Courtesy Bus Problem

Giulia Calanca

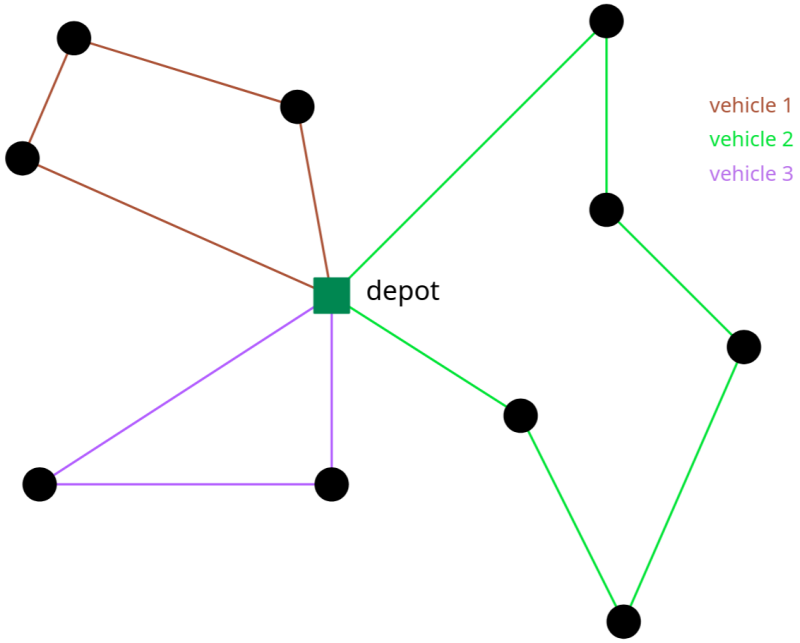
A.A 2021/2022

Contents

1 Il problema in letteratura

Il problema di vehicle routing (VRP) è un classico problema di ottimizzazione combinatoria e programmazione intera. Introdotto da Dantzig and Ramser (1959) [1] può essere visto come una generalizzazione del problema del commesso viaggiatore (TSP, Travelling salesman problem). In VRP si richiede di trovare l'insieme ottimo di percorsi che un gruppo di veicoli deve percorrere affinché tutti i clienti vengano serviti. Esistono numerose varianti del VRP e altrettante implementazioni commerciali di casistiche specifiche. Trovare una soluzione ottima al problema è NP-hard; per questo motivo nelle implementazioni reali, dove dimensione e frequenza di applicazione non permettono di raggiungere l'ottimo, si ricorre solitamente all'uso di euristiche. In figura 1 è rappresentato un esempio di istanza di VRP: il quadrato verde al centro è il deposito da dove partono e ritornano i veicoli, mentre i cerchi neri sono i clienti da servire.

[1] G. Dantzig, J. Ramser, The truck dispatching problem, Management Science, 6 (1959), pp. 80-91



Come già menzionato esistono parecchie varianti del problema VRP nelle quali cambiano, o si aggiungono, vincoli di capacità, tempo, punti di partenza, numero di viaggi, etc. La variante del problema implementata in questo progetto può essere considerata come una sotto-variante del CVRPTW (i.e. *Capacitated Vehicle Routing Problem with Time Windows*) nella quale i veicoli hanno una capacità limitata e sono presenti vincoli di tempo per la consegna.

1.1 Modellazione del problema

Esistono alcune modellazioni del problema - nelle sezioni successive ne vengono descritte due tra le più conosciute: la modellazione compatta, con un numero di variabili e vincoli polinomiale e la modellazione estesa, ovvero con variabili e vincoli in numero esponenziale.

