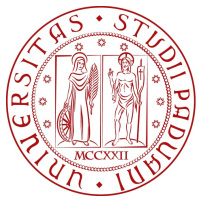

Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Laurea triennale in Ingegneria Informatica

Un applicativo per il problema del percorso minimo applicato ai veicoli elettrici

Relatore:

Prof. Domenico Salvagnin

Laureando:

Daniele Moschetta

Anno Accademico 2021-2022

Data di laurea 20/07/2022

Abstract

In questo elaborato si sono voluti sfruttare i principi della programmazione lineare intera mista (MILP) per sviluppare un applicativo in linguaggio Python in grado di automatizzare, a partire dalle caratteristiche di un veicolo elettrico, la ricerca del percorso ottimo tra due luoghi che includa la scelta delle stazioni di ricarica e delle relative tempistiche di sosta. L'applicativo sviluppato ha l'obiettivo di incentivare l'utilizzo dei mezzi elettrici, sollevando l'utente dalla complicata organizzazione dei lunghi tragitti e ottimizzando il consumo delle limitate risorse di autonomia dei veicoli.

Indice

Capitolo 1

Introduzione

La diffusione dei veicoli elettrici nel mercato automobilistico sta subendo una costante crescita, dovuta soprattutto al loro ridotto impatto ambientale e all'economicità dell'energia elettrica rispetto al carburante tradizionale [6]. Nonostante questo, l'utilizzo di tali mezzi impone diverse limitazioni legate in primo luogo alla percorrenza di lunghe tratte, come le ridotte caratteristiche di autonomia, la disuniforme distribuzione delle colonnine di ricarica sul territorio e le tempistiche di ricarica [6].

L'obiettivo di questo elaborato è quello di sviluppare un applicativo che sollevi l'utente dalla complicata organizzazione dei tragitti, così da incentivare l'utilizzo dei veicoli elettrici per la percorrenza di lunghe tratte. L'applicativo sviluppato dovrà essere in grado, a partire dalle caratteristiche di un veicolo elettrico, di automatizzare la ricerca del percorso ottimo tra due luoghi, che includa la scelta delle stazioni di ricarica e delle relative tempistiche di sosta.

Il primo capitolo consiste in un'introduzione alle tematiche di base dell'ottimizzazione, ai principali algoritmi di risoluzione dei problemi e a Pyomo, il software di ottimizzazione utilizzato per l'implementazione. Nel secondo capitolo viene illustrata la formulazione del modello matematico che permette di rappresentare la problematica in analisi. Nel terzo capitolo viene illustrata l'implementazione, in linguaggio Python, dell'applicativo. Infine, il quarto capitolo consiste in un'analisi dei risultati ottenuti dall'utilizzo dell'applicativo sviluppato su una serie di istanze.

1.1 Programmazione lineare intera mista

La *ricerca operativa* è una disciplina che consiste nell'analisi e nella risoluzione di problematiche decisionali reali attraverso l'utilizzo di modelli matematici. L'*ottimizzazione* (o programmazione matematica) è una branca della ricerca operativa che si occupa della minimizzazione/massimizzazione di una funzione obiettivo sottoposta ad una serie di vincoli. Un problema di ottimizzazione che ammette solo vincoli lineari si dice di *programmazione lineare* (LP) ed in generale è esprimibile secondo la *forma canonica* [3]:

$$\begin{cases} \min \mathbf{c}^T \mathbf{x} \\ A\mathbf{x} \geq \mathbf{b} \\ \mathbf{x} \geq 0 \end{cases} \tag{1.1}$$

Dal punto di vista geometrico, ogni vincolo applicato ad un problema costituisce un piano nello spazio e l'intersezione di tutti i piani forma un poliedro, che rappresenta lo spazio delle soluzioni ammissibili del problema. Per il *Teorema Fondamentale della Programmazione Lineare*, se un problema ammette una soluzione ottima finita, allora almeno un vertice del poliedro rappresenta la soluzione ottima [3]. Da questo principio deriva il principale metodo di risoluzione di problemi LP, ovvero l'*algoritmo del simplesso*. L'algoritmo consiste nel visitare, a partire da un vertice qualsiasi del poliedro, i vertici adiacenti non peggiori, fino a terminare quando si trova quello ottimo.

Un problema di *programmazione lineare intera mista* (MILP) è un problema di ottimizzazione vincolato dal fatto che alcune variabili debbano assumere valori interi. Nel caso in cui il vincolo di interezza riguardi tutte le variabili, il problema si dice di *programmazione lineare intera pura*. In generale, nei problemi MILP, i vertici del poliedro che rappresenta lo spazio delle soluzioni ammissibili possono assumere valori non interi, quindi gli algoritmi basati sui vertici non sono direttamente applicabili.

1.2 Algoritmo Branch & Bound

Si definisce *rilassamento* di un problema di ottimizzazione P, un problema R ottenuto dal problema iniziale eliminando alcuni vincoli e/o sostituendo la funzione obiettivo $f(x)$ con un'approssimazione inferiore $g(x)$. É utile studiare il rilassamento di un problema perché permette ottenerne una versione più semplice dalla cui risoluzione è possibile derivare delle informazioni sul problema principale, in particolare:

- se R è impossibile, allora P è impossibile;
- se x^* è soluzione ottima di R, $x^* \in F(P)$ e $g(x^*) = g(x^*)$, allora (x^*) è soluzione ottima di P;

- se x^* è soluzione ottima di R, allora $g(x^*)$ è un limite inferiore per la soluzione ottima di P.

L'algoritmo Branch & Bound è una tecnica di ottimizzazione generica basata sulla strategia *divide-and-conquer*, che consiste nel partizionare il problema principale in sottoproblemi più semplici da risolvere separatamente.

Definiamo come F l'insieme delle soluzioni ammissibili di un problema di minimizzazione, $c : F \to R$ la funzione obiettivo del problema e $\overline{x} \in F$ una soluzione ammissibile nota. Si definisce *incumbent* il limite superiore (*upper bound*) sul valore della soluzione ottima, in questo caso $z = c(\overline{x})$. L'algoritmo B&B consiste in:

- fase di *bounding*: si risolve un rilassamento del problema, che ammette un insieme di soluzioni ammissibili più ampio $G \supseteq F$. La soluzione del rilassamento fornisce una stima pessimistica (*lower bound*) sul valore della soluzione ottima. Se la soluzione trovata appartiene a F o corrisponde all'incumbent attuale, allora è la soluzione ottima del problema principale, altrimenti si effettua il branching;

- fase di *branching*: si identifica una *separazione* F^* di F, definita come un insieme finito di sottoinsiemi di F, detti figli, tale che $\bigcup_{F_i \in F^*} F_i = F$. A questo punto, dato che:

$$\min \{c(x) \mid x \in F\} = \min \{\min \{c(x) \mid x \in F_i\} \mid F_i \in F^*\} \tag{1.2}$$

  ogni figlio appartenente alla separazione viene aggiunto ad un albero di sottoproblemi attivi da risolvere. Uno alla volta si seleziona un sottoproblema tra quelli attivi e se ne risolve un rilassamento, trovandosi in quattro possibili casistiche:

  1. il sottoproblema ha soluzione ammissibile migliore dell'incumbent, allora si aggiorna $\overline{x}$ e si continua;
  2. il sottoproblema non ha soluzione ammissibile, allora si scarta il nodo. Questa operazione è chiamata *pruning by infeasibility*;
  3. il sottoproblema ha soluzione ammissibile peggiore dell'incumbent, allora si scarta il nodo. Questa operazione è chiamata *pruning by optimality*;
  4. altrimenti, si effettua branching sul sottoproblema.

