Progetto di Algoritmi Avanzati

Capacitated Vehicle Routing Problem
Algoritmi: Costruttivi, a 2 Fasi e Genetici

Giulio Pilotto - Matricola:1140718

Luglio 2019

Abstract

Questo elaborato presenta il progetto di Algoritmi Avanzati sottomesso al prof. Bresolin dell'Università di Padova. Il progetto richiede di implementare almeno 2 algoritmi che risolvono istanze di : Capacitated Vehicle Routing Problem. Nel seguente elaborato oltre ad implementare i 2 algoritmi richiesti , il primo come metodo costruttivo; Clarke and Wright e il secondo come metodo a 2 fasi:ClusterFirst - Route Second per il quale sono state implementate due tecniche di routing: Nearest-Neighbourn e Dijkastra. Sono stati implementati altri 2 algoritmi uno che cade sempre nella classe dei metodi a 2 fasi: RouteFirst - ClusterSecond, e un altro che cade nella categoria degli algoritmi metauristici detti anche algoritmi genetici. Inoltre, è stato implementato un metodo di selezione dei centroidi che tiene conto sia della distanza dal deposito sia della distanza inter-cluster. Tutti gli algoritmi sono stati testati sulle 16 istanze fornite dalla libreria TSPLIB95. I risultati confermano che a fronte di una minore accuratezza di una soluzione rispetto all'ottimo ma una maggiore richiesta di risorse in termini di tempo, possono portare a soluzioni ben approssimate. In particolare scegliere i centrodi con il metodo RadiusRadar permette di guadagnare qualche decimo a fronte di un tempo inore. Le soluzioni posso essere migliorate attrverso gli algoritmi genetici che fungono da esploratori di uno spazion locale.

1 Introduzione

Con la nascita dei nuovi servizi per il cliente, che danno assistenza e portao i prodotti direttamente sulla soglia di casa (es:Amazon Prime), la gestione della logistica predilige gran parte degli investimenti. Questo fatto diventa ancor più importante se diamo un occhiata ai dati forniti dall' Associazione Nazionale Filiera Industria Automobilistica (ANFIA) nel 2017 1 , dove viene sottolineato che i camion movimentano l'80% delle merci su terra [1]. Gli investimenti non riguardano solo le strutture o gli strumenti fisici, ma soprattutto i software per la gestione e l'ottimizzazione della distribuzione dei beni. Grazie all'informatizzazione dei processi da parte del nuovo programma di industrializzazione chiamato "industria 4.0", si può godere di banche dati ricche ed aggiornate, attraverso le quali, possono essere implementate tecniche di ricerca operativa e ottimizzazione.

Il Vehicle Routing Problem **(VRP)** è un classe reale di problemi di soddisfacimento di vincoli, in inglese detto anche Constraint Satisfaction Problem **(COP)**. Alla fine degli anni cinquanta Dantzig and Ramser hanno formalizzato il problema [2] che riguarda la distribuzione di benzina da un deposito princiapale ad un grande numero di stazioni di servizio. Da quel momento l'interesse nei problemi VRP è evoluto da un piccolo gruppo di matematici ad un grande gruppo di ricercatori da differenti discipline ancora oggi coinvolti.

VRP consiste nell'ottimizzare l'uso di un insieme di veicoli per prelevare della merce da uno o più depositi e consegnarle presso dei clienti che richiedono una certa quantità di merce. Le stazioni , i depositi sono distribuiti in uno spazio rappresentato da delle distanze. I mezzi si spostano percorrendo tali distanze, che possono essere rappresentate in maniera diversa. La maggioranza the problemi reali sono sepsso più complessi del classico VRP. Quindi in pratica VRP è aumentato con dei vincoli, come ad esempio la capacità del veicolo: dove ogni veicolo è caratterizzato da una capacità limitata di carico delle merci, passando quindi da VRP a **Capacitated - VRP**. Gli obiettivi sono molteplici:

- Minimizzare dei costi di trasporto (istanze percorse, consumo di carburante)
- Minimizzare il numero di veicoli
- Rispettare il vincolo di capacità imposto sui veicoli.
- Assegnare un percorso ad ogni singolo veicolo.

VRP è un problema di ottimizzazione combinatoria NP hard,che può essere risolto esattamente trovando l'ottimo solo per piccole istanze del problema. In ogni caso gli approcci euristici che non garantiscono l'ottimalità, forniscono ottimi risultati in pratica. Negli ultimi 30 anni sono stati applicati anche dei metodi meta-euristici