1.1.1 Modello CVRP

Una possibile formulazione del problema CVRP tramite ILP è la seguente:

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

$$\text{sub. to}$$

$$\sum_{i \in V} \sum_{k \in K} x_{ijk} = 1 \ \forall j \in V \setminus 0 \tag{1}$$

$$\sum_{j \in V} \sum_{k \in K} x_{ijk} = 1 \ \forall j \in V \setminus 0 \tag{2}$$

$$\sum_{i \in V} \sum_{k \in K} x_{ihk} - \sum_{j \in V} \sum_{k \in K} x_{hjk} = 0 \ \forall k \in K, h \in V \tag{3}$$

$$\sum_{i \in V} q_i \sum_{j \in V} x_{ijk} \leq Q_k \ \forall k \in K \tag{4}$$

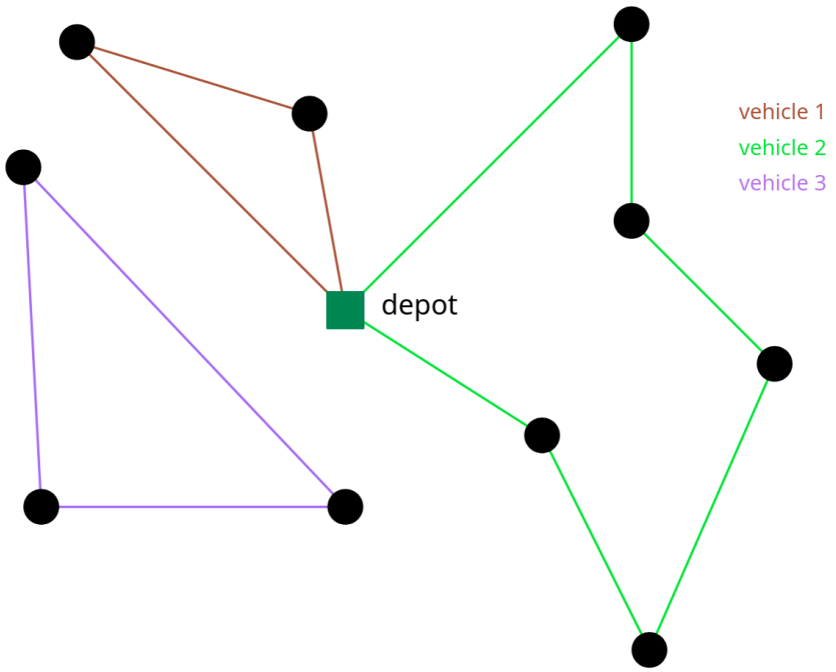
$$\sum_{ijk} x_{ijk} = |S| - 1 \ \forall S \subseteq P(V), 0 \notin S \tag{5}$$

$$x_{ijk} \in 0, 1 \tag{6}$$

Con:

- V l'insieme dei nodi (clienti e depot), con depot = 0
- K l'insieme dei bus
- c_{ij} costo dell'attraversamento dell'arco (i, j)
- $x_{ijk} = 1$ se il bus k attraversa l'arco (i, j), $x_{ijk} = 0$ altrimenti
- Q_k la capacità del bus $k \in K$

I vincoli (1) e (2) garantiscono che ci siano rispettivamente un solo veicolo in entrata ed uno solo in uscita per ogni nodo (i.e. cliente). Il vincolo (3) garantisce che il veicolo in entrata coincida con quello in uscita per un determinato nodo, mentre il (4) garantisce sia rispettato il vincolo di capacità dei veicoli. Il vincolo (5) invece riguarda l'eliminazione dei *subtour*, ovvero dei percorsi chiusi che pur rispettando i vincoli da (1) a (4) non generano una soluzione accettabile (esempio in figura 1.1.1)



Si indica con S il subtour e con $|S|$ il numero dei nodi che lo compongono. Per tutti i subtour nell'insieme dei path $P(V)$ che non contengono il depot (indicato con 0) il numero di archi deve coincidere con la cardinalità del subtour $|S|$ meno 1. Il numero dei subtour è esponenziale nel numero di nodi del problema, per questo motivo spesso si ricorre a tecniche "lazy", ovvero si agisce sui subtour solo nel caso siano presenti nella soluzione.

1.1.2 Modello compatto CVRPTW

La modellazione del CVRPTW prevede gli stessi vincoli del CVRP con l'aggiunta dei vincoli sulle time windows e la conseguente esclusione di quelli che regolano i subtour. I vincoli temporali di fatto eliminano la formazione di subtour, rendendo dunque quei vincoli non più necessari. Sia:

- $[a_i, b_i]$ la finestra di tempo nella quale un veicolo deve visitare il i
- s_{ik} l'istante in cui inizia il servizio del veicolo k in i
- t_{ij} il tempo di attraversamento dell'arco (i, j)
- M, detta big-M, una costante dal valore tendente a ∞.