L'algoritmo B&B termina quando nell'albero non sono più presenti nodi attivi da risolvere oppure quando l'*optimality gap*, definito come la differenza tra l'upper e il lower bound sul valore della soluzione ottima, è nullo.

Nello specializzare il Branch & Bound generico per la risoluzione di problemi MILP è necessario specificare come debbano essere calcolati i rilassamenti. La tecnica comunemente utilizzata consiste nel rilassare il vincolo di interezza sulle variabili. Nel caso in cui la soluzione x_i^* trovata al rilassamento non sia intera è possibile procedere considerando la partizione $x_i \leq \lfloor x_i^* \rfloor \vee x_i \geq \lceil x_i^* \rceil$.

1.3 Algoritmo Branch & Cut

L'algoritmo Branch & Cut è una versione migliorata del B&B, sviluppata per la risoluzione di problemi MILP. In particolare, ad ogni nodo dell'albero dei sottoproblemi, la formulazione del rilassamento può essere rafforzata dall'applicazione di *piani di taglio*, dei vincoli che riducono la dimensione del dominio delle soluzioni senza però scartare soluzioni ammissibili. Lo scopo di tale rafforzamento è ottenere:

- una soluzione intera del rilassamento lineare del problema;
- un lower bound migliore sul valore della soluzione ottima del problema, così da rendere più efficace il *pruning*.

1.4 Pyomo

Pyomo [1, 4] (Python Optimization Modeling Objects) è un progetto *open-source* basato sul linguaggio Python che permette di formulare, risolvere ed analizzare una vasta gamma di tipologie di modelli matematici per la risoluzione di problemi di ottimizzazione.

Il processo di ottimizzazione di un problema con Pyomo è suddiviso in più fasi:

1. formulazione del modello: avviene utilizzando i costrutti tipici della programmazione Python orientata agli oggetti (OOP), in particolare è possibile rappresentare insiemi, parametri, variabili e vincoli di un modello matematico tramite degli appositi oggetti;
2. istanziazione del modello: un modello può essere di tipo concreto, ovvero basato su dati disponibili nel momento della definizione del modello, oppure astratto, cioè basato su dati che vengono resi disponibili al momento dell'istanziazione;
3. applicazione di un risolutore: è possibile scegliere tra un elenco di risolutori quale verrà applicato al modello sulla base della tipologia del problema da risolvere;
4. interrogazione dei risultati: una volta terminata la risoluzione, nel caso il risolutore abbia individuato una soluzione ottima, è possibile visualizzarne l'esito ed interrogarne i risultati.

Capitolo 2

Formulazione MILP del problema

Dato un grafo orientato $G = (V, E)$, in cui la percorrenza di ogni arco $(i, j) \in E$ comporta un costo, l'Elementary Shortest Path Problem (ESPP) è un problema di ottimizzazione che consiste nella ricerca del percorso con costo minimo che consente di raggiungere un nodo di arrivo $b \in V$ da un nodo di partenza $a \in V$. Il percorso cercato è definito *elementare* perché si impone che ogni nodo $v \in V$ possa essere visitato al massimo una volta.

L'obiettivo di questa formulazione è quello di descrivere una variante del problema ESPP vincolata dall'utilizzo di un veicolo elettrico. In particolare, a differenza della formulazione standard, è necessario considerare le limitate caratteristiche di autonomia del veicolo, il consumo della batteria e la conseguente necessità di effettuare delle soste di ricarica quando quest'ultima è in esaurimento.

L'insieme V dei nodi del grafo è composto da un nodo di partenza a, un nodo di arrivo b e dei nodi di ricarica v_i con $i = 1, \ldots, M$ dove è possibile effettuare delle soste di ricarica. La dimensione del grafo risulta quindi essere $N = |V| = M + 2$.

In generale, il grafo in analisi può essere *non completo*, ovvero data una coppia di nodi $i, j \in V$ non è necessario che esista sempre un arco $(i, j) \in E$ che li connette.

2.1 Insiemi

Gli insiemi di un problema definiscono quali sono gli oggetti fondamentali che verranno descritti nella formulazione. Gli insiemi individuati per il problema sono descritti nella Tabella 2.1.

Insieme	**Descrizione**
$V = \{a, v_1, \ldots, v_M, b\}$	nodi del grafo
$E = \{(a, v_1), \ldots, (v_M, b)\}$	archi del grafo

Tabella 2.1: Descrizione degli insiemi

Si definiscono inoltre come $\delta^+(v) \subset E$ e $\delta^-(v) \subset E$ rispettivamente l'insieme degli archi uscenti e degli archi entranti di un nodo $v \in V$.

2.2 Parametri

I parametri di un problema sono dei valori fissati che definiscono le caratteristiche della realtà di riferimento. I parametri possono essere dei singoli valori oppure dei gruppi di valori che specificano una caratteristica di ogni elemento di un insieme del problema. I parametri del problema sono descritti nella Tabella 2.2.

Parametro	**Descrizione**
C_{max}	capacità massima della batteria del veicolo
C_{min}	capacità minima consentita della batteria del veicolo
C_{start}	capacità della batteria del veicolo alla partenza
E	efficienza del veicolo, definita come rapporto di conversione tra capacità della batteria e distanza percorribile
R_{max}	tempo massimo di ricarica desiderato
R_{min}	tempo minimo di ricarica desiderato
t_{ij}	tempo impiegato per percorrere l'arco (i, j)
d_{ij}	distanza impiegata per percorrere l'arco (i, j)
p_i	potenza della stazione di ricarica presente al nodo i, in particolare per i nodi di partenza e di destinazione si ha $p_a = p_b = 0$

Tabella 2.2: Descrizione dei parametri

2.3 Variabili

Le variabili decisionali sono le grandezze che rappresentano la soluzione del problema. L'obiettivo finale del processo di ottimizzazione è infatti quello di assegnare alle variabili dei valori che rispettino i vincoli imposti e al tempo stesso ottimizzino la funzione obiettivo. Le variabili scelte per il problema sono descritte nella Tabella 2.3.

Variabile	**Descrizione**
$x_{ij} \in \{1, 0\}$	indica se si percorre o meno l'arco (i, j)
$y_i \in \{1, 0\}$	indica se si visita o meno il nodo i
$r_i \in \mathbb{Z}_+$	indica il tempo di ricarica al nodo i
$C_{in,i} \in \mathbb{Z}_+$	indica la capacità residua della batteria in entrata al nodo i
$C_{out,i} \in \mathbb{Z}_+$	indica la capacità residua della batteria in uscita dal nodo i

Tabella 2.3: Descrizione delle variabili

2.4 Funzione obiettivo

La funzione obiettivo di un problema rappresenta la grandezza che si vuole ottimizzare. In particolare, in questa formulazione, la soluzione ottima è quella che minimizza il tempo totale impiegato per raggiungere il nodo di destinazione b, dato dalla somma dei tempi di percorrenza t_{ij} degli archi percorsi e dei tempi di ricarica r_i nei nodi di ricarica visitati

$$\min \sum_{(i,j)\in E} x_{ij} t_{ij} + \sum_{i \in V} r_i \tag{2.1}$$

2.5 Vincoli

I vincoli di un problema rappresentano le condizioni che devono essere soddisfatte dalle variabili decisionali perché una soluzione possa essere considerata ammissibile. Nelle formulazioni MILP i vincoli devono obbligatoriamente essere lineari, motivo per cui alcune condizioni complesse, come il Vincolo di consumo della batteria descritto nella Sezione 2.5.8, necessitano di essere *linearizzate* [3].