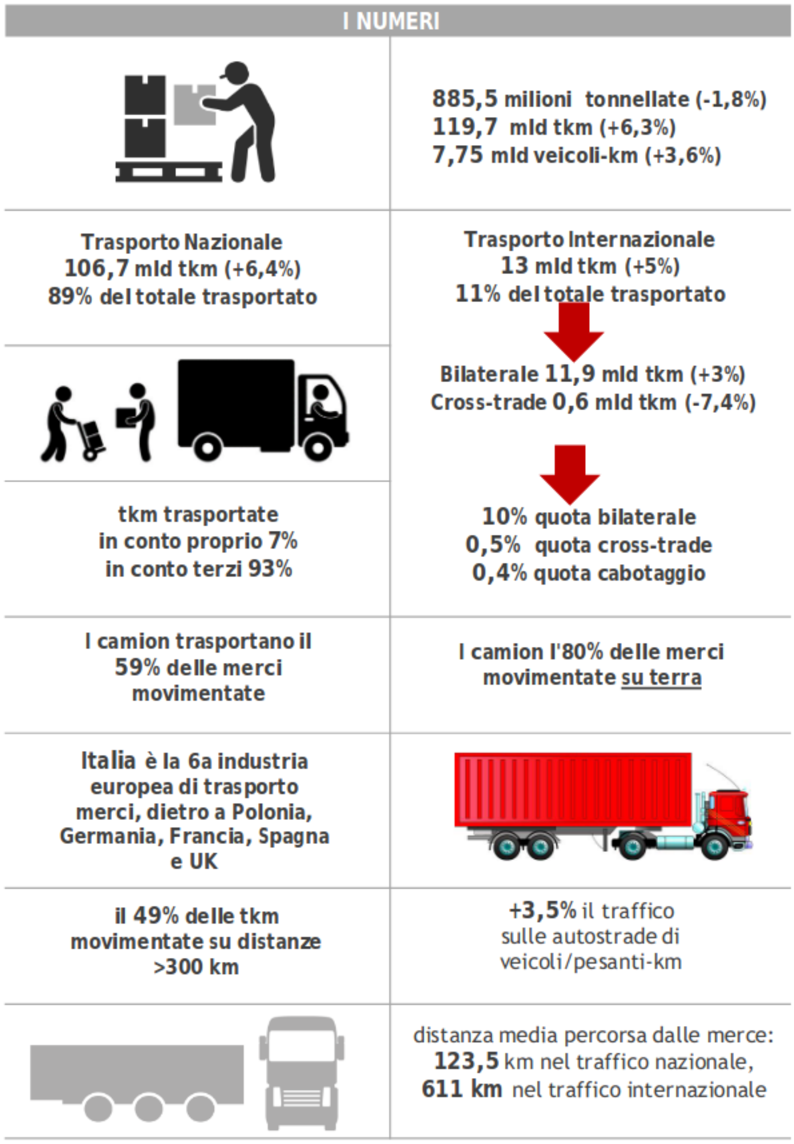


Figure 1: Italia, traffico merci su strada, ANFIA su dati ISTAT 2017

che hanno mostrato una nuova direzione alla ricerca sulla famiglia di problemi VRP. In questo elaborato vengono prese in esame le istanze fornite dalla libreria TSPLIB95[3].

Sono stati implementati i seguenti metodi per risolvere le istanze CVRP:

- Metodo Costruttivo: Clarke and Wright
- Metodo a due fasi:
  - Cluster-First Route-Second: Fisher and Jaikumar
  - Route-First Cluster-Second: Dijkastra
- Meta-euristiche: Algoritmi Genetici

1.1 Rappresentazione del problema

CVRP è la versione più comune dei problemi VRP. Ciò che caratterizza queta tipologia di problemi è il fatto che l servizio è di semplice ocnsgna senza raccolta. Inoltre le richieste dei clienti sononote a pripri e deterministiche e devono essere soddisfatte da un solo veicolo; tuti i veicoli sono identici e basati su un singolo deposito centrale. Gli unici vincoli imposti riguardano le capacità di veicoli. L'obbiettivo è minmizzare il costo totale di servizio che può essere una funzione del numero dei route, della lunghezza complessiva o del tempo di percorrenza. Consideraimo ora la rappresentazione su grafo di questo problema.

- Sia $G = (V,A)$ un grafo completo dove $V = \{0,...,n\}$ è l'insieme dei vertici e A quello degli archi.
- I vertici $i = i,...,n$ corrispondono ai clienti, mentre il vertice 0 corrisponde al deposito.
- Ad ogni arco $(i,j) \in A$ è associato un costo non negativo $c_{i,j}$, che rappresenta il costo di trasferimento dall'indice i all'indice j
- In genere l'uso di loop non è consentito e ciò è imposto definendo $c_{i,j}$=+∞ $\forall\ i \in\ V$.Nella nostra implementazione invece abbiamo definito $c_{i,j}$=0 $\forall\ i \in\ V$.
- Dato un insieme $S \subseteq V$, $d(S)$ denota la richiesta complessiva dei clienti in S: $d(S) = \sum_{s\in S} d_s$.
- Indicando con r una *route*, $d(r)$ denota la richiesta complessiva dei vertici da esso visitati.
- Un insieme K di veicoli è disponibile presso il deposito. Tali veicoli sono tutti identici di capacità C; una semplice condizione di ammissibilità del problema richiede $d_i \leq C\ \forall i \in V \setminus \{0\}$.
- Ogni veicolopuò percorrere al più un *route*.
- Si assume che $K \geq K_{min}$ dove K_{min} è il minimo numero di veicoli necessari per servire tutti i clienti.
- In questa implementazione utilizziamo il numero minimo di veicoli necessari per servire tutti i vertici in S, pari al *lower bound* $K_{min} = d(S)/C$

Gli obbiettivi ed i vincoli già citati nella sezione precedente vengono ora descritti in maniera formale, associando ai vincoli il modello matematico che li esprime.

$$M^k_{i,j} = \begin{cases} 1 : sse(i,j) \in A \wedge M_{i,j} \in r(k) \mid k \in K \wedge r \in route \\ 0 : altrimenti \end{cases}$$

Definiamo inoltre q_i la richiesta associata ad ogni cliente visitato da un circuito e C_k la capacità del veicolo $k \in K$.

La funzione obbiettivo descritta formalmente è la seguente:

$$\min \sum_{k\in K} \sum_{(i,j)\in A} c_{i,j} \cdot M^k_{i,j} \tag{1}$$

al quale sono applicati i seguenti vincoli:

$$\sum_{k\in K} \sum_{j\in V} M^k_{i,j} = 1\ \forall i \in V \tag{2}$$

$$\sum_{i\in V} d_i \sum_{j\in V} M^k_{i,j} \leq C_k\ \forall k \in K \tag{3}$$

$$\sum_{j\in V} M^k_{0,j} = 1\ \forall k \in K \tag{4}$$

$$\sum_{i\in V} M^k_{i,h} - \sum_{j\in V} M^k_{h,j} = 0\ \forall k \in K,\ \forall h \in V \tag{5}$$

$$\sum_{i\in V} M^k_{i,0} = 1\ \forall k \in K \tag{6}$$

La funzione obbiettivo 1, minimizzare il costo dei km totali percorsi da ogni singolo veicolo. Il vincolo 2 impone che ogni cliente deve essere servito da un solo veicolo. Il vincolo 3 assicura che il limite sulla capacità dei veicoli venga rispettato. I vincoli 4, 5 , 6 sono vincoli che impongono ad ogni veicolo di partire dal deposito il nodo 0, e collegarsi ad un nodo h, unico per ogni route,e di ritornare al deposito, indicato nella nostra implementazione sempre dal nodo 0. Questi vincoli definiscono la struttura della route. **Questa sezione va rivista**