Allora i vincoli addizionali sono:

$$a_i \leq s_{ik} \leq b_i \tag{7}$$

$$s_{ik} + t_{ij} - M(1 - xijk) \leq s_{jk} \tag{8}$$

Essi garantiscono il rispetto delle time windows (vincolo (7)) e la consecutività dei tempi di arrivo nei nodi (vincolo (8)).

1.1.3 Modello esteso CVRPTW

Un'altra formulazione del problema CVRPTW è possibile tramite la column generation. Si immagini una tabella nella quale ogni riga rappresenta un cliente/nodo, mentre ogni colonna rappresenta un cammino ammissibile.

	89	76	99	45	32	...
1	1	1	1	0	0	...
2	0	1	1	0	1	...
3	0	0	0	0	0	...
4	1	0	1	1	0	...
5	1	0	0	0	0	...

Nell'esempio la prima colonna rappresenta il percorso che attraversa i nodi $1, 4, 5$ (non necessariamente in questo ordine) ed ha un costo di 89 unità, la seconda colonna passa per i nodi $1, 2$ e costa 76, e così via per tutti i possibili percorsi. Il problema allora diventa trovare i cammini che, attraversando tutti i nodi, minimizzano il costo totale della route. Sia K l'insieme di tutte le colonne (ovvero i possibili path) , $a_{ik} = 1$ se il percorso definito dalla colonna k serve il cliente i e c_k il costo di quel percorso, allora il problema si descrive in questo modo:

$$min \sum_{k \in K} c_k x_k$$

$$\text{sub. to}$$

$$\sum_{k \in K} a_{ik} x_k = 1 \ \forall i$$

$$x_k \in 0, 1$$

La formulazione del VRP così espressa diventa un problema di *set partitioning* ed è di per sè semplice. La difficoltà sta nel fatto che le variabili sono in numero esponenziale e nel trovare i cammini ammissibili per i quali devono essere rispettati i vincoli temporali e di capacità. Nel sottoproblema dunque

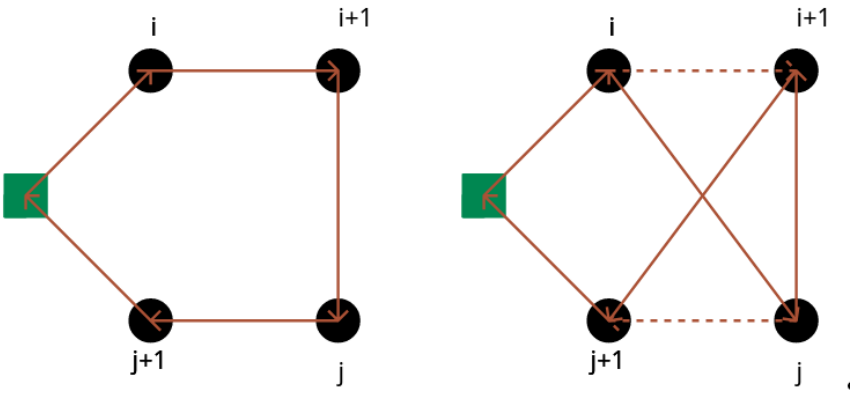
si richiede di risolvere il problema noto come *elementary shortest path with time windows and capacity constraints* (ESPPTCC) nel quale si cerca il cammino minimo che tenga conto delle finestre temporali, delle capacità e che sia elementare, ovvero visiti ogni vertice una sola volta.