2.5.1 Vincoli di flusso

I vincoli di flusso impongono che il percorso debba essere continuo e che i nodi a e b vengano considerati rispettivamente nodo di partenza e di arrivo. In particolare si ha:

- Il nodo di partenza a deve avere un nodo uscente in più di quelli entranti

$$\sum_{(a,i)\in\delta^+(a)} x_{ai} - \sum_{(i,a)\in\delta^-(a)} x_{ia} = +1 \tag{2.2}$$

- Il nodo di arrivo b deve avere un nodo entrante in più di quelli uscenti

$$\sum_{(b,i)\in\delta^+(b)} x_{bi} - \sum_{(i,b)\in\delta^-(b)} x_{ib} = -1 \tag{2.3}$$

- Tutti gli altri nodi devono avere un numero uguale di nodi entranti ed uscenti

$$\sum_{(v,i)\in\delta^+(v)} x_{vi} - \sum_{(i,v)\in\delta^-(v)} x_{iv} = 0 \qquad \forall v \in V \setminus \{a,b\} \tag{2.4}$$

2.5.2 Vincolo di visita nodo

Per effetto dei vincoli (2.2) e (2.3) il nodo di partenza a e il nodo di arrivo b vengono considerati come sempre visitati. Per quanto riguarda gli altri nodi, il vincolo di visita nodo impone che un nodo di ricarica v venga considerato visitato se viene raggiunto da un altro nodo tramite uno degli archi percorsi

$$y_v = \sum_{(i,v)\in\delta^-(v)} x_{iv} \qquad \forall v \in V \setminus \{a,b\} \tag{2.5}$$

É importante notare che, dato che la variabile y_i è binaria, il vincolo (2.5) impone implicitamente che un nodo possa essere visitato al massimo una volta.

2.5.3 Vincolo di tempo di ricarica

Il vincolo di tempo di ricarica impone che, dato un nodo di ricarica v, il tempo di ricarica r_v debba essere nullo nel caso in cui il nodo non venga visitato e compreso tra quello minimo e massimo desiderato nel caso in cui venga visitato

$$R_{min} y_v \leq r_v \leq R_{max} y_v \qquad \forall v \in V \setminus \{a,b\} \tag{2.6}$$

Il vincolo (2.6) implica inoltre che la visita ad un nodo di ricarica debba sempre comportare una sosta di ricarica.

2.5.4 Vincolo di batteria alla partenza

Il vincolo di batteria alla partenza impone che la capacità residua della batteria in uscita dal nodo di partenza a debba essere pari a quella iniziale del veicolo

$$C_{out,a} = C_{start} \tag{2.7}$$

2.5.5 Vincolo di ricarica della batteria

Il vincolo di ricarica della batteria impone che la capacità residua della batteria in uscita da un determinato nodo di ricarica v debba essere pari alla somma tra quella in entrata e quella ricaricata durante l'eventuale sosta

$$C_{out,v} = C_{in,v} + r_v \cdot p_v \qquad \forall v \in V \setminus \{a, b\} \tag{2.8}$$

2.5.6 Vincolo di batteria in entrata

Il vincolo di batteria in entrata impone che, nel caso in cui si transiti in un determinato nodo v, la capacità residua della batteria in entrata al nodo non debba eccedere i limiti di batteria consentiti

$$C_{min} y_v \leq C_{in,v} \leq C_{max} y_v \qquad \forall v \in V \setminus \{a\} \tag{2.9}$$

Non è necessario imporre alcun vincolo su $C_{in,a}$, dato che il nodo di partenza non è un nodo di ricarica.

2.5.7 Vincolo di batteria in uscita

Il vincolo di batteria in uscita impone che, nel caso in cui si transiti in un determinato nodo v, la capacità residua della batteria in uscita dal nodo non debba eccedere i limiti di batteria consentiti

$$C_{min} y_v \leq C_{out,v} \leq C_{max} y_v \qquad \forall v \in V \setminus \{b\} \tag{2.10}$$

Non è necessario imporre alcun vincolo su $C_{out,b}$, dato che il nodo di destinazione non è un nodo di ricarica.

2.5.8 Vincolo di consumo della batteria

Il vincolo di consumo della batteria impone che, nel caso in cui venga percorso l'arco (i, j) tra due nodi, la capacità residua della batteria in entrata al secondo nodo debba essere pari alla differenza tra quella in uscita dal primo nodo e quella consumata percorrendo l'arco

$$x_{ij} = 1 \rightarrow C_{in,j} = C_{out,i} - E \cdot d_{ij}$$

È possibile linearizzare questo vincolo utilizzando il metodo big-M [3]:

$$C_{in,j} \leq C_{out,i} - E \cdot d_{ij} + M_1(1 - x_{ij}) \qquad \forall i, j \in V \tag{2.11}$$

$$C_{in,j} \geq C_{out,i} - E \cdot d_{ij} - M_2(1 - x_{ij}) \qquad \forall i, j \in V \tag{2.12}$$

Scegliendo per i valori big-M:

$$M_1 = C_{max} + E \cdot d_{ij}$$

$$M_2 = C_{max} - E \cdot d_{ij}$$

si ottiene

$$\begin{cases} C_{in,j} = C_{out,i} - E \cdot d_{ij} & \text{se } x_{ij} = 1 \\ C_{out,i} - C_{max} \leq C_{in,j} \leq C_{out,i} + C_{max} & \text{se } x_{ij} = 0 \end{cases}$$

In questo modo, nel caso in cui l'arco tra i due nodi non venga percorso, si ottiene un vincolo su $C_{in,j}$ meno stringente del vincolo (2.9).

2.6 Modello completo

$$
\begin{cases}
\min \sum_{(i,j)\in E} x_{ij} t_{ij} + \sum_{i \in V} r_i & & (2.1)\\
\sum_{(a,i)\in\delta^+(a)} x_{ai} - \sum_{(i,a)\in\delta^-(a)} x_{ia} = +1 & & (2.2)\\
\sum_{(b,i)\in\delta^+(b)} x_{bi} - \sum_{(i,b)\in\delta^-(b)} x_{ib} = -1 & & (2.3)\\
\sum_{(v,i)\in\delta^+(v)} x_{vi} - \sum_{(i,v)\in\delta^-(v)} x_{iv} = 0 & \forall v \in V \setminus \{a,b\} & (2.4)\\
y_v = \sum_{(i,v)\in\delta^-(v)} x_{iv} & \forall v \in V \setminus \{a,b\} & (2.5)\\
R_{min} y_v \leq r_v \leq R_{max} y_v & \forall v \in V \setminus \{a,b\} & (2.6)\\
C_{out,a} = C_{start} & & (2.7)\\
C_{out,v} = C_{in,v} + r_v \cdot p_v & \forall v \in V \setminus \{a,b\} & (2.8)\\
C_{min} y_v \leq C_{in,v} \leq C_{max} y_v & \forall v \in V \setminus \{a\} & (2.9)\\
C_{min} y_v \leq C_{out,v} \leq C_{max} y_v & \forall v \in V \setminus \{b\} & (2.10)\\
C_{in,j} \leq C_{out,i} - E \cdot d_{ij} + M_1(1 - x_{ij}) & \forall i,j \in V & (2.11)\\
C_{in,j} \geq C_{out,i} - E \cdot d_{ij} - M_2(1 - x_{ij}) & \forall i,j \in V & (2.12)\\
x_{ij} \in \{1,0\}, y_i \in \{1,0\}, r_i \in \mathbb{Z}_+, C_{in,i} \in \mathbb{Z}_+, C_{out,i} \in \mathbb{Z}_+ & &
\end{cases}
$$

Capitolo 3

Sviluppo dell'applicativo

Lo scopo dell'applicativo è automatizzare il calcolo del percorso minimo che permette ad un veicolo elettrico di raggiungere una determinata destinazione. Definiamo come *percorso base* il tragitto ottimo che consente di raggiungere il luogo di arrivo dal luogo di partenza, indipendentemente dall'autonomia del veicolo. In base alle caratteristiche del mezzo, il percorso base deve essere modificato per garantire che il veicolo sia in grado di raggiungere la destinazione senza esaurire la batteria. L'applicativo elabora i dati necessari all'istanziazione del modello descritto nel Capitolo 2, per poi delegare la decisione del percorso minimo all'ottimizzatore. Il flusso di esecuzione è rappresentabile secondo lo schema della Figura 3.1.

