POLITECNICO DI TORINO

Master's Degree in AUTOMOTIVE ENGINEERING

Master's Degree Thesis

Multi-criteria Electric Vehicle Routing Planning

Supervisors

Prof. Angelo BONFITTO

Candidate

Hongrun ZHU

May 2023

Summary

Electric vehicle (EV) route planning is widely studied recent days. The key problem is always how to deal the different criteria, drivers may choose the shortest traveling time or prefer a path with the least energy consumption. Of course, it is more common to consider both and choose the best. Doing this requires optimizing for multiple criteria at the same time.

A common approach is to use a Dijkstra's algorithm with a Pareto set. Obviously, as the size of the map increases and the criteria considered increase, the resulting Pareto set will become unmanageable. Therefore, how to deal with the Pareto set becomes the key to the route planning problem of electric vehicles. For example, use a comprehensive criterion instead of multiple criteria, such as using the 'cost' criterion to integrate energy consumption and charging price. But what combination will yield the best results? This thesis explores this question.

In this thesis, a classic Dijkstra's algorithm with Pareto sets is constructed in Matlab. It is used to compare the impact on the final output when different types of criteria are used and different linear combination ratio of these criteria too. Then, as temperature has an impact on energy request, this can affect routing. At the end of the thesis, how temperature can cause bad routes will be examined, along with possible solutions to deal with the consequent energy request variation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Along with the strengthening of people's awareness of environmental protection, more and more electric vehicles have appeared on the market and on traffic roads. On the one hand, these electric vehicles have only a very limited range due to the limitation of the battery energy, and the use of traditional routing planning may cause the electric vehicle to break down on the road. On the other hand, by increasing the optimization criteria, energy consumption can be further reduced, thereby reducing the charging time during the journey, and a comprehensively excellent route can be obtained with two birds with one stone.

The existing research on the algorithm of routing planning for electric vehicles still focuses on the algorithm itself, that is, the calculation speed of the algorithm. However, there is still a lack of research on the impact of routing on travel time and energy consumption. This thesis will fill this gap, exploring what routes the algorithm will give under different criteria, how these routes affect energy consumption, traveling time and distance, and how to choose better criteria.

In this chapter, we will give a comprehensive introduction to the basics of routing planning on transportation networks. There are four sections in this chapter. In the **first section**, the relevant literature on this issue will be studied. It can be seen that most of the research is still focused on improving the computing speed rather than thinking about scenarios related to electric vehicles. The **second section** focuses on several methods of expressing and storing map data. For different representation and storage methods, the algorithms developed and used are quite different. Therefore, it is particularly important to choose a suitable method of expressing and storing map data at the beginning. The **third section** introduces Dijkstra's algorithm, which is the most widely used and is the basis of most modern navigation algorithms. The algorithm developed in this thesis is also based on Dijkstra's algorithm. The **fourth section** will introduce the very commonly used Fibonacci heap in the Dijkstra's algorithm. Due to the difficulty of developing the Fibonacci heap, the appendix of this thesis will also directly give the Matlab code

of the Fibonacci heap.

1.1 Literature analysis

As an important part of graph theory, the shortest path problem has been extensively studied in the field of mathematics and computer algorithms. Since Edsger W. Dijkstra published his famous Dijkstra's algorithm [1], due to its excellent performance, many follow-up studies have focused on trying to expand the scope of application of Dijkstra's algorithm and further improve its performance, especially for Research on real traffic road map [2]. The algorithm does not try to "spot" the target directly. Instead, the only thing considered when determining the next "current" intersection vertex is the distance from the starting vertex. Therefore, the algorithm is extended to interactively consider all the nearest nodes with the shortest path distance from source to target. This insight shows how algorithms naturally find the shortest paths. However, this can also reveal a weakness of the algorithm: it is relatively slow for certain topologies.

Today, with the development of electric vehicles, more and more research on the shortest path problem begins to consider the range and charging of electric vehicles. This requires real-time tracking of state of charge (SoC). The usual practice is to expand the Pareto set to obtain the state of charge for the traditional Dijkstra's algorithm to achieve multi-object or multi-criteria electric vehicle routing.

Some algorithmic research on EV routing planning is mainly aimed at the energy recovery of EVs [3]. The Dijkstra's algorithm and the A* algorithm developed based on the Dijkstra's algorithm do not allow negative cost on the edge. The purpose of these studies is to allow the Dijkstra's algorithm and subsequent algorithms based on Dijkstra's to allow negative cost in terms of battery energy.

Other studies hope to find ways to plan better paths. For example, the global path planning is divided into two parts [4], in this way, the obtained path can have two completely different search objectives in two stages. For example, when the battery energy is sufficient, it is hoped to find the path that consumes the least time. When the battery energy is almost exhausted, the goal is to reduce energy consumption.

Obviously, how to improve the speed of the route planning algorithm in transportation networks is still one of the focuses of current research [5]. Improving the data structure is the most direct way, Dijkstra's algorithm use a priority queue Q of vertices ordered by (tentative) distances from Source s. Then replacing the traditional priority queue Q with the Fibonacci heap becomes an effective method to improve the operation speed [6].

Another fundamental way to reduce running time is using bidirectional search by effectively reduce the search space. This method searches from start to end and

end to start at the same time. When intending to use this method save computing time, it is best to use the forward/backward star representation introduced in **Section 1.2** to store map data. Usually it reduces the search space by more than half, so it also cuts the computation time by half.

Since the Dijkstra's algorithm does not consider the direction, there are some Goal-Directed Techniques to guide the search toward the target, thereby reducing the search space and achieving the effect of speeding up. The most widely used goal-directed shortest path algorithm is A* search [7]. It uses a potential function to make the vertices that are closer to the target t to be scanned earlier during the algorithm. Like bidirectional search, it achieves the purpose of reducing computing time by reducing the search space. It is generally believed that when using A* algorithm, the use of bidirectional search will not further reduce the calculation time.

Another widely used goal-directed shortest path algorithm is the Arc Flags approach [8]. This algorithm does a pre-processing to the map, and partitions the graph into K cells that are roughly balanced. In the arc-flag method at each arc a a vector fa of arc-flags is stored such that fa[i] indicates if a is on a shortest path into cell i.