1.2 Metodi di risoluzione

I metodi per risolvere il VRPTW in maniera esatta possono essere classificati in tre categorie: basati sul rilassamento lagrangiano, column generation e dynamic programming. Come già anticipato trovare una soluzione per il VRPTW è NP-hard, dunque la dimensione dei problemi risolvibili in modo ottimo è molto limitata. A causa della sua elevata difficioltà computazionale in tanti si sono dedicati allo studio di euristiche il cui scopo è trovare una soluzione approssimata al VRPTW. Un primo approccio possibile è quello di utilizzare un algoritmo costruttivo per generare una soluzione accettabile: ad ogni iterazione un nodo (cliente) non visitato viene aggiunto al cammino. La costruzione della route può essere fatta in maniera sequenziale, un nodo alla volta, o parallela, un nodo alla volta per ogni route. L'inserzione sequenziale fu proposta da Solomon [2], il quale pensò ad un algoritmo strutturato in due fasi, una dove la scelta del cliente ricarda sul nodo più vicino e l'altra dove si seleziona un cliente sulla base del concetto di *maximum savings* teorizzato da Clark e Wright[3]. Esistono inoltre degli algoritmi di miglioramento dei cammini che iterativamente modificano la soluzione corrente sfruttando tecniche di *local search*. Queste tecniche si basano sull'utilizzo *neightbourhood*, un insieme di soluzioni raggiungibili a partire da quella attuale tramite l'utilizzo di "mosse". Si definisce mossa ogni tipo di spostamento di nodi o archi applicabile in un'iterazione. Esempi di mosse sono: 2-opt, 3-opt o OR-opt nelle quali si scambiano rispettivamente 2, 3 o più archi, l'inserzione, ovvero lo spostamento di un nodo da una posizione all'altra e la swap, mossa nella quale due nodi vengono scambiati di posto. Si veda in figura 1.2 l'esempio dell'applicazione della mossa 2-opt nella quale gli archi $(i, i+1)$ e $(j, j+1)$ vengono scambiati.

[2] M.M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. Operations Research, 35:254–265, 1987.

[3] G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. Operations Research, 12:568–581, 1964.



L'obiettivo della local search è quello di arrivare ad un ottimo: tentando una mossa dopo l'altra si arriva ad una soluzione non più migliorabile, dunque si arresta l'algoritmo. Il principale limite della local search sta nel fatto che spesso si arresta la ricerca in un ottimo locale - un modo semplice per ovviare al problema a è quello di ricorrere al *multistart*, ovvero eseguire la local search più volte a partire da soluzioni iniziali differenti. Oltre al multistart esistono numerose tecniche di risoluzione approssimata che cercano di ampliare lo spazio di ricerca della soluzione - esse vengono chiamate *metaeuristiche*. Una delle prima metaeuristiche fu introdotta da Glover [4] con il nome di *tabu search*. In questo algoritmo ad ogni iterazione viene esplorata la *neighbourhood* della soluzione e la miglior soluzione trovata viene selezionata come soluzione corrente. Per fare in modo che l'algoritmo si possa allontanare da un ottimo locale, la soluzione corrente viene aggiornata con la miglior soluzione trovata nella *neighbourhood* anche se non è effettivamente migliore di quella attuale. Per evitare cicli viene proibita la visita di soluzioni recenti - questo concetto è implementato tramite *tabu list*, ovvero una lista contenente mosse "proibite". Esistono dei criteri per andare contro le regole della tabu list, detti *aspiration criteria*, che permettono di fermare la tabu search dopo, ad esempio, un numero di iterazioni senza miglioramento. Tra le altre metaeuristiche più conosciute si elencano ancora: algoritmi genetici, simulated annealing, ALNS (Adaptive Large Neighbourhood Search).

[4] Glover, F., 1989. Tabu search. Part 1, ORSA. Journal on Computing 1 (3), 190–206.

2 Il Courtesy Bus Problem

Come già menzionato questo problema è una variante del più noto CVRPTW (*Capacitated Vehicle Routing Problem with Time Windows*). Il problema si può descrivere nel seguente modo: in un pub si hanno alcuni courtesy bus, ovvero dei veicoli che effettuano un servizio taxi per i suoi clienti. Essi devono appunto riportare a casa i clienti dopo la serata trascorsa al pub a partire dall'orario da loro richiesto. Ogni cliente fornisce un lower bound dell'orario al quale vuole arrivare a casa e la soddisfazione del cliente dipenderà da quanto l'effettiva ora d'arrivo differirà da quella richiesta.