L'applicativo è stato implementato in linguaggio Python v3.7.0[1], utilizzando:

- Bing Maps APIs[2]: servizi di Microsoft che permettono di reperire, tramite delle richieste HTTP GET, informazioni riguardanti le mappe, come ad esempio luoghi, percorsi e traffico in tempo reale. In particolare, in questo applicativo verranno utilizzati Route API e Distance Matrix API;

- Open Charge Map APIs[3]: servizi che permettono di reperire, tramite delle richieste HTTP GET, informazioni riguardanti le stazioni di ricarica per veicoli elettrici presenti in un territorio. In particolare, in questo applicativo verrà utilizzato POI (Point Of Interest) API;

- Polyline[4]: libreria Python che implementa l'algoritmo Encoded Polyline Algorithm Format di Google, che permette di codificare una lista di coordinate geografiche in una stringa di testo;

- Requests[5]: libreria Python che permette di effettuare richieste HTTP.

[1] `https://docs.python.org/release/3.7.0/`
[2] `https://docs.microsoft.com/en-us/bingmaps/rest-services/`
[3] `https://openchargemap.org/site/develop#api`
[4] `https://pypi.org/project/polyline/`
[5] `https://pypi.org/project/requests/`

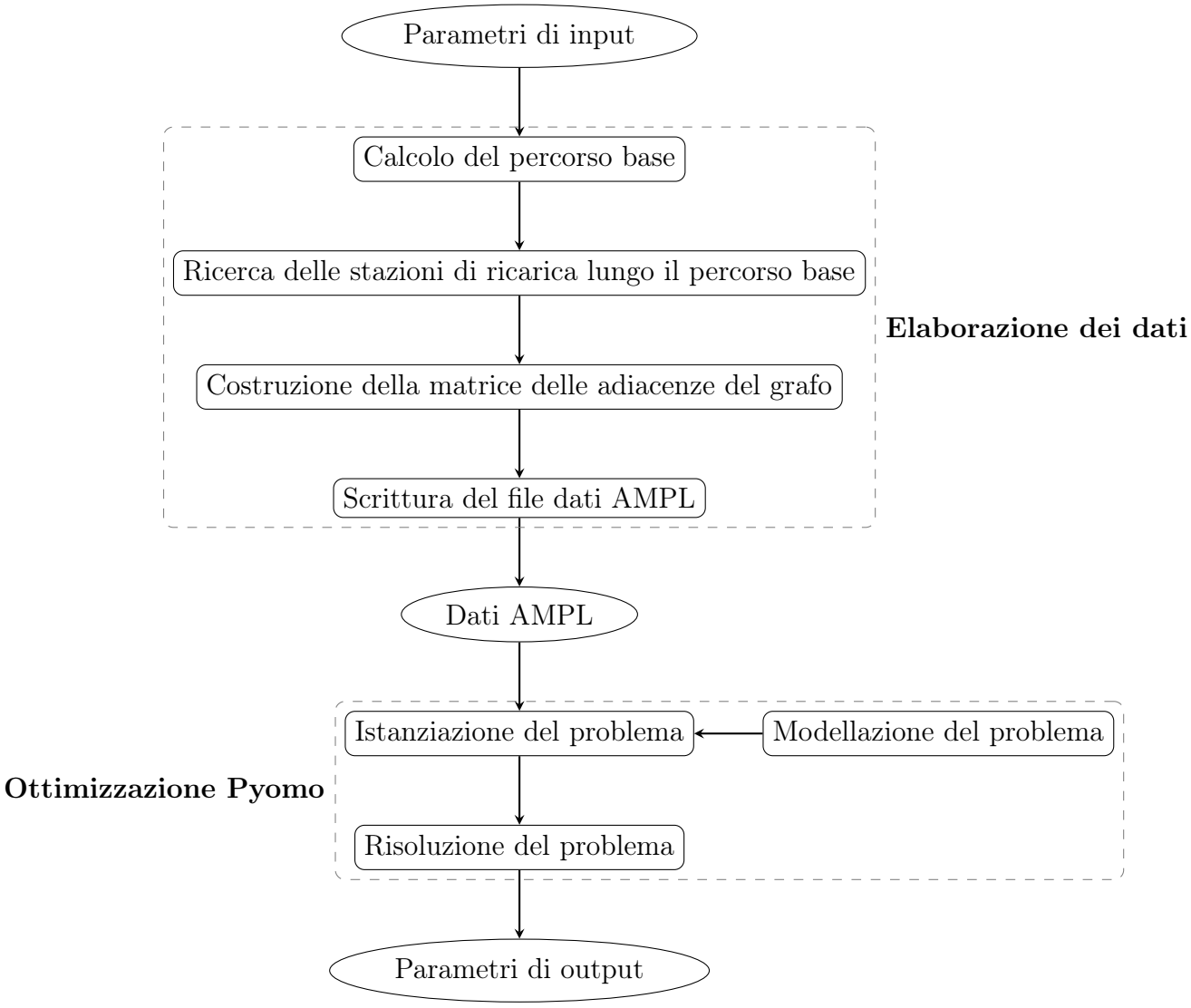

Figura 3.1: Flusso di esecuzione dell'applicativo

3.1 Parametri di input

I parametri di input necessari all'esecuzione dell'applicativo sono descritti nella Tabella 3.1.