Separator-Based Techniques[9] and Bounded-Hop Techniques[10] are also common ways to speed up Dijkstra's algorithm. Separator-Based Techniques sets the separator through pre-processing to separate the graphs. Reduces computation time by computing shortest paths inside delimiters as well as delimiter-to-delimiter shortest paths.

The idea behind bounded-hop techniques is to pre-compute distances between pairs of vertices, implicitly adding "virtual shortcuts" to the graph. Queries can then return the length of a virtual path with very few hops. Furthermore, they use only the pre-computed distances between pairs of vertices, and not the input graph. A naive approach is to use single-hop paths, i. e., pre-compute the distances among all pairs of vertices $u, v \in V$. A single table lookup then suffices to retrieve the shortest distance.

1.2 Network representations

To perform a shortest path algorithm on a transportation networks, it is needed to know how the network is represented. Normally there are two types of information which need to be stored. **a. the network topology**, the network's node and arc structure and **b. the cost function**, such as distance, travelling time, and energy consumption associated with the network's nodes and arcs.

There are five typical network representations:

- **Node-Arc Incidence Matrix**

- **Node-Node Adjacency Matrix**
- **Adjacency Lists**
- **Forward and Reverse Star Representations**
- **Compact Forward and Reverse Star Representation**

Node-Arc Incidence Matrix stores the network as an $n \times m$ matrix $\mathcal{N}$. It contains one row for each node of the network and one column for each arc. The column corresponding to arc (i, j) has only two nonzero elements: It has a '+1' in the row corresponding to node i and a '-1' in the row corresponding to node j.

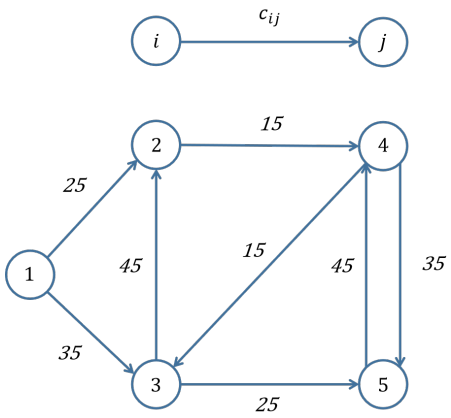

Figure 1.1: Graph simple

For a graph like the one in Figure 1.1, the Node-Arc Incidence Matrix data is like:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 \end{bmatrix}$$

It's easy to find that the Node-Arc Incidence Matrix has a very special structure:

1. Only 2 m out of its nm entries are nonzero;
2. All of its nonzero entries are +1 or -1;
3. Each column has exactly one +1 and one -1;
4. The number of + 1's in a row equals the out-degree of the corresponding node and the number of -1's in the row equals the in-degree of the node.

Because the Node-Arc Incidence Matrix $\mathcal{N}$ contains so few nonzero coefficients, the incidence matrix representation of a network is not space efficient. Therefore, this structure rarely produces efficient algorithms.

Node-Node Adjacency Matrix stores the network as an $n \times n$ matrix $\mathcal{H} = \{h_{ij}\}$. This matrix has a row and a column corresponding to every node. Its ijth entry h_{ij} equals 1 if $(i, j) \in A$ and equals 0 otherwise.

For a graph like 1.1, the Node-Node Adjacency Matrix data is like:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

To store the arc costs as well as the network topology, it can use a additional $n \times n$ matrix $\mathcal{L}$.

The adjacency matrix has n^2 elements, only m of which are nonzero. Consequently, this representation is space efficient only if the network is sufficiently dense; For sparse networks this representation wastes considerable space. The simplicity of the adjacency representation permits us to use it to implement most network algorithms rather easily.

It can determine the cost of any arc (i, j) simply by looking up the ij th element in the matrix $\mathcal{L}$; It can obtain the arcs emanating from node i by scanning row i, if the jth element in this row has a nonzero entry, (i, j) is an arc of the network; Similarly, it can obtain the arcs entering node j by scanning column j, if the ith element of this column has a nonzero entry, (i, j) is an arc of the network. These steps permit us to identify all the outgoing or incoming arcs of a node in time proportional to n. For **dense networks** it can usually afford to spend this time to identify the incoming or outgoing arcs; while for **sparse networks** these steps might be the bottleneck operations for an algorithm.

Adjacency list stores the **node adjacency list** of each node as a singly **linked list**. A linked list is a collection of cells each containing one or more fields. The node adjacency list for node i will be a linked list having $|A(i)|$ cells and each cell will correspond to an arc $(i, j) \in A$. Each cell corresponding to the arc (i, j) will have:

1. One data field will store node j;
2. One other data fields to store the **arc cost** c_{ij};
3. One additional link field, which stores a pointer to the next cell in the adjacency list.

If a cell happens to be the last cell in the adjacency list, by convention we set its link to value zero. It also needs an array of pointers that point to the first cell in each linked list. For a graph like Figure 1.1, the Adjacency Lists data is like Figure 1.2.

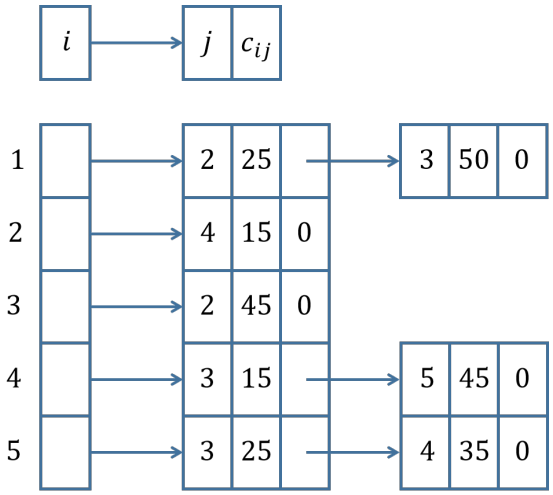

Figure 1.2: Adjacency lists

Forward Star Representations stores the node adjacency list of each node. It is similar to the adjacency list representation, but instead of maintaining these lists as linked lists, it stores them in a **single array**. It provides us with an efficient means for determining the set of outgoing arcs of any node. To develop this representation:

1. It first associates a unique sequence number with each arc, thus defining an ordering of the arc list.
2. First those emanating from node 1, then those emanating from node 2, and so on.
3. It numbers the arcs emanating from the same node in an arbitrary fashion.
4. It then sequentially stores information about each arc in the arc list.
5. It stores the tails, heads and costs of the arcs in three arrays: tail, head and cost. If arc (i, j) is arc number 20, we store the tail, head and cost data for this arc in the array positions tail(20), head(20) and cost(20).