2 Related Works

Quasi tutti i metodi implementati sono euristici perchè nessuno degli algoritmi può garantire di trovare una soluzione ottima per grandi istanze, in un limite di tempo computazionale ragionevole. Un approccio euristico non esplora l'intero spazio di ricerca, piuttosto cerca di trovare una soluzione basandosi sulle informazioni che ha sul problema. Le euristiche che risolvono istanze CVRP sono identificati come metodi costruttivi (**costructive**) e di raggruppamento **()clustering)**. I metodi costruttivi costruiscono una soluzione gradualmente,aggiornando continuamente l'informazione che riguarda il costo, ma potrebbe non contenere nessuna fase di miglioramento o di ottimizzazione della soluzione. Gli algoritmi costruttivi più famosi sono: Clarke and Wright's savings algorithm [4], [5] , [6] , Matching based algorithm e Multi-route improvement heuristic [6]. I metodi di clustering, risolvono il problema in due fasi , ed è per questo che vengono chiamati metodi a 2 fasi. l'approccio route-first cluster second è caratterizzato dalle due fasi seguenti:

- **Fase 1 - Clustering:** Un algoritmo di clustering viene utilizzato per raggruppare i clienti in cluster da dare in pasto alla seconda fase
- **Fase 2- Routing** Nella seconda fase, per ogni cluster creato nella prima fase viene ricercata la strada più corta sfruttando tecniche di ottimizzazione.

Alcuni famosi algoritmi a due fasi sono: Petal algortihm [7], Sweep algorithm [8] e Fisher and Jaikumar [9]. Questo studio investiga e compara le performance dei metodi costruttivi e di raggruppamento per risolvere istanze di CVRP. In particolare, nell'algoritmo di Fisher and Jaikumar, è stato implementato un algoritmo per la selezione dei centroidi chiamato Radar-Radius.Questo massimizza la distanza tra centroidi (inter-centroid distance) e la distanza tra i centroidi e il deposito in modo da garantire la migliore copertura, a seconda della distribuzione del deposito e dei clienti. Questo algoritmo è stato implementato scegliendo un deposito per ogni istanza in TSPLIB95,solitamente il deposito con id identificativo 1. Sono stati usati due metodi diversi per realizzare il routing tra i clienti di uno stesso cluster: Nearest Neighbourn e Dijkastra. I raggruppamenti formati dai metodi di clustering sono successivamente sottomesse a tecniche di ottimizzazione del routing. In particolare è stato implementato un Algoritmo Genetico **(GA)** [10], i quali sono famosi algoritmi utilizzati per risolvere istanze Travelling Salesman Problem (TSP). **Bensi i metodi costruttivi generano già delle soluzioni molto vicino all'ottimo, l'algoritmo Genetico è applicato anche al Savings algorithm per avere una misura di performance.**

3 Materiali e Metodi

3.1 Metodi Costruttivi

Un metodo in questa categoria costruisce le strade per i veicoli mentre cerca di minimizzare la distanza percorsa. L'algoritmo di Savings di Clarke and Wright [4], trova una soluzione utilizzando un'euristica, quindi il risultato non è sempre la soluzione ottima, ma dovrebbe risultare mol+to vicina all'ottimo con un risparmio di tempo computazionale elevato rispetto ad un algoritmo brute force.

3.2 Algoritmo di Savings di Clarke and Wright

L'algoritmo si basa su una coda ordinata di risparmi, detta anche *savings list*. Ogni record della coda è ottenuto collegando due *route* che non hanno nodi in comune eccetto il deposito. Dall'unione si ottiene una singola *route*, come mostrato nella figura 2, dove il nodo 0 rappresenta il deposito.

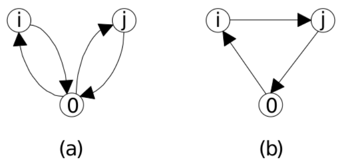


Figure 2: Inizialmente nella figura 1(a) i clienti i e j sono visitati da *route* separate. Un alternativa per visitare i due clienti con la stessa *route* è presentata in 1(b).

I costi di trasporto calcolati in 1(a) sono:

$$D_a = C_{o,i} + C_{i,0} + C_{o,j} + C_{j,0} \tag{7}$$

Mentre i costi di trasporto in 1(b) sono:

$$D_a = C_{o,i} + C_{i,j} + C_{o,j} \tag{8}$$

Ora il costo di trasporto risparmiato detto *saving distance* per visitare i e j nella stessa *route* invece che visitarli con due *route* separate è:

$$D_a = D_a + D_b = +C_{i,0} + C_{0,j} + C_{i,j} \tag{9}$$

3.2.1 Sequenziale e Parallelo

La distanza risparmiata è direttamente proporzionale alla vicinanza dei clienti e inversamente proporzionale alla vicinanza al deposito. Più grande è la distanza risparmiata più vicini si troveranno i clienti, e più distanti essi saranno dal deposito. Le coppie di clienti sono ordiate in ordine decrescente secondo il valore di *savings* che è stato calcolato sulla coppia. La costruzione delle *routes* parte dal primo record della coda. Quando una coppia i e j viene considerata se questa è già presente in una *route* allora si passa alla prossima coppia per l'inserimento. Ci sono due approcci proposti per l'algoritmo di *savings* **sequenziale** e **parallelo**.