Parametro	Descrizione
`origin (string)`	luogo di partenza
`destination (string)`	luogo di arrivo
`max_capacity (float)`	capacità massima della batteria del veicolo $[kWh]$
`start_capacity (float)`	capacità della batteria del veicolo alla partenza $[kWh]$
`autonomy (float)`	autonomia stimata del veicolo $[km]$
`std_type (string)`	tipologia del connettore di ricarica standard del veicolo, secondo lo standard IEC 62196 [2]
`std_power (float)`	potenza del connettore di ricarica standard del veicolo $[kW]$
`fast_type (string)`	tipologia del connettore di ricarica rapida del veicolo, secondo lo standard IEC 62196 [2]
`fast_power (float)`	potenza del connettore di ricarica rapida del veicolo $[kW]$
`min_charge_time (float)`	tempo minimo di sosta di ricarica desiderato $[h]$
`max_charge_time (float)`	tempo massimo di sosta di ricarica desiderato $[h]$
`max_drive_time (float)`	tempo massimo di guida consecutivo desiderato $[h]$

Tabella 3.1: Descrizione dei parametri di input

Sulla base dei parametri della Tabella 3.1, vengono calcolati i parametri:

```
max_charger_distance = autonomy / 10 # 10% of the total autonomy
min_capacity = max_capacity / 20 # 5% of the total battery capacity
car_efficiency = max_capacity / autonomy
```

Codice 3.1: Parametri calcolati

3.2 Elaborazione dei dati

3.2.1 Calcolo del percorso base

Il percorso base viene calcolato utilizzando il servizio Route API[6], che permette di reperire informazioni riguardanti il percorso tra più luoghi chiamati *waypoints*, in questo caso luogo di origine e di destinazione, tramite la richiesta:

```
def get_route(self, origin, destination):
    url = 'http://dev.virtualearth.net/REST/V1/Routes/Driving'
    params = {
        'key': self.key, # API key
        'routeAttributes': 'routePath', # Request for route points
        'wp.0': origin, # First route waypoint
        'wp.1': destination # Last route waypoint
    }
    response = requests.get(url, params = params)
    if not response.ok:
        raise requests.RequestException(response.text)
    data = response.json()
    return data['resourceSets'][0]['resources'][0]
```

Codice 3.2: Richiesta Route API

Il parametro `routeAttributes`, utilizzato nel Codice 3.2, consente di richiedere che venga restituita la lista delle coordinate dei punti che compongono il tragitto. Tramite la libreria `polyline`, l'insieme delle coordinate che costituisce il percorso base viene codificato in una stringa di testo chiamata *polilinea*:

```
route = bingmaps_client.get_route(origin, destination)
all_points = route['routePath']['line']['coordinates']
route_polyline = polyline.encode(all_points)
```

Codice 3.3: Codifica della polilinea

[6] https://docs.microsoft.com/en-us/bingmaps/rest-services/routes/calculate-a-route

3.2.2 Ricerca delle stazioni di ricarica lungo il percorso base

Ogni nodo del grafo viene rappresentato come un oggetto `Node`, nel caso in cui si tratti di una stazione di ricarica è necessario specificarne tipologia e potenza.

```
class Node:
    def __init__(self, name, coordinates, charger_type = None, charger_power = 0):
        self.name = name
        self.coordinates = coordinates
        self.charger_type = charger_type
        self.charger_power = charger_power
```

Codice 3.4: Oggetto Node

Il servizio POI API[7] consente di reperire la lista delle stazioni di ricarica localizzate entro una certa distanza dai punti di una polilinea, tramite la richiesta:

```
def get_chargers(self, polyline, distance, max_results):
    url = 'https://api.openchargemap.io/v3/poi/output=json'
    params = {
        'key': self.key, # API key
        'polyline': polyline, # Route poyline
        'distance' : distance, # Distance from polyline points
        'distanceunit' : 'km',
        'maxresults': max_results
    }
    response = requests.get(url, params = params)
    if not response.ok:
        raise requests.RequestException(response.text)
    data = response.json()
    return data
```

Codice 3.5: Richiesta POI API

Il parametro `maxresults`, utilizzato nel Codice 3.5, è un parametro obbligatorio che indica quale debba essere il numero massimo di stazioni di ricarica restituibili dall'API. É importante notare che questo parametro definisce la dimensione massima del problema che dovrà essere ottimizzato. Gli effetti della variazione del numero massimo dei risultati sulla risoluzione del problema sono riportati nel Capitolo 4.

La lista delle stazioni di ricarica ottenuta viene poi filtrata in base alla disponibilità dei dati necessari e alla compatibilità con il veicolo, tramite il codice:

```
chargers = openchargemap_client.get_chargers(route_polyline, max_charger_distance,
    max_chargers)
nodes = []
for charger in chargers:
    current = check_charger(charger)
```

[7] https://openchargemap.org/site/develop/api#/operations/get-poi

```
    if current:
        nodes.append(current)
```

Codice 3.6: Filtraggio stazioni di ricarica

La funzione di controllo utilizzata nel Codice 3.6 è definita come:

```
def check_charger(charger):
    name = charger['AddressInfo']['Title']
    coordinates = (round(charger['AddressInfo']['Latitude'], 5),
        round(charger['AddressInfo']['Longitude'], 5))
    power = None
    for connection in charger['Connections']:
        # Check if required data are aviable
        if connection['PowerKW'] and connection['ConnectionType']['FormalName']:
            # Check if fast type connector is compatible
            if fast_type in connection['ConnectionType']['FormalName']:
                charger_type = connection['ConnectionType']['Title']
                # The real charging power it's the lower between veichle's and
                    charger's ones
                power = min(fast_power, connection['PowerKW'])
                break
            if std_type in connection['ConnectionType']['FormalName']:
                charger_type = connection['ConnectionType']['Title']
                # The real charging power it's the lower between veichle's and
                    charger's ones
                power = min(std_power, connection['PowerKW'])
    # If a compatible charger has been found, then return a RouteNode with the
        collected data
    if power:
        return Node(name, coordinates, charger_type, power)
    else:
        return None
```

Codice 3.7: Controllo stazione di ricarica

3.2.3 Costruzione della matrice delle adiacenze del grafo

La *matrice delle adiacenze* è una struttura dati utile a rappresentare un grafo. In particolare, le informazioni sull'arco che connette un nodo i ad un nodo j sono contenute nella cella (i, j) della matrice. In questo applicativo, ogni cella contiene:

- la coppia $(durata, distanza)$ nel caso in cui l'arco tra i due nodi sia presente e valido;
- la coppia $(-1, -1)$ altrimenti.

Il servizio Distance Matrix API[8] permette di reperire una matrice contenente le informazioni di durata e distanza dei percorsi tra un insieme di luoghi di origine e di destinazione, che in questo caso coincidono entrambi con i nodi del grafo, tramite la richiesta:

```
def get_distance_matrix(self, origins, destinations):
    # Coordinates are formatted to the ';' separeted pattern requested from the
        Distance Matrix API
    formatted_origins = ';'.join('{},{}'.format(coordinate[0],coordinate[1]) for
        coordinate in origins)
    formatted_destinations = ';'.join('{},{}'.format(coordinate[0],coordinate[1]) for
        coordinate in destinations)
    url = 'https://dev.virtualearth.net/REST/v1/Routes/DistanceMatrix'
    params = {
        'key': self.key, # API key
        'travelMode': 'driving', # Travel model (driving/walking/transit)
        'origins': formatted_origins,
        'destinations' : formatted_destinations
    }
    response = requests.get(url, params = params)
    if not response.ok:
        raise requests.RequestException(response.text)
    data = response.json()
    return data['resourceSets'][0]['resources'][0]['results']
```

Codice 3.8: Richiesta Distance Matrix API

La matrice restituita viene poi ispezionata per scartare gli archi non validi (l'API restituisce valori di distanza negativi nel caso in cui non sia possibile calcolare il percorso tra due luoghi) e quelli con tempo di percorrenza superiore al tempo massimo di guida consecutivo desiderato, tramite il codice:

```
results = bingmaps_client.get_distance_matrix(origins = nodes, destinations = nodes)
for entry in results:
    distance = entry['travelDistance']
    time = entry['travelDuration'] / 60
    if time <= 0 or time > max_drive_time:
        route_matrix[entry['originIndex']][entry['destinationIndex']] = (-1, -1)
    else:
        route_matrix[entry['originIndex']][entry['destinationIndex']] = (time,
            distance)
return route_matrix
```

Codice 3.9: Ispezione della matrice delle adiacenze

[8] https://docs.microsoft.com/en-us/bingmaps/rest-services/routes/calculate-a-distance-matrix

3.2.4 Scrittura del file dati AMPL

Una volta elaborati i dati, è necessario scriverli in un formato leggibile dall'ottimizzatore. In questo caso è stato utilizzato AMPL[9], un linguaggio di programmazione matematica sviluppato dai Bell Laboratories per la descrizione di complessi modelli matematici.