It also maintain **a pointer** with each node i, denoted by $point(i)$, that indicates the smallest-numbered arc in the arc list that emanates from node i. If node i has no outgoing arcs, we set $point(i)$ equal to $point(i + 1)$. Therefore, the

outgoing arcs of node i at positions $point(i)$ to $point(i+1)-1$ in the arc list. If $point(i) > point(i+1)-1$, node i has no outgoing arc. For consistency, It sets $point(1) = 1$ and $point(n+1) = m+1$. For a graph like Figure 1.1, the Forward star representation data is like Figure 1.3.

	point
1	1
2	3
3	4
4	5
5	7
6	9

	tail	head	cost
1	1	2	25
2	1	3	35
3	2	4	40
4	3	2	10
5	4	3	30
6	4	5	60
7	5	3	20
8	5	4	50

Figure 1.3: Forward star representation

Reverse star representation is used to determine the set of incoming arcs of any node efficiently. For a graph like Figure 1.1, the Reverse star representation data is like Figure 1.4. To develop a reverse star representation:

1. It examines the nodes $i = 1$ to n in order and sequentially store the heads, tails and costs of the incoming arcs at node i.
2. It maintains a reverse pointer with each node i, denoted by $rpoint(i)$, which denotes the first position in these arrays that contains information about an incoming arc at node i.
3. If node i has no incoming arc, it sets $rpoint(i)$ equal to $rpoint(i+1)$.
4. For consistency, it sets $rpoint(1) = 1$ and $rpoint(n+1) = m+1$.
5. As before, it stores the incoming arcs at node i at positions $rpoint(i)$ to $rpoint(i+1)-1$.

Observe that by storing both the forward and reverse star representations, it will maintain a significant amount of duplicate information. This duplication can be avoid by storing arc numbers in the reverse star instead of the tails, heads and costs of the arcs.

So instead of storing the tails, heads and costs of the arcs, **Compact Forward and Reverse Star Representation** simply stores arc numbers; and once knowing

	tail	head	cost
1	3	2	45
2	1	2	25
3	1	3	35
4	4	3	15
5	5	3	25
6	5	4	35
7	2	4	15
8	4	5	45

	point
1	1
2	1
3	3
4	6
5	8
6	9

Figure 1.4: Reverse star representation

the arc numbers, it can always retrieve the associated information from the forward star representation. It stores arc numbers in an array trace of size m.

	point
1	1
2	3
3	4
4	5
5	7
6	9

	tail	head	cost
1	1	2	25
2	1	3	35
3	2	4	40
4	3	2	10
5	4	3	30
6	4	5	60
7	5	3	20
8	5	4	50

	trace
1	4
2	1
3	2
4	5
5	7
6	8
7	3
8	6

	rpoint
1	1
2	1
3	3
4	6
5	8
6	9

Figure 1.5: Compact Forward and Reverse Star Representation

For a graph like Figure 1.1, the Compact Forward and Reverse Star Representation data is like Figure 1.5. Due to the relatively simple structure, in less complex application scenarios, usually uses the Node-Node Adjacency Matrix to represent the transportation networks, like what Matlab build-in function will do. Also Matlab provides functions to convert the data to a Node-Node Adjacency Matrix. If do not consider the bi-directional search, Forward Star Representation

will provide good performance, while, obviously, the Compact Forward and Reverse Star Representation will do well in bi-directional search.

1.3 Dijkstra's algorithm

The Dijkstra's algorithm is a classic method for solving the shortest path problems. In real life, people often encounter the problem of how to travel from one place to another through the shortest path. For example in a map like Figure 1.6, how to travel from Turin to Venice.

Figure 1.6: Map example

If one wants to use computer algorithms to solve this problem, it must first abstract the map. From the most simple point of view, we can abstract the city as a vertex, and the path from city to city as an edge. This results in a directed graph like Figure 1.7.

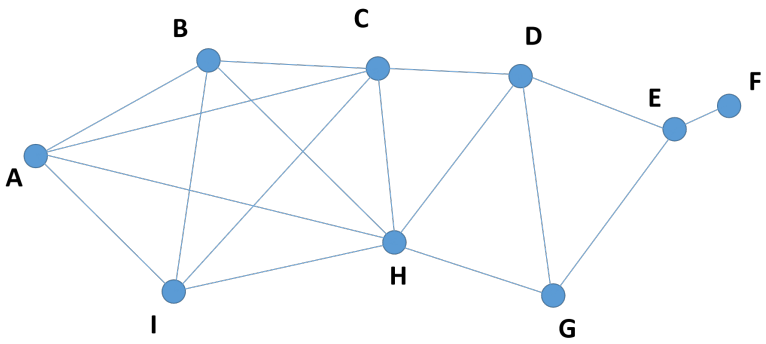

Figure 1.7: Graph example

For a directed weighted graph $G = (V, E, f)$, where:

- V is the set of vertices,
- E is the set of oriented edges,
- f is the cost function $f : E \mapsto \mathbb{R}$ which assigns a numeric value (normally the distance or travelling time) to an edge $e \in E$.

Given a graph $G = (V, E, f)$, the path $P = (v_1, ..., v_n) \in V_n$ is a sequence of vertices from a vertex $v_1 \in V$ to a vertex $v_n \in V$ such that exists an edge $e_i = (v_i, v_{i+1}) \in E$ for $i = 1, ..., n-1$ where $n = |P|$ is a number of vertices included in the path P. Then, we define the cost of the path as $c(P) = \sum_{i=1}^{n-1} f(e_i) \in \mathbb{R}$.

Shortest Path: Given a graph $G = (V, E, f)$, the shortest path $P^*_{u,v} \in G$ is a path $P = (u, ..., v)$ from a vertex $u \in V$ to a vertex $v \in V$ with the smallest cost $c(P^*_{u,v}) = min(c(P)) \in \mathbb{R}$ of all possible paths $P \in G$ from u to v.