- Nell'approccio **sequenziale** se una coppia di cliente non corrisponde con uno dei nodi presenti nella strada, la coppia viene ignorata. Ogni volta che un cliente è inserito in una *route*, l'algoritmo deve cominciare dall'inizio siccome le combinazioni che non erano disponibili fino a prima dell'inserimento potrebbero essere disponibili ora. Questo approccio crea una *route* alla volta, richiedendo di ripassare la coda di *savings* più di una volta. Il codice allegato Listing1 descrive come è stato implementato l'algoritmo sequenziale di Clarke and Wright. La riga 2 e 3, importano il valore di capacità per ogni veicolo, e la rispettiva domanda di ogni cliente. Alla riga 4, la lista di savings viene calcolata attraverso la funzione *calculateSavings* per ogni arco presente nel grafo. Alla riga 8 l'istruzione di controllo *while*, continua a eseguire il codice al suo interno fino a quando tutti i clienti sono assegnati ad una *route* e la lista di *savings* contiene elementi di risparmio. All'interno del *while* alla riga 9 viene estratta la prima coppia di risparmi,alla riga 10 si inizializza il puntatore *k*, che servirà a scorrere la lista *savings* durate la costruzione della strada *route*, inizializzata poco più sotto alla riga 12, dopo l'istruzione di controllo *if* che controlla che nessun componente della coppia sia già servito da un'altra *route* e che nessuna *route* sia stata già creata. Se il controllo risulta vero allora si passa all'inizializzazione della strada con la coppia selezionata *pair* e quindi si prosegue scorrendo il resto della lista *savings* alla riga 14. Al suo interno i controlli alla riga 18 e 23, garantiscono al mutuale esclusività dei componenti della coppia all'interno della strada che viene estesa dalle rispettive istruzioni 20 e 25; in 21,22 *k* viene azzerato e *save* tolto dalla lista *savings*,per ricominciare dall'inizio della lista e scorrere eventuali coppie che adesso potrebbero essere collegate, grazie ai nuovi clienti. Le istruzioni solo uguali in 26,27. Le istruzioni di controllo 19 e 24, garantiscono che il cliente aggiunto rispetti i vincoli di capacità del veicolo, in caso questo. Infine in 30 si aggiunge la *route* alla soluzione e 31 aggiorna la lista *savings* con la nuova distanza della coppia *pair*. In 32 si ritorna la soluzione.

- L'approccio **parallelo** crea più *routes* contemporaneamente.Quando una coppia di clienti è presa in considerazione, viene confrontata con tutte le *routes* che sono già state generate fino a quel momento. Se la coppia viene inserita in una route, tutte le *routes* generate fino ad ora vengono confrontate per trovare possibili *route* da unire ulteriormente con attraverso un *merge*. Altrimenti,se la coppia di clienti non corrisponde con nessuna *route*,allora una nuova *route* viene create, questa nell'approccio sequenziale invece verrebbe scartata. Nella versione parallela la lista dei *savings* viene visitata una sola volta per tutta la sua lunghezza. Nel codice Listing2 Dalla riga 2 alla riga 6 le istruzioni sono identiche al codice Listing 1 già presentato. Dalla riga 7 invece si differenzia, in quanto per ogni cliente viene creata una strada: deposito $\rightarrow$ cliente $\rightarrow$ deposito. In 0 il ciclo *for* si occupa di scorrere la lista. Al suo intenro a riga 10 si estrae la tupla che forma la coppia di clienti che appartengono al risparmio. 11,12 e 13 inizializzano a *null* tre di appoggio contenitori *route*. Una volta estratti i clienti, si scorrono tutte le *routes* come in 14, il controllo in 15 si controlla se una *route* passa per *i*, quando questa viene trovata, viene allocata sia in *routeA* in 16, la stessa cosa viene fatta per j in 18, 19 con *routeB*. Il senso di questa doppio allocazione viene si spiega nella righe successive: in 22 si verifica se *routeA* e *routeA* hanno valori diversi, siccome tutti i clienti hanno una strada assegnata, *routeA* e *routeB* saranno sicuramente diverse da *null* di conseguenza si controlla in 21 se i e j provengono da strade diverse. In caso positivo si controlla se i componenti della *routeA* e i componenti della *routeB* rispettano i vincoli di capacità, come descritto in riga 22.Se anche questo controllo viene passato con successo, allora si passa al metodo fondamentale di questo algoritmo, *il Merge* riga 23. Questo metodo si preoccupa di capire se i due clienti presenti nelle due diverse *route*, stanno in testa o in coda alla *route*, di conseguenza unisce le due strade importando i clienti nella *routeA* o nella *routeB* a seconda dell'ordine dato da *save*. In 24 si rimuove da *routes* la *route* che è stata copiata e in 26 si ritorna la soluzione.

La differenza sostanziale tra i due metodi è la seguente: il metodo sequenziale tende a generare una *route* unica molto grande, e le *route* seguenti invece sono più piccole. Mentre il metodo parallelo tende a formare più strade larghe e quindi riduce il numero finale di *routes*

```
def Clarke&Wright_Sequenziale(graph):
  Capacity = graph.getCapacity()
  Demands:list = graph.getDemands()
  list savings = []calculateSavings (i,j)  ∀ i,j ∈ V | i ≠ j ∧ i ≠ 0]
  savings.sort(descending)
  list routes = []

  while(∑_(r)∈R ∑_(v)∈V (v ∈ r) ≠ | V | and len(savings)>0):
    pair = savings.popFirst()
    k=0
    if(pair not served yet and No route is selected):
      route = Route.create(pair)

      while(k< lenghtOf(savings)):
        save = savings[k]
        i , j = tuple(save)
        k=k+1

        if(route already served i):
          if(checkCustomer(j)):
            route.AddCustomer(j)
            k=0
            savings.remove(save)
        elif (route already served j):
          if(checkCustomer(i)):
            route.AddCustomer(i)
            k=0
            savings.remove(save)

      routes.append(route)
      savings.update(route)

  return routes
```

Listing 1: Implementazione sequenziale di Clarke and Wright

```
def Clarke&Wright_Parallelo(graph):
  Capacity = graph.getCapacity()
  Demands:list = graph.getDemands()
  list savings = [calculateSavings (i,j)  ∀ i,j ∈ V | i ≠ j ∧ i ≠ 0]
  savings.sort(descending)
  list routes = []
  routes.append([0 - i - 0])  ∀ i ∈ V | i ≠ 0

  for save in savings:
    i , j = tuple(save)
    routeA = null
    routeB = null
    routeSelected = null
    for route in routes
      if route.served(i):
        routeA = route

      elif route.served(j)
        routeB = route

    if rotues.index(routeA) != routes.index(routeB):
      if routeA.capacity() + routeB.capacity() <= Capacity:
        Merge(routeA,RouteB)
        routes.remove(RouteA or RouteB)

  return routes
```

Listing 2: Implementazione parallela di Clarke and Wright

3.3 Metodi di Raggruppamento (Clustering)

I metodi di raggruppamento chiamati anche clustering method, si differenziano dagli altri metodo perchè la risoluzione del probelma è divisi a in due fasi.