In questo applicativo, verrà utilizzato per descrivere insiemi e parametri del problema, come nell'esempio seguente:

```
set nodes := 0 1 2 3 4 5 6 7 /*...other entries...*/ 91 92 93 94 95 96 97 98 ;
param start_node := 0 ;
param end_node := 98 ;
param max_capacity := 60.0 ;
param min_capacity := 3.0 ;
param start_capacity := 60.0 ;
param car_efficiency := 0.15 ;
param max_charging_time := 0.75 ;
param min_charging_time := 0.25 ;
param charger_power :=
0 0
1 11.0
# ...other entries...
97 50.0
98 0
;
param arc_time :=
0 0 0.0
0 1 1.195
# ...other entries...
98 96 1.262
98 97 -1
98 98 0.0
;
param arc_distance :=
0 0 0
0 1 122.264
# ...other entries...
98 96 101.4
98 97 -1
98 98 0
;
```

Codice 3.10: Esempio di file dati AMPL

I parametri `start_node` e `end_node` sono stati aggiunti per permettere all'ottimizzatore di riconoscere la tipologia di un nodo nell'applicazione dei vincoli.

[9] `https://ampl.com/`

3.3 Ottimizzazione Pyomo

3.3.1 Modellazione del problema

Il modello Pyomo è stato definito come astratto, in modo che questo possa essere istanziato con i dati forniti tramite il file AMPL descritto nella Sezione 3.2.4:

```
model = AbstractModel()
```

Codice 3.11: Modello Pyomo

3.3.1.1 Insiemi

Gli insiemi definiti nella Sezione 2.1 sono stati modellati come:

```
model.nodes = Set()
```

Codice 3.12: Insiemi modello Pyomo

3.3.1.2 Parametri

I parametri definiti nella Sezione 2.2 sono stati modellati come:

```
model.max_capacity = Param()
model.min_capacity = Param()
model.start_capacity = Param()
model.car_efficiency = Param()
model.max_charging_time = Param()
model.min_charging_time = Param()
model.charger_power = Param(model.nodes)
model.arc_time = Param(model.nodes, model.nodes)
model.arc_distance = Param(model.nodes, model.nodes)
model.start_node = Param()
model.end_node = Param()
```

Codice 3.13: Parametri modello Pyomo

3.3.1.3 Variabili

Le variabili definite nella Sezione 2.3 sono state modellate come:

```
model.x = Var(model.nodes, model.nodes, domain = Binary)
model.y = Var(model.nodes, domain = Binary)
model.r = Var(model.nodes, domain = NonNegativeReals)
model.C_in = Var(model.nodes, domain = NonNegativeReals)
model.C_out = Var(model.nodes, domain = NonNegativeReals)
```

Codice 3.14: Variabili modello Pyomo

3.3.1.4 Vincoli

I vincoli definiti nella Sezione 2.5 sono stati modellati come:

- Vincoli di flusso 2.5.1

```
def node_flow_rule(model, v):
    outgoing_arcs = sum(model.x[v,i] for i in model.nodes if model.arc_time[v,i]
        != -1)
    incoming_arcs = sum(model.x[i,v] for i in model.nodes if model.arc_time[i,v]
        != -1)
    if v == model.start_node:
        return outgoing_arcs - incoming_arcs == 1
    elif v == model.end_node:
        return outgoing_arcs - incoming_arcs == -1
    else:
        return outgoing_arcs - incoming_arcs == 0
```

Codice 3.15: Vincoli di flusso

- Vincolo di visita nodo 2.5.2

```
def node_visit_rule(model, v):
    if v == model.start_node or v == model.end_node:
        return Constraint.Skip
    else:
        incoming_arcs = sum(model.x[i,v] for i in model.nodes if
            model.arc_time[i,v] != -1)
        return model.y[v] == incoming_arcs
```

Codice 3.16: Vincolo di visita nodo

Il costrutto `Constraint.Skip` consente di indicare al risolutore che non è necessario applicare un vincolo in una determinata situazione.

- Vincolo di tempo di ricarica 2.5.3

```
def min_charge_time_rule(model, v):
    if v == model.start_node or v == model.end_node:
        return Constraint.Skip
    else:
        return model.min_charging_time * model.y[v] <= model.r[v]

def max_charge_time_rule(model, v):
    if v == model.start_node or v == model.end_node:
        return Constraint.Skip
    else:
        return model.r[v] <= model.max_charging_time * model.y[v]
```

Codice 3.17: Vincolo di tempo di ricarica

- Vincolo di batteria alla partenza 2.5.4

```
def starting_battery_rule(model, v):
    if v == model.start_node:
        return model.C_out[v] == model.start_capacity
    else:
        return Constraint.Skip
```

Codice 3.18: Vincolo di batteria alla partenza

- Vincolo di ricarica della batteria 2.5.5

```
def battery_charging_rule(model, v):
    if v == model.start_node or v == model.end_node:
        return Constraint.Skip
    else:
        return model.C_out[v] == model.C_in[v] + model.r[v] *
            model.charger_power[v]
```

Codice 3.19: Vincolo di ricarica della batteria

- Vincolo di batteria in entrata 2.5.6

```
def min_in_battery_rule(model, v):
    if v == model.start_node:
        return Constraint.Skip
    else:
        return model.min_capacity * model.y[v] <= model.C_in[v]

def max_in_battery_rule(model, v):
    if v == model.start_node:
        return Constraint.Skip
    else:
        return model.C_in[v] <= model.max_capacity * model.y[v]
```

Codice 3.20: Vincolo di batteria in entrata

- Vincolo di batteria in uscita 2.5.7

```
def min_out_battery_rule(model, v):
    if v == model.end_node:
        return Constraint.Skip
    else:
        return model.min_capacity * model.y[v] <= model.C_out[v]

def max_out_battery_rule(model, v):
    if v == model.end_node:
        return Constraint.Skip
    else:
        return model.C_out[v] <= model.max_capacity * model.y[v]
```

Codice 3.21: Vincolo di batteria in uscita

- Vincolo di consumo della batteria 2.5.8

```
def battery_consumption1_rule(model, i, j):
    M_1 = model.max_capacity + model.car_efficiency * model.arc_distance[i,j]
    return model.C_in[j] <= model.C_out[i] - model.car_efficiency *
        model.arc_distance[i,j] + M_1 * (1 - model.x[i,j])

def battery_consumption2_rule(model, i, j):
    M_2 = model.max_capacity - model.car_efficiency * model.arc_distance[i,j]
    return model.C_in[j] >= model.C_out[i] - model.car_efficiency *
        model.arc_distance[i,j] - M_2 * (1 - model.x[i,j])
```

Codice 3.22: Vincolo di consumo della batteria

3.3.2 Istanziazione del problema

Il modello astratto deve essere istanziato a tempo di esecuzione a partire dal file AMPL, tramite il codice:

```
model = buildmodel()
instance = model.create_instance(ampl_file)
```

Codice 3.23: Istanziazione del modello Pyomo

3.3.3 Risoluzione del problema

IBM ILOG CPLEX Optimizer v20.1.0.0 [5] è un risolutore di problemi di programmazione matematica di proprietà di IBM. La risoluzione di problemi MILP di CPLEX avviene in due fasi principali:

- *presolve*: il risolutore ispeziona il problema fornito, con l'obiettivo di semplificarlo eliminando in modo intelligente variabili e vincoli non necessari. In questa fase CPLEX tenta inoltre di dimostrare che non esiste una soluzione ammissibile al problema, così che in caso affermativo non sia necessaria la risoluzione;
- *risoluzione*: il risolutore applica l'algoritmo Branch & Cut descritto nella Sezione 1.3.