Using **Dijkstra's algorithm**, we can solve the shortest path problem. A pseudo code version of the Dijkstra's algorithm is given in Algorithm 1. Besides calculating the distance between s and every other vertex it also outputs an implicit representation of a shortest path tree, containing a shortest path from s to every other vertex.

Algorithm 1 Dijkstra's Algorithm

```
1:  Input: Graph G = (V, E), edge cost c, source vertex s
2:  Data: Priority queue Q
3:  Output: Distances from s given by dist[ · ], shortest-path tree given by par-
    ent[ · ]
4:  for each v ∈ V do                                          ▷ initialization
5:      dist[v] ← ∞
6:      parent[v] ← ⊥
7:  dist[s] ← 0
8:  Q.insert(s, dist[s])
9:  while not Q.isEmpty() do                                   ▷ main loop
10:     u ← Q.deleteMin()
11:     for each (u, v) ∈ E do
12:         if dist[v] > dist[v]+c((u, v)) then
13:             dist[v] ← dist[v]+c((u, v))
14:             parent[v] ← u
15:             if Q.contains(v) then
16:                 Q.decreaseKey((v, dist[v]))
17:             else
18:                 Q.insert((v, dist[v]))
```

As input the algorithm gets a graph $G = (V, E)$, an edge cost $c : E \to \mathbb{R} \geq 0$ and a source vertex s. The two arrays dist[·] and parent[·] are the output of the algorithm, which dist[v] array gives the minimum distance (or any other optimal object set in cost function) from source s each vertex v, and parent[v] array gives the previous vertex of this vertex in the shortest path from the source s to this vertex v. This means that for every vertex $v \in V$ reachable from s, which is equivalent to dist[v] $< \infty$, the shortest path from s to v can be reconstructed by following the pointers given by parent[·] from v to s. Thus $SP_w(s, v) = (s = v_1, v_2, ..., v_k = v)$ is given by $v_i - 1 := parent[v_i]$ for all $1 < i \leq k$.

The algorithm makes use of a priority queue data structure Q. This queue maintains key-value pairs, where the value represents a vertex $v \in V$ and the key represents the tentative distance from s to v. Implementation details of the priority queue are not important for the understanding of Dijkstra's algorithm and there are many other alternatives to priority queues. However, the priority queue normally match the following interface.

- isEmpty() returns a Boolean value, which is false if and only if the queue holds at least one element.
- deleteMin() returns the vertex v associated with the minimal key currently contained in the queue. Afterwards v is removed from the queue.
- contains(v) returns a Boolean value, which is true if and only if v is held by the queue.
- decreaseKey(v, d) sets the key associated with v to d. Preconditions for this operation are, that v is already contained in the queue and that the key currently associated with v is greater than or equal to d.
- insert(v, d) inserts v into the queue and sets its key to d. Precondition for this operation is, that v is not contained in the queue.

The basic approach of Dijkstra's algorithm is to grow a graph-theoretic circle around s, such that for every vertex v contained in this circle the correct distance form s to v is known and held in dist[v]. Furthermore parent[·] implies a correct shortest path tree for all vertices within this circle.

The algorithm starts by initializing the arrays dist[·] and parent[·], as well as the priority queue Q. Since s is the only vertex for which the distance from s is known at the beginning, dist[s] is set to 0. For all other vertices it is set to dist[v] $= \infty$. Accordingly, the parent array is initialized with all entries set to $\perp$, representing an empty shortest path tree. The queue Q is initialized by inserting the source vertex s into it. For a Graph like Figure 1.7, for source $s \leftarrow A$, we then have a dist[·] and a parent[·] like the matrices below:

$$\begin{bmatrix} A & 0 \\ B & \infty \\ C & \infty \\ D & \infty \\ E & \infty \\ F & \infty \\ G & \infty \\ H & \infty \\ I & \infty \end{bmatrix} \begin{bmatrix} A & \perp \\ B & \perp \\ C & \perp \\ D & \perp \\ E & \perp \\ F & \perp \\ G & \perp \\ H & \perp \\ I & \perp \end{bmatrix}$$

After the initialization, the main loop of the algorithm begins in line 9 of algorithm 1. As long as there are vertices contained in Q, the vertex u with minimal key, which means vertex u having the minimal distance from s, will be removed from Q.

Once a vertex has been removed from Q it is called **settled**. The settled vertex will not be visited again by this algorithm, this is the main reason why Dijkstra's algorithm is fast. At this point, dist[u] contains the correct minimum distance from s to u and parent[u] implies a shortest path from s to u. Then, the algorithm examines all outgoing edges of u. This step is not directly reflected in the algorithm, but implicit in the process of settled vertex u.

For every such edge $e = (u, v)$ it is checked if the shortest path from s to u extended by the edge e yields a shorter distance from s to v than the current tentative distance dist[v]. If this is the case, the edge e is relaxed. This means that the tentative distance of v is set to dist[u]+$w((u, v))$. In next step, parent[v] is set to u, since we reached v via u.

Finally, v is either inserted into Q or its key is decreased, depending on it being previously contained in Q or not. Once a vertex got inserted into Q it is called visited.

As starting from vertex A, it should check the arcs $(a, b), (a, c), (a, h)$ and (a, i), like Figure 1.8, and renew the dist[$\cdot$] and parent[$\cdot$]:

$$\begin{bmatrix} A & 0 \\ B & 145 \\ C & 230 \\ D & \infty \\ E & \infty \\ F & \infty \\ G & \infty \\ H & 245 \\ I & 172 \end{bmatrix} \begin{bmatrix} A & \perp \\ B & A \\ C & A \\ D & \perp \\ E & \perp \\ F & \perp \\ G & \perp \\ H & A \\ I & A \end{bmatrix}$$

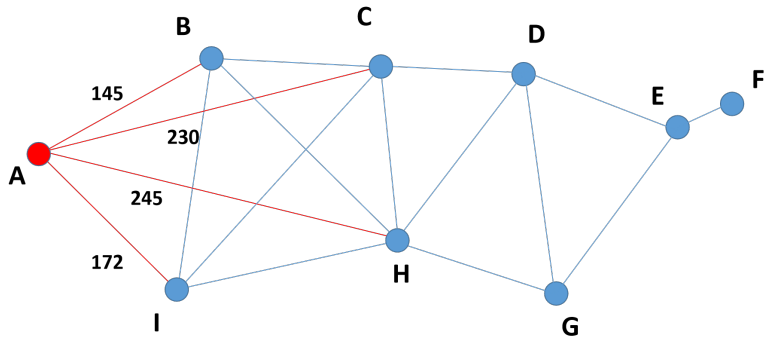

Figure 1.8: Checking Source Vertex

As dist[B] is minimum for any vertex in queue Q, the algorithm will set $u \leftarrow B$ then, Searching the arcs connected to vertex B and renew the dist[·] and parent[·] again, like showing in Figure 1.9 Continuing this process, it will eventually get the shortest path from Turin (vertex A) to Venice (vertex F).