- **Fase A** Tutti i nodi sono divisi in gruppi (cluster) utilizzando un algoritmo di raggruppamento.

- **Fase B** Per ogni gruppo di nodi si utilizza un algoritmo di ottimizzazione per trovare il miglior routing, per servire tutti i nodi del gruppo.

Ecco perchè questi algoritmi sono conosciuti anche come algoritmi a 2 fasi. In questo elaborato vengo presentati i seguenti metodi di raggruppamento a due fasi:

- Cluster First,Route Second: Fisher and Jaikumar.
  - K centroid selector algoritms : Random
  - K centroid selector algoritms : Radar Radius
  - General Asignement Problem solver
  - Routing algoritms: Nearest Neighborn
  - Routing algoritms: Dijkastra
- Route First,Cluster Second: using an auxiliary graph and Dijkastra

3.3.1 Algoritmo di Fisher and Jaikumar

L'algoritmo di *Fisher and Jaikumar* si basa su un'assunzione il numero di veicoli per servire tutti i clienti è dato da input. Inoltre questo metodo non è applicabile direttamente a VRP che hanno vincoli sulla distanze percorribile da parte del veicolo. L'algoritmo si svolge in due fasi come abbiamo già detto sopra, in particolare.

- Fase 1 - clustering
  1. Selezione dei centroidi: scegli un vertice $k \in V$ t.c. $\mid K \mid =$ *numero veicoli.*
  2. Calcola il costo di inserimento $c_{i,k}$ del clienti i al cluster k:
     $a_{i,k} = min(c_{o,i} + c_{i,k} + c_{k,0} + c_{0,k} + c_{k,i} + c_{i,0}) - (c_{0,k} + c_{k,0})\ \forall k \in K.$
  3. Assegnamento Generale: risolvere un ***G****eneralized* ***A****ssignement* ***P****roblem* con i costi $c_{i,k}\ \forall i \in V, \forall k \in K$, le richieste dei clienti D e la capacità prestabilita C.
- Fase 2 - Routing: risolvere un TSP per ogni cluster.

3.3.2 Fase 1.1 - Selezione dei Centroidi

In questo elaborato per quanto riguarda la Fase 1.1 sono stati implementati due metodi: La selezione casuale e la selezione utilizzando un algoritmo chiamato dall'autore Radar Radius.

- Selezione casuale dei centroidi: vengono scelti casualmente un numero di centroidi uguale al numero di veicoli.
- Selezione con Radius Radar Frontier:

$$max \sum_{i \in K} c_{i,0} \tag{10}$$

$$c_{i,j} \geq argmax(c_{n,m})/2 \quad \forall i, j \in K \wedge n, m \in V \tag{11}$$

$$d_k \geq C/2\ \forall k \in K \tag{12}$$

I centroidi vengono individuati con la lettera K,in quanto il numero di centroidi corrisponde al numero di veicoli. La funzione di obbiettivo 10 massimizza la distanza dei centroidi dal deposito *0*. Il vincolo 11 richiede ai centroidi selezionati, di rispettare una distanza che sia uguale o superiore alla metà della distanza massima tra centroidi e clienti, ovvero la distanza massima tra un centroide e qualsiasi altro cliente nel grafo. 12 vincola i centroidi ad avere una richiesta che sia maggiore della metà della capacità del veicolo, in quanto i clienti più onerosi in termini di richiesta hanno una probabilità maggiore di risiedere in *route* diverse. I vicoli 11 e 12 durante l'esecuzione vengono rilassati ad ogni esplorazione dello spazio delle soluzioni, in modo da avere più clienti candidabili. In Table 1 viene rappresentata la selezione dei primi due centrodi, da parte dell'algoritmo.