2.1 Obiettivi

Gli obiettivi sono:

- trovare l'insieme di route che minimizzino il costo di percorrenza dei vari courtesy bus
- massimizzare la soddisfazione dei clienti portandoli a casa il prima possibile considerando l'orario richiesto

2.2 Vincoli

I vincoli da rispettare sono:

- non eccedere la capacità dei bus
- portare a casa tutti i clienti
- rispettare le time windows
- far partire e ritornare ogni route dal pub

3 Formalizzazione del problema

Sia:

- K l'insieme dei bus di capacità Q
- C l'insieme dei clienti del pub
- a_i l'orario di arrivo a casa desiderato, richiesto da ogni cliente $i \in C$

- $[a_i, +\infty]$ la time windows in cui portare a casa il cliente associato al nodo i
- $G = (V, A)$ un grafo orientato con $V = \{0\} \cup C$, dove il nodo $\{0\}$ rappresenta il pub e con A insieme degli archi (i, j)
- $t_{i,j}$ il tempo di attraversamento dell'arco $(i, j) \in A$
- $c_{i,j}$ il costo di attraversamento dell'arco $(i, j) \in A$

4 Modello Courtesy Bus Problem

4.1 Variabili

Oltre alle variabili $t_{i,j}$ e $c_{i,j}$ che rappresentano rispettivamente il tempo ed il costo di attraversamento, definiamo una variabile tridimensionale $x_{i,j,k}$ per capire quali bus percorrono quali archi.

$$x_{i,j,k} = \begin{cases} 1 & \text{if bus } k \text{ travels from } i \text{ to } j \text{ directly} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Per implementare le time windows c'è bisogno di un modo per determinare quando un cliente viene riportato a casa. Introduciamo quindi due variabili temporali:

- z_i rappresenta l'istante in cui il cliente i arriva a casa
- $y_{i,k}$ che rappresenta l'istante nel quale il bus k arriva a casa del cliente i

In ultimo aggiungiamo la variabile $w_{i,k}$ che determina se il bus k porta a casa il cliente i:

$$w_{i,k} = \begin{cases} 1 & \text{if bus } k \text{ takes customer } i \text{ home} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

4.2 Funzione obiettivo

Considerando di dover minimizzare anche il tempo che impiego a riportare a casa il cliente la f.o. diventa:

$$\min \alpha \sum_{k \in K} \sum_{(i,j) \in A} c_{i,j} x_{i,j,k} + \beta \sum_{i \in C} z_i - a_i \tag{11}$$

Con α e β parametri per stabilire a quale delle componenti della f.o. dare più importanza.

4.3 Vincoli

1. Non eccedere la capacità dei bus

$$\sum_{(i,j)\in A(-,-,k)} x_{i,j,k} \leq Q;\ k \in K \tag{12}$$

2. I clienti vengono portati a casa ognuno una sola volta e da un solo bus

$$\sum_{k\in K} \sum_{i\in A(-,j,k)} x_{i,j,k} = 1;\ j \in C \tag{13}$$

3. Bilanciamento del flusso

$$\sum_{i\in A(-,h,k)} x_{i,h,k} - \sum_{j\in A(h,-,k)} x_{h,j,k} = 0;\ h \in C, k \in K \tag{14}$$

4. Vincoli viaggi bus: ogni bus parte dal pub e vi ritorna alla fine del giro. Ogni bus, se tra quelli selezionati, è utilizzato una sola volta.

   a. Il bus parte dal pub, nodo $\{0\}$.

$$\sum_{j\in A(0,-,k)} x_{0,j,k} <= 1;\ k \in K \tag{15}$$

   b. Il bus ritorna al pub, nodo $\{0\}$.

