumar**



(b) Algoritmo **Bramel Simchi-Levi**



(c) Miglior soluzione in letteratura

dalla soluzione ottimale. Inoltre, dato il numero di punti abbastanza alto, il *B.S.* impiega 83 secondi per trovare un risultato.

0.4 Conclusioni

Quello che si deduce dai risultati di questo progetto è che la strada della programmazione mista intera (modello assegnamento generalizzato) per eseguire i clustering del CVRP con un numero di cluster fissato può funzionare bene con problemi di relativa semplicità. É evidente come, nonappena il numero di cluster e/o di clienti aumenti abbiamo problemi di tempo di esecuzione per quanto riguarda il clustering *B.S.* e crescente imprecisione nella soluzione *F.J.*, che fissando i semi a priori può essere troppo approssimativa.

Nonostante questi problemi abbiamo che, comunque, risuciamo a trovare delle soluzioni che hanno un errore relativo al massimo del 10%, che è un buon risultato considerando la semplicità delle nostre euristiche.