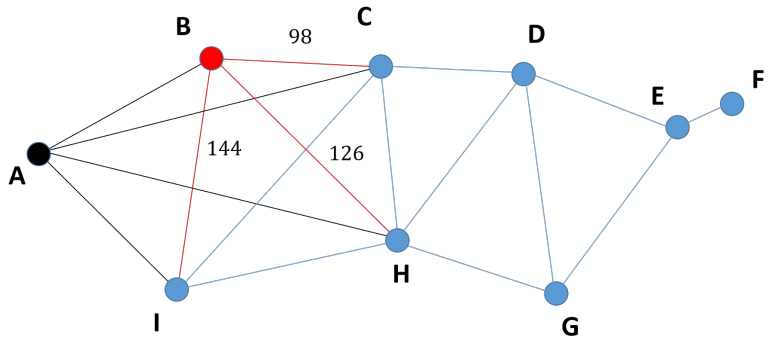

Figure 1.9: Checking Vertex u

If only want to search for the shortest path between two vertices, it can end the loop when judging if $u ==$ target t. This will reduce the search time slightly.

1.4 Fibonacci heap

It is obvious that how the queue Q is constructed is critical to the speed of the algorithm. Fredman and Tarjan [11] introduced **Fibonacci heaps**. Their paper also describes the application of Fibonacci heaps to the problems of single-source shortest paths, all-pairs shortest paths, weighted bipartite matching, and the minimum-spanning-tree problem.

A Fibonacci heap is a collection of rooted trees that are min-heap ordered. That

is, each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. The Fibonacci heap needs to provide several functions:

- **Creating a new Fibonacci heap**

  To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where $H.n = 0$ and $H.min = NIL$; there are no trees in H. Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$. The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

- **Inserting a node**

  The following algorithm 2 inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that x.key has already been filled in.

  Lines 1–4 initialize some of the structural attributes of node x. Line 5 tests to see whether Fibonacci heap H is empty. If it is, then lines 6–7 make x be the only node in H's root list and set $H.min$ to point to x. Otherwise, lines 8–10 insert x into H's root list and update $H.min$ if necessary. Finally, line 11 increments $H.n$ to reflect the addition of the new node.

Algorithm 2 FIB-HEAP-INSERT(H, x)

```
 1: x.degree = 0
 2: x.p = NIL
 3: x.child = NIL
 4: x.mark = FALSE
 5: if H.m == NIL
 6:     create a root list for H containing just x
 7:     H.min = x
 8: else insert x into H's root list
 9:     if x.key < H.min.key
10:         H.min = x
11: H.n = H.n + 1
```

- **Finding the minimum node**

  The minimum node of a Fibonacci heap H is given by the pointer $H.min$, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

Algorithm 3 FIB-HEAP-UNION(H_1, H_2)

```
1: H =MAKE-FIB-HEAP()
2: H.min = H_1.min
3: concatenate the root list of H_2 with the root list of H
4: if(H_1.min == NIL) or (H_2.min ≠ NIL and H_2.min.key < H_1.min.key)
5:     H.min = H_2.min
6: H.n = H_1.n + H_2.n
```

Algorithm 4 FIB-HEAP-EXTRACT-MIN(H)

```
1: z = H.min
2: if z ≠ NIL
3:     for each child x of z
4:         add x to the root list of H
5:         x.p = NIL
6:     remove z from the root list of H
7:     if z == z.right
8:         H.min = NIL
9:     else H.min = z.rihgt
10:        CONSOLIDATE(H)
11:    H.n = H.n − 1
12: return z
```

- **Uniting two Fibonacci heaps**

  The algorithm 3 unites Fibonacci heaps H_1 and H_2, destroying H_1 and H_2 in the process. It simply concatenates the root lists of H_1 and H_2 and then determines the new minimum node. Afterward, the objects representing H_1 and H_2 will never be used again.

  Lines 1–3 concatenate the root lists of H_1 and H_2 into a new root list H. Lines 2, 4, and 5 set the minimum node of H, and line 6 sets $H.n$ to the total number of nodes. Line 7 returns the resulting Fibonacci heap H. As in the FIB-HEAP-INSERT procedure, all roots remain roots.

- **Extracting the minimum node**

  The process of extracting the minimum node algorithm 4 is the most complicated of the operations presented in Fibonacci heap. It is also where the delayed work of consolidating-trees in the root list finally occurs. The following pseudo-code extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also calls the auxiliary procedure CONSOLIDATE.

  FIB-HEAP-EXTRACT-MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

To use Fibonacci heap as a priority queue, it is needed to use Fibonacci heap nodes to save vertex data. The Fibonacci heap node object only needs to provide the function to create the Fibonacci heap node and the properties used to save specific data:

- **parent**: pointer to parent
- **child**: pointer to one of the children
- **left**: pointer to left siblings
- **right**: pointer to right siblings
- **degree**: number of children
- **mark**: indicates whether node x has lost a child since the last time x was made the child of another node
- **key**: key of the node, it is used to save the cost of vertex

- **value**: value of the node, it is used to save the ID of the vertex
- **nil**: it is used to indicate whether this Fibonacci heap node saves data

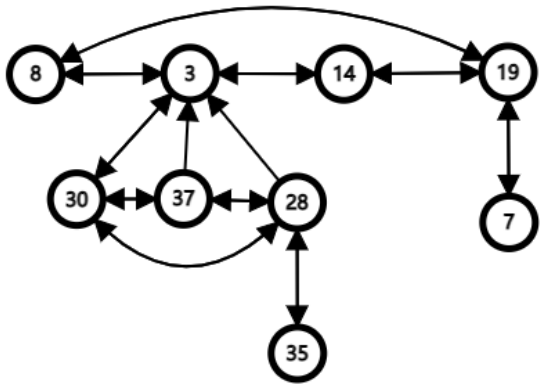

Figure 1.10: Fibonacci heap

Figure 1.10 gives an example of a Fibonacci heap structure. The circle represents a Fibonacci heap node, the number inside is the key of the node and the black arrow means there is a pointer points to this node.