$$\sum_{i\in V} x_{i,0,k} <= 1;\ k \in K \tag{16}$$

5. Lower bound del tempo di arrivo desiderato

$$z_i \geq a_i, \text{ for } i \in C \tag{17}$$

6. Valorizzazione $y_{i,k}$: consecutività tempi di arrivo di un bus

   a. Lower bound

$$y_{j,k} \geq y_{i,k} + t_{i,j}x_{i,j,k} - M(1 - x_{i,j,k}) \tag{18}$$

   b. Upper bound

$$y_{j,k} \leq y_{i,k} + t_{i,j}x_{i,j,k} + M(1 - x_{i,j,k}) \tag{19}$$

7. Valorizzazione z_i

$$z_i = \sum_{k \in K} y'_{i,k};\ i \in I \tag{20}$$

8. Valorizzazione $w_{i,k}$

$$w_{i,k} = \sum_{j \in A(i,-,k)} x_{i,j,k};\ i \in C,\ k \in K \tag{21}$$

9. MW

$$Mw_{i,k} = M \cdot w_{i,k};\ i \in C,\ k \in K \tag{22}$$

10. Valorizzazione $y'_{i,k}$

$$y'_{i,k} = min(Mw_{i,k},\ y_{i,k}) \tag{23}$$

4.4 Note

- M, detta big-M, è una costante dal valore tendente a $+\infty$
- A è una matrice che rappresenta quali archi vengono percorsi da quale bus. È formata da 3 elementi (arco_i, arco_j, bus). La notazione $A(-, j, k)$ indica gli archi con arco i fisso e j, k non fissi.

5 Euristiche e metaeuristiche

5.1 Euristica costruttiva

Per generare una prima semplice soluzione accettabile sono stati testati tre differenti algoritmi greedy:

1. Il primo algoritmo riempie un bus alla volta aggiungendo ad ogni passo il cliente più vicino
2. Il secondo algoritmo scorre invece la lista dei bus e aggiunge un cliente alla volta scegliendo quello più vicino
3. L'ultimo algoritmo abbina in modo casuale ogni cliente ad un bus che abbia sufficiente capacità.

Tutti e tre gli algoritmi generano soluzioni simili in termini di costo, ma l'ultimo algoritmo genera una soluzione che risulta più adatta (con più possibilità di miglioramento) come soluzione iniziale da dare in input alla local search.

5.2 Local search

La local search consiste nell'esplorazione di una neighbourhood generata essenzialmente a partire una mossa:

- `MoveAndOptTime(node,bus,pos)`, una versione della *insert* che sposta il nodo `node` nella lista di clienti del bus `bus` nella posizione `pos`. `bus` può essere lo stesso di partenza o un altro.

All'interno della mossa è presente una sotto-mossa che ottimizza il tempo di partenza di `bus`. Viene calcolato per ogni nodo il tempo minimo di partenza che rispetti le time-windows e tra questi viene selezionato il massimo. Un'altra mossa presa in considerazione è stata la `2-opt`, che prende due archi all'interno di un trip e li scambia. Questa mossa però è risultata meno efficiente della `MoveNode` in quanto crea cambiamenti troppo grandi all'interno dei percorsi, soprattutto in termini di rispetto delle time windows dei clienti.

5.3 Local search multi-start

Come prima metaeuristica si è implementata la possibilità di utilizzare la local search con un multistart. Il risolutore greedy costruisce ogni volta una soluzione diversa, che viene ottimizzata tramite local search finchè non si supera il tempo limite stabilito.

5.4 Simulated annealing

Un'altra metaeuristica implementata è una versione dell'algoritmo di *simulated annealing*. Il nome deriva da un processo utilizzato in ambito metallurgico, chiamato appunto "annealing" (letteralmente ricottura), usato per determinare lo stato di minima energia di un metallo: il materiale viene scaldato e portato in uno stato nel quale gli atomi sono più liberi di muoversi, per poi essere raffreddato e tornare nuovamente allo stato solido. L'idea è quella di accettare, oltre a soluzioni all'interno della neighbourhood che migliorano la soluzione corrente, anche, con una certa probabilità, soluzioni peggiori. Questa probabilità è determinata da un valore decrescente di "temperatura" che permette di rendere più selettiva la scelta delle soluzioni man mano che si va avanti nella ricerca dell'ottimo. L'algoritmo in questo caso è strutturato nel seguente modo:

```
solution = constructive_solver.solve()
```

```
while temperature > min_temperature & (time < end_time):
  for i < n_of_iterations:
    new_solution = solution

    MoveAndOptTime(new_solution, random_src_node,
   random_dst_bus, random_dst_pos).apply()

    if new_solution is feasible do
    delta = old_cost - new_cost
      if random() <= exp(delta/temperature)
solution = new_solution

      if new_cost < best_cost:
best_solution = solution
    i++
  temperature = temperature * cooling_rate

return best_solution
```