Una volta istanziato il modello, viene applicato il risolutore CPLEX, tramite il codice:

```
opt = SolverFactory('cplex_persistent')
opt.set_instance(instance)
opt.solve(tee = True)
```

Codice 3.24: Applicazione del risolutore CPLEX

Il parametro `tee`, utilizzato nel Codice 3.24, consente di visualizzare in tempo reale l'evoluzione del tentativo risoluzione del problema.

3.4 Parametri di output

Una volta terminata la risoluzione, nel caso in cui sia stata individuata una soluzione ottima, è possibile visualizzare i valori attribuiti dal risolutore alle variabili decisionali del problema. In questo caso, le variabili `model.x` e `model.r` forniscono le informazioni necessarie a determinare in quali stazioni di ricarica e per quanto tempo sia opportuno effettuare delle soste.

Capitolo 4

Analisi dei risultati

4.1 Istanze utilizzate

L'ottimizzazione CPLEX descritta nella Sezione 3.3 è stata provata su 12 istanze. Per ottenere risultati confrontabili, per tutte le istanze sono stati utilizzati gli stessi parametri (corrispondenti ad un veicolo Tesla Model 3[10]), riportati nella Tabella 4.1.

Parametro	**Valore**
`max_capacity`	57.5
`start_capacity`	57.5
`autonomy`	380.0
`std_type`	IEC 62196-2
`std_power`	11.0
`fast_type`	IEC 62196-3 Configuration FF
`fast_power`	95.0

Tabella 4.1: Parametri del veicolo

La descrizione dei restanti parametri delle istanze è riportata nella Tabella 4.2. Le casistiche sono state raggruppate in base al parametro *maxresults*, definito nel Codice 3.5 della Sezione 3.2.2. Il numero massimo di stazioni di ricarica definisce infatti la dimensione massima del problema che dovrà essere risolto, che è pari a $N = maxresults + 2$.

[10] `https://ev-database.org/car/1555/Tesla-Model-3`

Istanza	Parametro					
	`max results`	`origin`	`destination`	`min charge time`	`max charge time`	`max drive time`
ev50m	50	Padova,IT	Milano,IT	0.25	0.75	2.50
ev50r	50	Padova,IT	Roma,IT	0.25	0.75	2.50
ev50p	50	Padova,IT	Parigi,FR	0.25	0.75	2.50
ev100m_1	100	Padova,IT	Milano,IT	0.25	0.75	2.50
ev100m_2	100	Padova,IT	Milano,IT	0.00	1.00	4.50
ev100r_1	100	Padova,IT	Roma,IT	0.25	0.75	2.50
ev100r_2	100	Padova,IT	Roma,IT	0.00	1.00	4.50
ev100p_1	100	Padova,IT	Parigi,FR	0.25	0.75	2.50
ev100p_2	100	Padova,IT	Parigi,FR	0.00	1.00	4.50
ev500m	500	Padova,IT	Milano,IT	0.25	0.75	2.50
ev500r	500	Padova,IT	Roma,IT	0.25	0.75	2.50
ev500p	500	Padova,IT	Parigi,FR	0.25	0.75	2.50

Tabella 4.2: Descrizione delle istanze

Nella Tabella 4.3 sono fornite le informazioni riguardanti la dimensione dei grafi e dei problemi generati dalle varie istanze.

	Grafo		**Problema**		
Istanza	**Distanza percorso base** $[km]$	**Nodi**	**Vincoli**	**Variabili binarie**	**Variabili continue**
ev50m	246	52	5865	2756	156
ev50r	495	50	5439	2550	150
ev50p	1080	52	5865	2756	156
ev100m_1	246	100	20889	10100	300
ev100m_2	246	100	20889	10100	300
ev100r_1	495	99	20482	9900	297
ev100r_2	495	99	20482	9900	297
ev100p_1	1080	99	20482	9900	297
ev100p_2	1080	99	20482	9900	297
ev500m	246	475	455514	226100	1425
ev500r	495	479	463182	229920	1437
ev500p	1080	485	474804	235710	1455

Tabella 4.3: Dimensione delle istanze

4.2 Risultati ottenuti

La macchina utilzzata per la risoluzione delle istanze dispone di processore AMD Ryzen 7 4700U 2.00 GHz e 16.00 GB di memoria RAM.

Nella Tabella 4.4 sono riportati i risultati ottenuti dal risolutore CPLEX.

	Risoluzione				
Istanza	**Tempo impiegato** [s]	**Valore ottimo** [hh:mm:ss]	**Optimality gap** [%]	**Numero di soste**	**Tempo totale di ricarica** [hh:mm:ss]
ev50m	0,272	02:21:24	0,00	0	00:00:00
ev50r	0,341	05:33:46	0,00	2	00:35:31
ev50p	0,201	infeasible	-	-	-
ev100m_1	0,663	02:21:24	0,00	0	00:00:00
ev100m_2	0,775	02:21:24	0,00	0	00:00:00
ev100r_1	0,720	05:15:14	0,00	2	00:30:30
ev100r_2	3,002	04:55:36	0,00	1	00:15:09
ev100p_1	0,403	infeasible	-	-	-
ev100p_2	13,139	12:25:57	0,00	3	01:21:15
ev500m	24,648	02:21:24	0,00	0	00:00:00
ev500r	16,773	05:15:06	0,00	1	00:24:02
ev500p	28,197	10:52:24	0,00	4	01:14:07

Tabella 4.4: Risultati delle istanze

Nella Figura 4.1 sono riportati, per l'istanza ev100r_1, il percorso base e quello con le soste ottimali individuate dall'applicativo. Le tempistiche di sosta sono state indicate nel formato [hh:mm:ss].