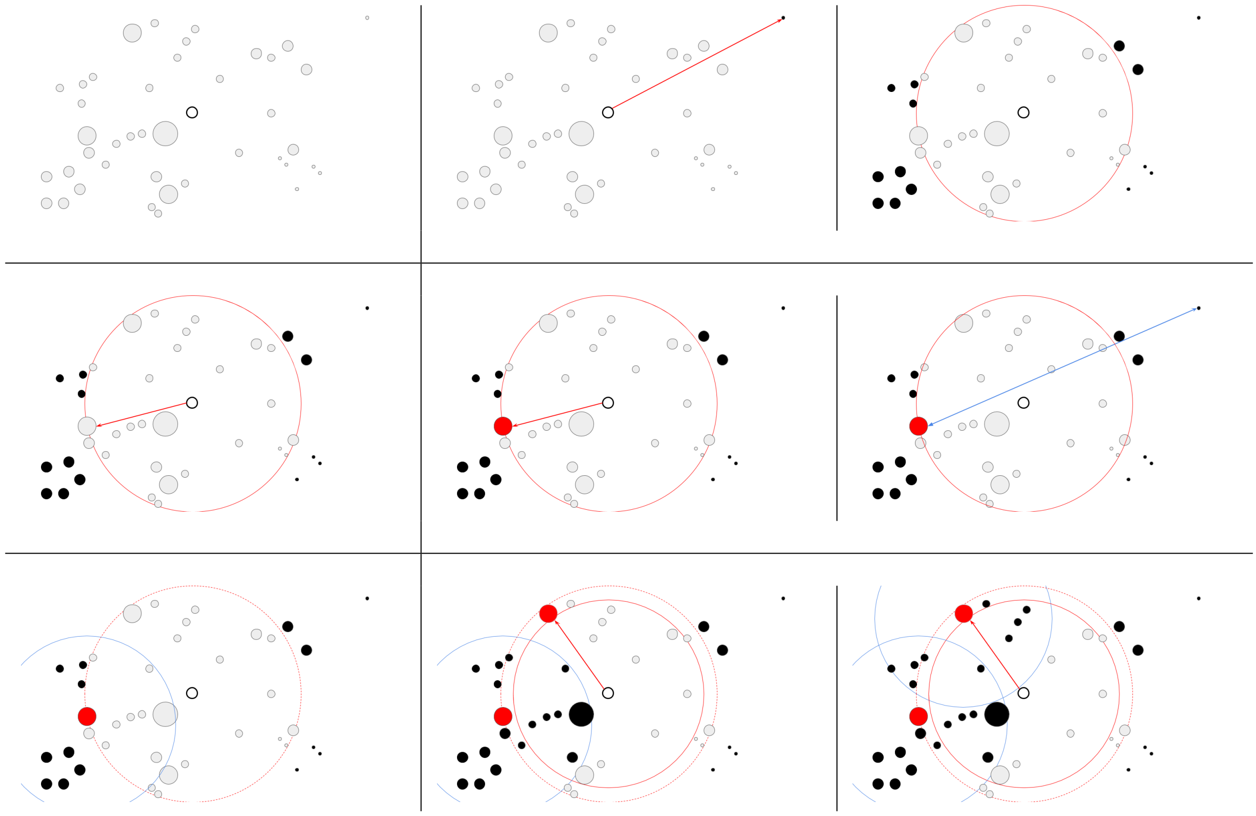


Table 1: In alto a sinistra il deposito al centro bianco e contorno nero, con tutti clienti attorno da servire di colore grigio. In alto al centro l'algoritmo traccia il raggio del utilizzando la distanza massima tra il deposito e il cliente più lontano, si definisce così il radar del deposito. Il raggio diminuisce in quanto i candidati che rispettano il vincolo 12 si trovano più vicino al deposito, i clienti che stazionano sulla frontiera diventano i possibili candidati, mentre gli esterni vengono esclusi,i clienti interni devono ancora essere esplorati . La figura centrale mostra il primo candidato sulla frontiera, selezionato come centroide con la distanza massima dal deposito, nella figura seguente, l'algoritmo traccia la distanza massima tra il centroide e tutti i clienti. Questa serve per tracciare il raggio del radar del centroide selezionato. Il raggio è scalato dividendolo per il numero di veicoli. Di conseguenza nella figura in basso a sinistra tutti i clienti all'interno del raggio del centroide sono esclusi dalla lista dei possibili candidati.Il raggio del radar del deposito diminuisce e dando la possibilità ad altri clienti di diventare condidati se e solo se rispettano 12. Viene quindi selezionato il nuovo centroide sulla frontiera e tracciato il raggio che comporrà il suo radar, come in figura in basso a destra.

```
def RadarRadiusFrontier(graph,n_vehicles):
  Dimension = graph.getDimension()
  Capacity = graph.getCapacity()
  Demands:list = graph.getDemands()
  scaleDown = 2
  depotDistance:list = [c_{v,0} ∀v ∈ V]
  seeds = []
  scannerRadius = DepotDistance
  maxCoverDistance = 0

  while (len(seeds)< n_vehicles):
    candidates = []
    if ( ∑_{i∈V} d_i ≠ Dimension − 1):
      for i in range(Dimension):
        if(demand[i] ≥ capacity/scaleDown)
          candidates.append(i)
    else:
      candidates = np.arange(Dimension)

    for c in candidates:
      if(depotDistance[c] ≥ argmax(scannerRadius)):
        if(seeds is empty):
          seeds.append(c)
          scannerRadius[c] = 0
          maxCoverDistance = graph.getArgMaxNodeDistance(c)
          break
```

```
27
28        for v in seeds:
29          if((graph.getValue(c,v) >= maxCoverDistance/n_vehicles) and (c not in v)):
30            seeds.append(c)
31            scannerRadius[c] = 0
32            if(maxCoverDistance < graph.getArgMaxNodeDistance(c)):
33              maxCoverDistance = graph.getArgMaxNodeDistance(c)
34            break
35          else:
36            scaleDown = scaleDown + 0.5
37            scannerRadius[c]=0
38            continue
39
40    if ( ∑_{i∈V} d_i ≠ Dimension − 1):
41      print("Decrease Radius")
42      scannerRadius[np.argmax(scannerRadius)] = 0
```

Listing 3: RadarRadiusFrontier selettore di centroidi per Fisher and Jaikumar

L'algoritmo descritto in Listing 3, presenta quanto già descritto formalmente sopra. Dalla riga 2 alla 4 si riportano le informazioni acquisite dal grafo, in 5 si inizializza il fattore di scala applicato ad ogni ciclo, che verrà aggiornato rilassando così il vincolo 12. In 6 si estraggono le distanze di ogni cliente dal deposito,in 7 si inizializza seeds, il contenitore di centroidi. In 8 si crea una copia di *DepotDistance* andando così a creare *ScannerRadius* che sarà utilizzato poi per ridurre il raggio del radar del deposito. 9 *MaxCoverDistance* rappresenta la distanza massima tra centroidi e clienti. Si inizia con la riga 11 dove il while termina solo quando avrà trovato un numero di centroidi uguale al numero di veicoli. In 13 si effettua un importante controllo: alcune istanze presenti nel dataset fornito in questo elaborato hanno una richiesta uguale per tutti i clienti spesso apri a 1. Di conseguenza se la richiesta è uguale per tutti i clienti, il vincolo sulla capacità può essere evitato, immettendo tutti i clienti come a riga 18. In caso contrario si aggiungono solo quelli che soddisfano il vincolo 12 rilassato da scaleDown, righe 14,15,16. In 20 si scorrono tutta la lista di candidati, in 21 accetta solo i candidati con una distanza maggiore o uguale alla distanza massima tra i componenti di *ScannerRadius*. Se troviamo un candidato che rispetta entrambi i controlli,in caso seeds sia vuoto possiamo aggiungerlo direttamente 22 - 26, azzeriamo il suo valore in *ScannerRadius* e aggiorniamo *MaxCoverDistance* con la distanza massima tra il candidato selezionato e il cliente più distante. Altrimenti scorriamo la lista di seeds, se il candidato è distante almeno *MaxCoverDistance* diviso il numero di veicoli e il candidato non è già presente in seeds, lo aggiungiamo a seeds e aggiorniamo di conseguenza *MaxCoverDistance* righe 28 - -34.In caso contrario aumentiamo il fattore scaleDown e azzeriamo il candidato in *ScannerRadius*. Le righe 40,41,42, vengono eseguire nel caso le richieste dei clienti siano diverse, rilassa il vincolo 11 aumentando il bacino di candidati.