I parametri da specificare sono:

- **cooling rate**, ovvero il tasso di raffreddamento, che serve a regolare la velocità con la quale scende la temperatura. Ha un valore massimo di 1 (la temperatura rimane costante) e minimo di 0 (la temperatura cala subito al minimo)
- **initial** e **minimum temperature**, ovvero la temperatura di partenza e quella minima alla quale arrestare l'ottimizzazione
- **iterations per temperature**, il numero di iterazioni nelle quali è permesso applicare una mossa nella neighbourhood

I valori dei paramentri utilizzati di default e che in generale hanno dato risultati migliori sono i seguenti:

- COOLING_RATE = 0.98
- INITIAL_TEMPERATURE = 10
- MINIMUM_TEMPERATURE = 1
- ITERATIONS_PER_TEMPERATURE = 10000

Ma rimane comunque la possibilità di parametrizzare la simulated annealing a piacimento in base al dataset del problema.

6 Struttura del progetto

A livello di struttura il progetto si compone di alcuni moduli.

6.1 Model

In `model.py` è definita la classe `Model`, che rappresenta l'istanza del problema. Gli attributi della classe sono:

- `N` il numero di bus
- `Q` la capacità del bus
- `customers` la lista dei clienti rappresentati da un numero >= 1
- `alpha` il peso del costo delle route nella f.o. (di default = 1)
- `beta` il peso della *customer satisfaction* nella f.o. (di default = 1)
- `omega` il parametro distanza/tempo di percorrenza di una route (di default = 1)

I metodi di `Model` sono:

- `parse(filname)` che legge un'istanza del problema da file
- `distance(v1, v2)` calcola la distanza euclidea tra due nodi/clienti
- `time(v1, v2)` calcola la distanza in termini di tempo tra due nodi/clienti
- `buses()` genera la lista di `N` bus
- `des_arrived_times()` ritorna il tempo di arrivo richiesto dai clienti
- `customer_arr_time(v)` ritorna il tempo di arrivo richiesto dal cliente `v`

6.2 Solution

In `solution.py` è definita la classe `Solution` che rappresenta una soluzione dell'istanza del problema. Gli attributi della classe sono:

- `model`, il modello dell'istanza del problema
- `trips` una lista di liste contenente i cammini facenti parte della soluzione

Come metodi la classe `Solution` include:

- `save(filename)` salva la soluzione su file
- `descrition` esprime la soluzione in formato leggibile

6.3 Validator

Il modulo `validator.py` contiene la definizione della classe `Validator` la quale valida una soluzione in base modello corrispondente e calcola i costi sia in termini di distanza percorsa, sia in termini di soddisfazione dei clienti. Ha due attributi `model` e `solution` che rispettivamente l'istanza del problema e la soluzione che si sta validando. I metodi `check_H1, check_H2, check_H4, check_H5, check_H6` effettuano un controllo sui vincoli come definiti in 2.2. Il metodo `customer_satisfaction()` calcola la soddisfazione del cliente (minore è il valore, maggiore è la soddisfazione) mentre `trips_cost()` calcola il costo dei percorsi contenuti nella soluzione. La funzione `validate()` restituisce un oggetto `ValidationResult` contenente il risultato della validazione, ovvero se è *feasible* o se infrange qualche vincolo.

6.4 GurobiSolver

In questo modulo viene definita la classe `GurobiSolver` nella quale viene descritto il modello tramite Gurobi e viene implementato il metodo `solve()` che genera - quando possibile - la soluzione ottima. All'interno del metodo `solve()` avviene la gestione dei subtour (problema descritto in 1.1.1): anzichè aggiungere un numero esponenziale di vincoli per la loro eliminazione, viene implementato - in maniera *lazy* - una serie di *callback* chiamate solamente nel caso in cui siano presenti dei subtour nella soluzione. In questo modo si riescono ad imporre dei vincoli in modo dinamico, senza andare ad impattare il resto del modello Gurobi.