Chapter 2

Electric Vehicle Route Planning Algorithm

As mentioned before, Dijkstra's algorithm is very suitable for solving the shortest path problem. But its limitations are also very obvious. It can only track one criteria. When considering electric vehicles, because of the limitation of the battery energy of the vehicle, it is necessary to expand the Dijkstra's algorithm.

In this chapter, the electric vehicle routing problem is first introduced. In order to enable EV routing for long-distance travel, charging station information must be added to the map so that the vehicle can be recharged. At the same time, we also need electric vehicle models to calculate the energy consumption and driving time of each road section. In the second section, Dijkstra's algorithm will be extended to solve this problem. Pseudo-code is provided, and points to be noticed when implementing this pseudo-code are introduced.

2.1 Electric Vehicle Route Planning Problem

Also for a map like Figure 1.6, we still want to travel from Turin to Venice, but what changes when we drive an electric vehicle? There are two main differences:

1. Vehicle must be recharged on the travel due to battery constraints
2. Even without considering traffic conditions, travel distance and travel time are not directly proportional due to the choice of charging stations.

This requires us to always pay attention to the state of charge of the battery of electric vehicles and to track multiple criteria such as distance, energy consumption and traveling time at the same time. The abstraction of the map will also change, the most important thing is to add charging stations to the map. And in addition

to the length of each edge, it is also necessary to give the driving time and energy consumption used to pass the edge. Also there are other factors may have influence on vehicle's battery energy, like environment temperature, traffic condition and etc. For the previous example, we can assume that there are convenient supercharging stations in provincial capital cities. Then the graph will be like Picture 2.1, which green vertices represents the supercharging charging station.

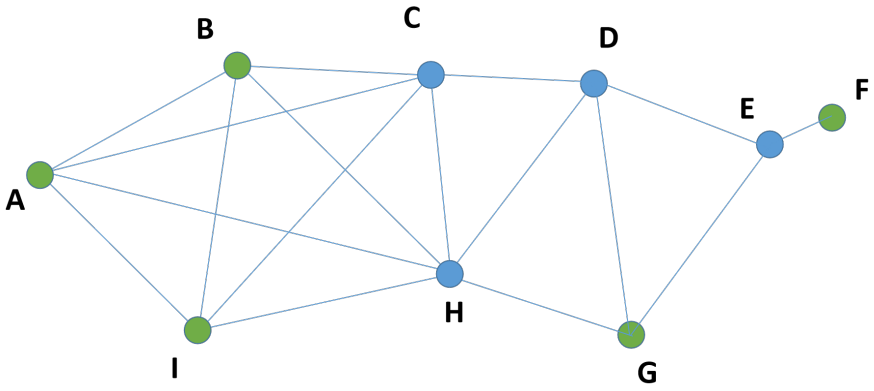

Figure 2.1: Graph with charging station

The actual map will obviously have more vertices and edges, but the resulting form will be the same. From this we get a directed weighted graph $G = (V, V_c, E, f_c, f_e)$, where:

- V is the set of vertices,
- V_c is the set of charging station vertices, which $V_c \subset V$,
- E is the set of oriented edges,
- f_c is the cost function $f_c : E \mapsto \mathbb{R}$ which assigns a numeric value (normally the distance or traveling time or the linear combination of two) to an edge $e \in E$.
- f_e is another cost function $f_e : E \mapsto \mathbb{R}$ which assigns a numeric value (energy consumption) to an edge $e \in E$. This is used to limit the feasible paths for electric vehicles.

Given a graph $G = (V, V_c, E, f_c, f_e)$, the path $P = (v_1, ..., v_n) \in V_n$ is a sequence of vertices from a vertex $v_1 \in V$ to a vertex $v_n \in V$ such that exists an edge $e_i = (v_i, v_{i+1}) \in E$ for $i = 1, ..., n-1$ where $n = |P|$ is a number of vertices included in the path P. Then, we define the cost of the path as $c(P) = \sum_{i=1}^{n-1} f(e_i) \in \mathbb{R}$ for both f_c and f_e.

After that, we can give the definition of the **feasible path** as: given a path $P = (v_1, ..., v_n) \in V_n$, If the energy consumption of the path $e(P) = \sum_{i=1}^{n-1} f_e(e_i) \in \mathbb{R}$ is less than the initial energy of the vehicle B_{init} + the total charged energy of the vehicle $B_{charged\_total}$ - the minimum allowable energy of the vehicle B_{bott_limit}, then this path is called a feasible path. In other words, $P \subset P_{feasible}$ when:

$$B_{init} + B_{charged\_total} > B_{bott\_limit}$$

Shortest Path: Given a graph $G = (V, V_c, E, f_c, f_e)$, the shortest path $P^*_{u,v} \in G$ is a feasible path $P_{feasible} = (u, ..., v)$ from a vertex $u \in V$ to a vertex $v \in V$ with the smallest cost $c(P^*_{u,v}) = min(c(P)) \in \mathbb{R}$ of all possible feasible paths $P_{feasible} \in G$ from u to v.

2.2 Electric Vehicle Route Planning Algorithm

To solve the Electric Vehicle Route Planning Problem, it is needed to extend the basic Dijkstra's algorithm to at least trace one more criterion: the SoC (State of Charge) of the electric vehicle. A common approach for solving optimization problems with multiple criteria is to utilize Pareto sets, which contain all Pareto-optimal solutions. But it should be noted that due to the complexity of the problem, it is unrealistic to consider all Pareto optimal solutions. Preserving the Pareto set but only considering linear combinations of each criterion is a more appropriate approach. So instead of maintaining a simple label of a single cost value, the extended Dijkstra's algorithm or Electric Vehicle Route Planning Algorithm needs to maintain a label-set for every vertex which include all criteria that need to be considered.