3.3.3 Fase 2.2 e 2.3 General Assignement Problem

Una volta individuati i centroidi si passa alla Fase 2.2. L'algoritmo deve assegnare ongni cliente ad ogni centroide. Per fare questo l'algoritmo risolve un problema di asseganemtno generale.Si richiede di trovare un assegnamento dei clienti ai cluster tale da non superare la capacità del veicolo assegnato al cluster. Il numero di chilometri totali percorsi dalla soluzione va minimizzato. Più formalmente:

$$x_{i,k} = \begin{cases} 1 : sei\ assegnato\ al\ cluster\ k \\ 0 : altrimenti \end{cases}$$

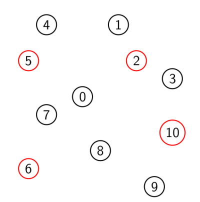
$$min \sum_{i \in V} \sum_{k \in K} x_{i,k} \cdot a_{i,k} \tag{13}$$

$$\sum_{k \in K} x_{i,k} = 1\ \forall i \tag{14}$$

$$\sum_{i \in V} d_i \cdot x_{i,k} \leq C\ \forall k \tag{15}$$

$$x_{i,k} \in \{0,1\} \tag{16}$$

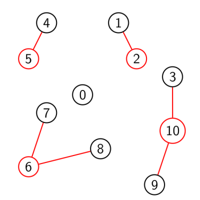
Clustering



4 vehicles
seed vertices: 2,5,6,10
compute a_{ik} values

a_{ik}	2	5	6	10
1	3.1	6.3	8.9	5.8
2	0.0	6.0	7.1	2.7
3	3.7	9.6	9.5	2.7
4	6.3	3.1	7.5	8.3
5	6.0	0.0	4.6	7.2
6	9.9	7.4	0.0	7.9
7	4.5	1.8	0.4	3.9
8	4.9	6.0	2.3	1.9
9	9.9	12.7	8.5	4.2
10	6.3	10.7	8.6	0.0

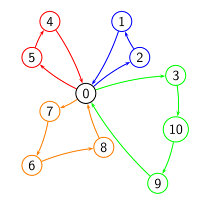
Clustering



solve the GAP
(e.g. with Excel solver)

x_{ik}	2	5	6	10
1	1	0	0	0
2	1	0	0	0
3	0	0	0	1
4	0	1	0	0
5	0	1	0	0
6	0	0	1	0
7	0	0	1	0
8	0	0	1	0
9	0	0	0	1
10	0	0	0	1

Routing (e.g. nearest neighbor)



c_{ij}	0	1	2	3	4	5	6	7	8	9	10
0	0.0	4.5	3.6	5.1	4.5	3.6	5.0	2.2	3.2	6.4	5.4
1	4.5	0.0	2.2	4.2	4.0	5.4	9.4	6.4	7.1	9.2	6.7
2	3.6	2.2	0.0	2.2	5.4	6.0	8.5	5.8	5.4	7.1	4.5
3	5.1	4.2	2.2	0.0	7.6	8.1	9.4	7.3	5.7	6.1	3.0
4	4.5	4.0	5.4	7.6	0.0	2.2	8.1	5.0	7.6	10.8	9.2
5	3.6	5.4	6.0	8.1	2.2	0.0	6.0	3.2	6.4	9.9	8.9
6	5.0	9.4	8.5	9.4	8.1	6.0	0.0	3.2	4.1	7.1	8.2
7	2.2	6.4	5.8	7.3	5.0	3.2	3.2	0.0	3.6	7.2	7.1
8	3.2	7.1	5.4	5.7	7.6	6.4	4.1	3.6	0.0	3.6	4.1
9	6.4	9.2	7.1	6.1	10.8	9.9	7.1	7.2	3.6	0.0	3.2
10	5.4	6.7	4.5	3.0	9.2	8.9	8.2	7.1	4.1	3.2	0.0

Table 2: Un esempio di risoluzione GAP una volta scelto i centroidi. A sinistra vengono calcolati i costi di inserimento per ogni vertice in ogni cluster. In centro avviene l'assegnamento, associando ogni nodo al cluster che ha un costo di inserimento minore. A destra si prosegue con il routing di ogni cluster: in questo caso si usa l'algoritmo Nearest Neighbor partendo dal deposito e proseguendo verso il nodo più vicino appartenente al cluster, si prosegue per vicinanza collegando il resto dei nodi in ogni singolo cluster.

```
1  def GAPsolver(graph,Kclusters):
2    Dimension = graph.getDimension()
3    Capacity = graph.getCapacity()
4    Demands:list = graph.getDemands()
5    Nclusters = len(Kclusters)
6    allocCosts:[list,list] = [allocCosts_{i,k} = 0 ∀i ∈ V ∧ k ∈ Nclusters]
7    clusterAssignement:list = []
8    clusterDemand:list =  [clusterDemand_k = 0  k ∈ Kclusters]
9
10   allocCosts:list = [allocCosts_{i,k} = calculateIns(i,k) ∀i ∈ V ∧ k ∈ Nclusters]
11
12   i=1
13   for alloc in allocCosts:
14     for k in Kclusters:
15       if(clusterDemand[argmin(alloc)] + Demands[i] <= Capacity):
16         clusterDemand[argmin(alloc)] =+ Demands[i]
17         clusterAssignment.append(np.argmin(alloc))
18         i++
19         break
20       else:
21         alloc[argmin(alloc)] = np.inf
22         print("Cluster Overloaded")
23
24   if(len(clusterAssignment)<Dimension-1):
25     print("Solution is Invalid")
26     return -1
27   else:
28     return clusterAssignment
```