6.5 Heuristics

In questo modulo sono raccolte le euristiche e metaeuristiche, ovvero:

- `GreedySolver` il risolutore costruttivo greedy che assegna un cliente random ad un bus random
- `MoveOptTime` la classe che definisce la mossa di inserimento utilizzata per generare le diverse neighbourhood
- `LocalSearch` la classe che implementa l'euristica di local search e utilizza la mossa del punto precedente
- `SimulatedAnnealing` che implementa la tecnica di simulated annealing sempre utilizzando la mossa `MoveOptTime` per generare le sue neighbourhood

7 Istanze del problema e risultati

In tabella 1 vengono riportati raccoglie i risultati ottenuti tramite i risolutori implementati all'interno del progetto. Le istanze del problema (presenti nella cartella `datasets`) sono state generate casualmente, aumentando man mano il numero di clienti e bus (fino ad arrivare a problemi non più risolvibili all'ottimo) e variando anche la sparsità dei nodi. Per ogni istanza viene descritto il numero di nodi (*# nodes*), ovvero clienti, da servire, il numero di bus (*# buses*), i secondi lasciati all'esecuzione del risolutore (*seconds to solve*) ed il valore della soluzione (solution value). Come si può notare dalla tabella (dati evidenziati con un asterisco), si riescono ad ottenere soluzioni ottime con Gurobi fino ad numero di clienti contenuto (5° blocco/riga) e altrettanto buone con local search multistart e simulated annealing. Passando al 6° blocco/riga notiamo invece che gurobi non riesce più ad arrivare all'ottimo e anche local search multi-start non si avvicina quanto simulated annealing. Nell'instanza del problma con 201 clienti si rende palese come simulated annealing performi in maniera decisamente migliore a parità di tempo di local search con multistart.

8 Bibliografia

- Cordeau, J.F., Desaulniers, G., Desrosiers J., Solomon, M.M., Soumis, F., 1999, The VRP with Time Windows, Les Cahiers du GERAD, G–99–13

Table 1: Tabella dei risultati -soluzioni indicate dall'asterisco rappresentano soluzioni ottime

solver	**# nodes**	**# buses**	**seconds to solve**	**solution value**
gurobi	3	2	10	47.2431*
ls	3	2	10	47.2431
ls-ms	3	2	10	47.2431
sa	3	2	10	47.2431
gurobi	4	2	10	56.2986*
ls	4	2	10	63.2672
ls-ms	4	2	10	56.2986
sa	4	2	10	56.2986
gurobi	7	3	10	116.6939*
ls	7	3	10	118.4272
ls-ms	7	3	10	116.6939
sa	7	3	10	116.6939
gurobi	10	3	10	202.4896*
ls	10	3	10	261.5168
ls-ms	10	3	10	202.4896
sa	10	3	10	202.4896
gurobi	11	10	100	213.2659*
ls	11	10	100	223.1647
ls-ms	11	10	100	213.2659
sa	11	10	100	213.2659
gurobi	20	5	100	3238.1498
ls	20	5	100	3379.7927
ls-ms	20	5	100	2998.8899
sa	20	5	100	2998.8899
gurobi	42	8	200	772.44067
ls	42	8	200	715.7607
ls-ms	42	8	200	660.1733
sa	42	8	200	588.4076
gurobi	52	11	200	4610.3419
ls	52	11	200	3131.1859
ls-ms	52	11	200	2984.3199
sa	52	11	200	2889.8812
gurobi	201	13	200	-
ls	201	13	200	59069.6611
ls-ms	201	13	200	57914.4437
sa	201	13	200	4914.3869

- El-Sherbeny, Nasser A., 2010, Vehicle routing with time windows: An overview of exact, heuristic and metaheuristic methods, Journal of King Saud University (Science)
- N.R. Achuthan, L. Caccetta, Integer linear programming formulation for a vehicle routing problem, European Journal of Operational Research, 52 (1991), pp. 86-89
- G. Laporte, The vehicle routing problem: An overview of exact and approximate algorithms, European Journal of Operational Research, 59 (1992), pp. 231-247
- Capacitated Vehicle Routing Problem formulation
- Implicit Dantzig-Fulkerson-Johnson formulation
- Explicit Dantzig-Fulkerson-Johnson formulation
- Esempio di callback gurobi