Label-Sets: Dijkstra's algorithm only uses one label with single criterion for each vertex v represented by dist[v] and parent[v]. In contrast to that, a label-set labelSet[v] is used for every vertex v if multiple criteria are given. This label-set contains all tentative Pareto-optimal s-v-paths. Of course, it is unrealistic to traverse all the paths, and then a preliminary selection of paths is required. The label-set object needs to implement the following three functions:

- **Key(labelSet)** returns a key based on the first unsettled label contained in the label-set, i.e. the label with the smallest driving time of all unsettled labels contained in the label-set. Since we will not consider all Pareto sets, which label is selected first is crucial to the final path. As key we only use the driving time, energy consumption or distance, SoC is used in order to break ties.
- **Settle(labelSet)** returns the smallest (wrt. key) unsettled label contained in the label-set. Additionally, this label gets marked as settled. This can be

accomplished in a variety of ways. In Matlab, we use the simplest way to create another label-set array to store all settled labels, and delete them from the original label-set array.

- **HasUnsettledLabels(labelSet)** returns a Boolean value, which is true if the label-set contains one or more unsettled labels. This is the case if the label set's internal pointer points to an actual label. The label set contains no unsettled labels if the internal pointer points behind the array are used to store the contained labels.

Just like the Fibonacci heap and the Fibonacci heap node, in addition to the label-set object, a label object needs to be created to store the required data. The properties of the label object depend on the criteria that will be traced, like distance, battery SoC, energy consumption and etc.

When the label-set and label object is completed, and the label containing cost and energy consumption, then it's able to build the Electric Vehicle Route Planning Algorithm 5.

It should be noted that due to the Pareto set setting, the computational complexity increases exponentially. If the Pareto set is abused, the program will become difficult to run when the amount of data reaches several hundred, let alone tens of thousands or even hundreds of thousands of data.

If you simply follow the pseudo-code provided by Algorithm 2, you will find that the program will fall into a situation similar to an infinite loop. This is due to the uncontrolled addition of labels to the algorithm. In the pseudo-code, as long as the original label cannot domain the new label, the new label will be added to the calculation. And this newly added label will undergo the same calculation as the previous label in subsequent calculations like a shadow. Because obviously under the same path, the label that could not be drained before still cannot be domain after going through the same calculation process.

What's even more frightening is that if the distance between several charging stations on the map is close enough, which is very common in maps of city centers, any independent label will generate tens of thousands or even more shadow labels after repeated cycles of charging at these charging stations. This is not only close to an infinite loop in terms of time, but also quickly fills up the memory of the computer in terms of space.

Another issue to note is that due to the characteristics of Dijkstra's algorithm's label settled, the electric vehicle route planning algorithm limited by battery energy may be preemptively settled by a local optimal solution with insufficient power but shorter time-consuming for some key vertices.

Let's have a map like a Figure 2.2, starting from vertex 1, and the target is vertex 6. The number in the edge is (x,y), which x: length and y: SoC it needs to across. The 1,2,4,5,6 is a feasible path, while 1,3,4,5,6 is not because the battery

Algorithm 5 Electric Vehicle Route Planning Algorithm

Input: Graph $G = (V, V_c, E, f_c, f_e)$
Input: Battery constraints $B_{cons} = [B_{bott\_limit}, B_{up\_limit}]$
Input: Source and Target vertices s, t
Input: Initial SoC $B(s)$
Data: Fibonacci Heap Q
Output: Array labelSet[·] containing label-sets for each vertex in V
for each $v \in V$ **do** ▷ initialization
    labelSet$[v] \leftarrow \infty$
labelSet$[v] \leftarrow \{(0, B(s), \perp)\}$
Q.insert(s, Key(labelSet$[s]$))
while not Q.isEmpty() **do** ▷ main loop
    $u \leftarrow Q$.deleteMin()
    if $u = t$ **then stop**
    $(dt, b,$ parent$) \leftarrow$ SETTLE(labelSet$[u]$)
    if HASUnsettledLabels(labelSet$[u]$) **then** Q.insert(u,KEY(labelSet$[u]$))
    for each $(u, v) \in V$ **do**
        $dt' \leftarrow dt + dt(u, v)$
        $b' \leftarrow \min(b-$ cons$(u, v), M)$
        if $(b' \in B)$ **and not** (labelSet$[v]$ domain (dt', b', u)) **then**
            labelSet$[v] \leftarrow$ labelSet$[v] \cup \{(dt', b', u)\}$
            if Q.contains(v) **then**
                Q.decreaseKey(v, KEY(labelSet$[v]$))
            else
                Q.insert(v, KEY(labelSet$[v]$))

runs out. The algorithm is label setting, which means when a vertex is settled, it will not be visited again. Vertex 4 will be first visited by routes 1,3,4 if we set the distance as the only optimal target. The distance of route 1,3,4 is 4 while route 1,2,4 is 9. And 1,3,4,5 is 5 which is still less than 9, which means vertex 4 will be settled before we compute the routes 1,2,4,5. In this case, 1,2,4,5 will not be computed again and the algorithm will tell you there is no feasible path from 1 to 6. But obviously, 1,2,4,5,6 is a feasible path.

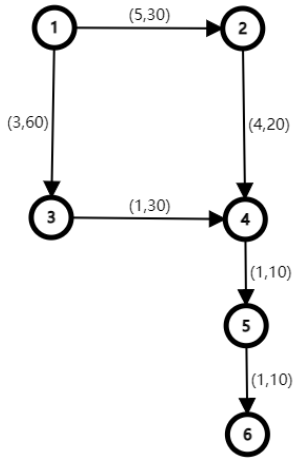

Figure 2.2: Label settled

In order to solve the above two problems, it is necessary to add some additional rules to the electric vehicle routing planning algorithm:

- **Renew settled vertex**

  If the battery energy of the new label exceeds the battery energy of the label with the largest battery energy of the settled vertex, reset the settled vertex to unsettled, and add this label to the vertex. This will solve the problem of Fig 2.2.

- **Prohibit going to the last computed vertex**

  This is achieved directly by the label-settled feature in the traditional Dijkstra algorithm. All calculated nodes will not be visited again, so obviously the previous node will definitely not be visited. But since we may renew the settled label, we must make some restrictions to avoid endless growth of computational complexity.