Listing 4: General Assignement Problem solver

Listing 4 presenta lo pseudocodice implementato del GAP solver implementato. Alla riga 1 la firma del metodo riceve il graph e i centroidi selezionati con le tecniche descritte nella sezione precedente.Dalla riga 2 alla riga 4 si recuperano le informazioni sul grafo, in 5 si estrae il numero di cluster. *allocCosts* è la lista che conterrà i costi di allocazione di ogni cliente per ogni cluster, alla riga 5 viene inizializzata a 0 e ha dimensione uguale a *Dimension* Ncluster*. In 7 viene inizializzata a vuoto la lista *clusterAssignement* che conterrà i nodi assegnati ad ogni cluster. In 8 si inizializzano a 0,il totale di domande soddisfatte per ogni cluster. In 10 vengono calcolati i costi di inserimento di ogni nodo per ogni cluster: *calculateIns(i,k)* calcola i costi come è stato già descritto nella sezione **3.3.1**. In 12 si inizializza il puntatore *i* che servirà per puntare alla domanda del nodo corrispondente in *Demands*. 13 e 14 si occupano di scorrere ogni riga di *allocCosts* e al suo interno ogni cluster di *Kcluster*. Il controllo a riga 15, assicura che il cluster con costo di inserimento minimo per il nodo, non superi la capacità prestabilita. Se il controllo viene superato con successo alla riga 16 si aggiorna la domanda totale soddisfatta dal cluster, in 17 si assegan al nodo il cluster e in 18 si incrementa di uno il puntatore. Siccome il nodo è stato assegnato con 19 si passa alla prossima riga di *allocCosts*. Altrimenti, se il controllo a riga 15 fallisce, la riga 21 pone ad infinito il valore minimo calcolato per quel nodo, in modo che nel ciclo seguente si prenda in considerazione il secondo valore minimo calcolato tra quel nodo e un altro cluster, questo viene ripetuto per k in Kclusters. Se alla fine dei due cicli, la lunghezza della lista *clusterAssignement* non è pari al numero di nodi escluso il deposito, vuol dire che l'allocazione clienti-clusters non è andata a buon fine. Quindi la soluzione è invalida, sarà necessario aumentare il numero di veicoli e quindi si ritorna -1 . Altrimenti se tutto va bene, si ritorna l'assegnamento attraverso la lista *clusterAssignement*.

3.3.4 Fase 2 - Routing

Ora che gli assegnamenti sono stati decisi, ci dobbiamo preoccupare di collegare i nodi all'interno di ogni cluster, in modo da formare le *routes* necessarie per descrivere la soluzione. La descrizione formale di questo problema è simile a quella già descritta nella sezione 1.1: La funzione obiettivo che descrive questa fase è la seguente:

$$\min \sum_{k \in K} \sum_{(i,j) \in A} c_{i,j} \cdot M_{i,j}^k \tag{17}$$

$$M_{i,j}^k \in \{0, 1\} \tag{18}$$

A differenza di ciò che è descritto nella sezione 1.1 i vincoli sono già stati rispettati dalla lista di assegnamento fornita dal GAP solver.

In questa elaborato sono state utilizzate due tecniche per implementare il routing:

- **Nearest Neighbourn**: Partendo dal deposito ci si collega al nodo successivo, si prosegue fino a quando tutti i nodi sono stati collegati.

```
def Routing_NearestNeighbourn(graph,clusterAssignement,Kclusters,saveFolder):
  rutes = []
  Demands = graph.getDemands()
  Capacity = graph.getCapacity()

  for k in Kclusters:
    cluster =  []
    for i in range(len(clusterAssignment)):
      if(clusterAssignment[i] == k ):
      cluster.append(i+1)

    appoRoute =  Route(Capacity)
    appoRoute.addCustomer(0,0,False)

    while(len(cluster)>0):
      prevnode = appoRoute.getCustomers()[len(appoRoute.getCustomers())-1]
      distPrevNode = graph.getValue(prevnode,[c for c in cluster if c not in appoRoute.
    getCustomers() ])
      nearestN = cluster[np.argmin(distPrevNode)]
      if nearestN not in appoRoute.getCustomers():
        appoRoute.addCustomer(nearestN,demand[nearestN],False)
        cluster.remove(nearestN)

    appoRoute.addCustomer(0,0,False)
    appoRoute.printRoute("Route cluster:" +str(k))
    routes.append(appoRoute)

  return routes
```

Listing 5: NearestNeighbourn

- **Modified Dijkastra**:

References

[1] Associazione Nazionale Filiera Industria Automobilistica ANFIA. Dossier trasporto merci su strada, 2019.

[2] G.B. Dantzig and J.H. Ramser. The truck dispatching problem. *Management Science, Vol. 6, No. 1 (Oct., 1959), pp. 80-91*, 2008.

[3] G. Reinelt. Tsplib95. *Institut fur Angewandte Mathematik*, 1990.

[4] J.W Clarke, G. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *perations Research, Vol. 12, 1964, pp. 568-581.*

[5] Jens Lysgaard. Clarke and wright's savings algorithm. *Department of Management of Science and Logistic, the Aarhus School of Business.*

[6] R. Kawtummachai T.Pichpibula. An improved clarke and wright savings algorithm for the capacitated vehicle routing problem. *Department of Management of Science and Logistic, the Aarhus School of Business.*

[7] C. Hjorring D. M. Ryan and F. Glover. Extensions of the petal method for vehicle routing. *Journal of the Operational Research Society,44:289-296.*

[8] Alias S.M. Shamsuddin G.W. Nurchahyo, R.A. and M.. md. Sap. Sweep algorithm in vehicle routing problem for public transport. *Journal Antarabangsa (Teknologi Maklumat).*

[9] M.L. Fisher and R. Jaikumar. A generalized assignment heuristic for vehicle routing. *Journal of the Operational Research Society,44:289-296.*

[10] D.E: Goldberg. Genetic algorithm in search, optimization and machine learning". *New York Addison-Wesley.*