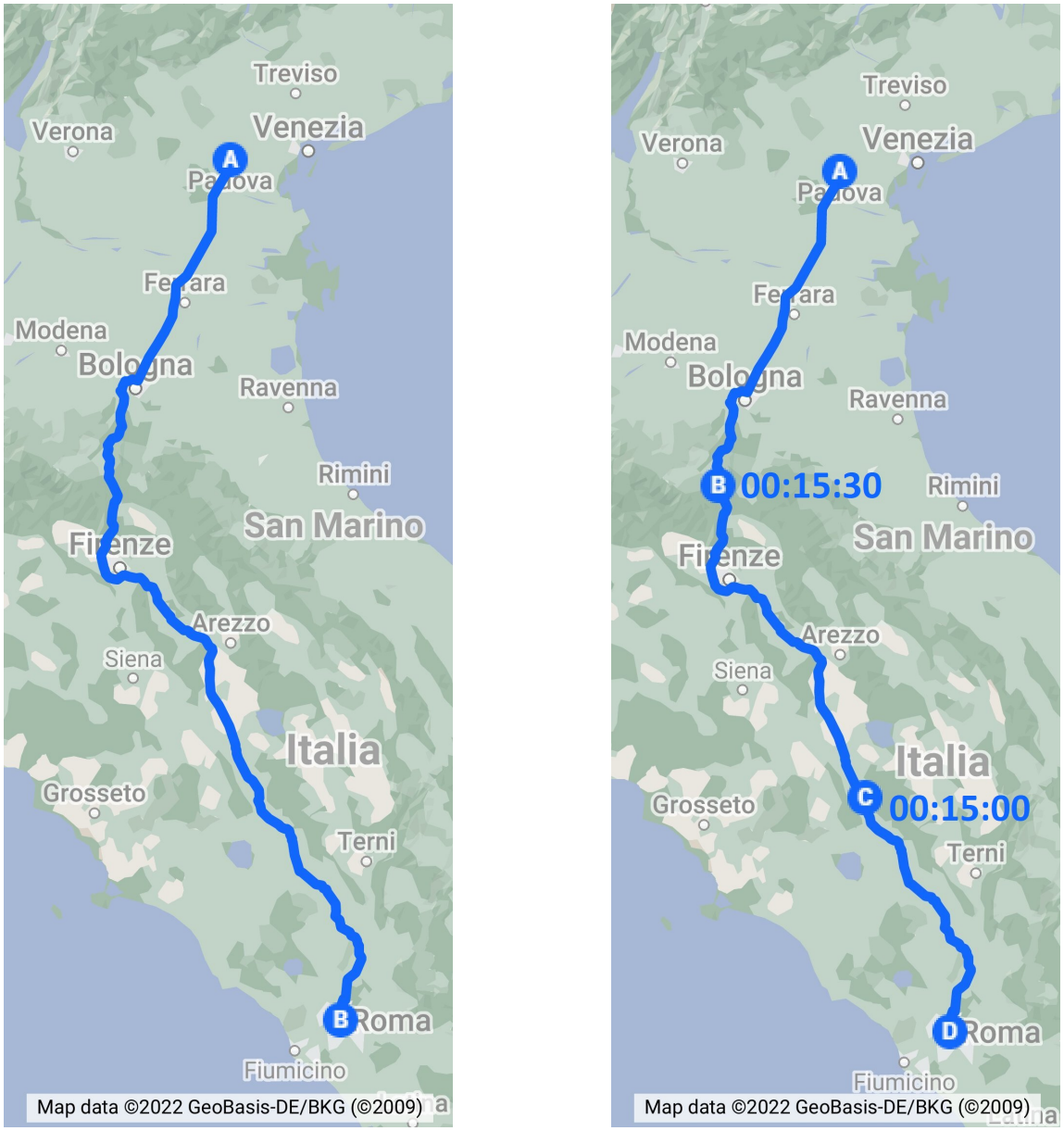

Figura 4.1: Percorso base (a sinistra) e percorso con soste (a destra)

Dai risultati ottenuti è possibile osservare che, dato un tragitto, all'aumentare del parametro *maxresults* aumenti sensibilmente il tempo impiegato da CPLEX per la risoluzione del problema. Questo fenomeno è dovuto al fatto che un sottoinsieme delle variabili e dei vincoli del problema siano legati agli archi del grafo. Il numero degli archi varia infatti in modo quadratico rispetto al numero dei nodi ed è pari, nel caso peggiore in cui non ci siano archi non validi, a $N^2 = (maxresults + 2)^2$.

Nel caso delle istanze ev50p e ev100p_1 è possibile notare come, per tragitti particolarmente estesi, un numero non abbastanza elevato di stazioni di ricarica porti all'impossibilità di trovare una soluzione ammissibile al problema.

Inoltre, all'aumentare del parametro *maxresults* diminuisce il valore ottimo della funzione obiettivo, che corrisponde al tempo totale di percorrenza. Questa correlazione è dimostrabile definendo come F, G gli insiemi delle soluzioni ammissibili dei problemi generati per un determinato tragitto utilizzando come parametro *maxresults* rispettivamente i valori m_f, m_g con $m_f \leq m_g$. Si ha che gli insiemi dei nodi dei grafi generati risultano essere $V_f \subseteq V_g$, da cui $F \subseteq G$. Allora, data funzione obiettivo c, se esiste una soluzione ottima ammissibile $\overline{x_f} \in F$, esiste una soluzione ottima $\overline{x_g} \in G$ tale che $c(\overline{x_g}) \leq c(\overline{x_f})$.

É necessario individuare un compromesso tra il tempo di risoluzione dei problemi e il valore della soluzione ottima individuata. Una soluzione potrebbe essere il calcolo del numero massimo delle stazioni di ricarica in proporzione all'estensione del percorso base relativo all'istanza da risolvere.

Capitolo 5

Conclusioni

Attraverso l'automatizzazione della ricerca del percorso ottimo è possibile semplificare ed incentivare l'utilizzo dei veicoli elettrici per la percorrenza di lunghe tratte. La qualità del miglior tragitto individuabile dal risolutore dipende ancora strettamente dalle caratteristiche del veicolo e dall'infrastruttura di ricarica disponibile, proprio per questo motivo l'ottimizzazione ricopre un ruolo fondamentale nel consentire uno sfruttamento efficiente delle già limitate risorse a disposizione. Il percorso ottimo viene infatti deciso in modo specifico sulla base delle caratteristiche del mezzo che lo deve percorrere e delle preferenze dell'utente, sollevando il guidatore dalla complicata organizzazione di soste e tempistiche di ricarica.

Come evidenziato dai risultati ottenuti dal risolutore CPLEX, le prestazioni di risoluzione sono sufficientemente elevate da consentire che il calcolo del percorso ottimo avvenga in tempi ragionevoli, soprattutto in relazione al tempo totale del tragitto da percorrere. Inoltre, le tempistiche di risoluzione possono essere minimizzate tramite il calcolo del numero massimo delle stazioni di ricarica, e quindi della dimensione del problema da risolvere, in proporzione all'estensione del tragitto da percorrere.

Ulteriori sviluppi dell'applicativo potrebbero essere volti a migliorare l'accuratezza nelle stime del consumo della batteria durante il tragitto e delle tempistiche di ricarica. In particolare, per quanto riguarda l'esaurimento della batteria potrebbero essere presi in considerazione la velocità media del veicolo e l'impatto delle condizioni atmosferiche [6], mentre per le tempistiche di sosta, il rallentamento dato dalla dispersione di energia durante la ricarica [6].

Elenco delle figure

Elenco delle tabelle

Elenco dei codici

Bibliografia

[1] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson e David L. Woodruff. *Pyomo–optimization modeling in python*. Third. Vol. 67. Springer Science & Business Media, 2021.

[2] International Electrotechnical Commission et al. «IEC 62196-3: 2014». In: *Plugs, Socket-Outlets, Vehicle Connectors and Vehicle Inlets–Conductive Charging of Electric Vehicles–Part* 3 (2014).

[3] Matteo Fischetti. *Lezioni di Ricerca Operativa*. 2018.

[4] William E Hart, Jean-Paul Watson e David L Woodruff. «Pyomo: modeling and solving mathematical programs in Python». In: *Mathematical Programming Computation* 3.3 (2011), pp. 219–260.

[5] IBM. *ILOG CPLEX Optimization Studio. User's Manual for CPLEX*. URL: `https://www.ibm.com/docs/en/icos/20.1.0?topic=cplex-users-manual` (visitato il 21/03/2022).

[6] Julio A Sanguesa, Vicente Torres-Sanz, Piedad Garrido, Francisco J Martinez e Johann M Marquez-Barja. «A review on electric vehicles: Technologies and challenges». In: *Smart Cities* 4.1 (2021), pp. 372–404.