- **Add bigger SoC only**

  In the pseudo-code, as long as the label cannot be domain will be added to the calculation. As mentioned above, this can cause major problems. Obviously, due to the characteristics of Dijkstra's algorithm, it is impossible for the new label to outperform the original label in a certain linear combination of our objects. Therefore, only those labels with larger SoC can be added to avoid the previous label being eliminated in the subsequent calculation due to insufficient battery energy and discard the current label that is worse but the battery energy can satisfy the path.

Chapter 3

Implementation EV route planning algorithm

In this chapter, we first introduce the vehicle model for the implementation of the algorithm. By using this electric vehicle model, we can calculate energy consumption, and track the state of charge of the battery. Obviously, using a Simulink 0-degree-of-freedom power model including batteries and motors as a vehicle model will have higher accuracy, but the simulation time may be unacceptable. In the second section, the map of northwestern Italy used is presented. The .XML file is a common data storage format, and both SUMO and Open Street Map use this format. In that part will accept how to extract useful map data from the .XML file, the same method is not only applicable to SUMO-edited maps but also available in the open source Open Street Map. The third section introduces the specific Matlab code implementation of the electric vehicle routing planning algorithm.

3.1 Vehicle model

In this thesis, a BMW i3 2017 as the vehicle model is used in the implementation of the algorithm, the detail of the vehicle's mechanical data can be found in Table 3.1 and Table 3.2 shows the electric details.

Because it is hoped that the average speed of the vehicle on different roads can be specified in advance, and the whole calculation process is simple enough, V_{max} and $V_{traffic}$ are defined in this thesis, which:

- V_{max}: The maximum speed allowed on the road is limited by the speed limit of traffic laws, the driving style of the driver, and the maximum speed of the vehicle.

Vehicle weight	1320 kg
Vehicle Length	3999 mm
Vehicle Width	1775 mm
Drag coefficient C_d	0.3
Frontal area S_f	2.38 mm^2
Tire size	245/45R20
f_0	0.0037
f_2	1.91×10^{-7}
Transmission ratio τ	4
Transmission efficiency η	0.99

Table 3.1: Vehicle model (mechanical)

Battery Voltage	352V
Motor Voltage	400V
Battery Energy	33200Wh
Battery Capacity	94.3Ah
Battery Maximum Charging Power	7400W
Battery Series	96
Battery parallel	1

Table 3.2: Vehicle model (electric)

- $V_{traffic}$: Average speed of road travel due to current traffic conditions. Obviously $V_{traffic} \leq V_{max}$

In this thesis, a random normal distribution with $V_{traffic}$ as the mean is used to simulate the speed trajectory of a vehicle on a traffic road. In the velocity trace, all velocities above V_{max} are redefined as V_{max}, and likewise, velocities below zero are redefined as zero. At the same time, the maximum permissible acceleration a_{max} is given by the driver's driving style. For the part of the velocity trajectory that exceeds the maximum acceleration a_{max}, adjust it to the maximum acceleration and recalculate until it satisfies the condition below at the same time:

- $V \leq V_{max}$
- $a \leq a_{max}$
- $\text{mean}(V) = V_{traffic}$

The influence of temperature on energy consumption is only used as a linearly changing factor from the SAE study on electric vehicle range [12], which is directly multiplied by the calculated battery power.

For a traffic speed $V_{traffic}$ of 28 m/s and a temperature of 24 °C, when driving 10,000 meters, the changes in energy consumption and driving time with the maximum speed V_{max} are shown in Figure 3.1: Since the traffic speed $V_{traffic}$ is set

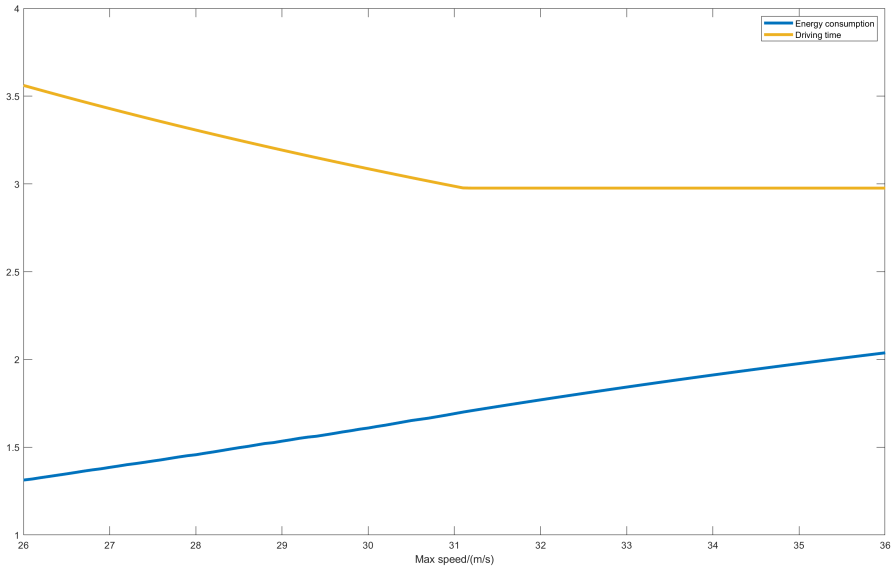

Figure 3.1: Energy consumption and driving time - V_{max}

as the average passing speed, obviously when the maximum speed V_{max} is higher than $V_{traffic}$, the driving time will not change with the change of V_{max}. When V_{max} is lower than $V_{traffic}$, $V_{traffic}$ is adjusted to 0.9 times V_{max}. In this case, the driving time varies linearly with V_{max}. For energy consumption, it is basically linear with V_{max}.

The change in energy consumption and driving time with traffic speed is shown in Figure 3.2, and the fixed maximum speed is 36m/s at this time. When the traffic speed is less than the maximum speed, both energy consumption and driving time have an approximately linear relationship with the traffic speed. When the traffic speed is higher than the maximum speed, the maximum speed limit is imposed. So neither energy consumption nor driving time will change anymore.

As mentioned before, the temperature is simply multiplied directly by the battery power as an influencing factor. So the curve as a function of temperature will be: energy consumption varies linearly and driving time is independent of temperature. As shown in Figure 3.3:

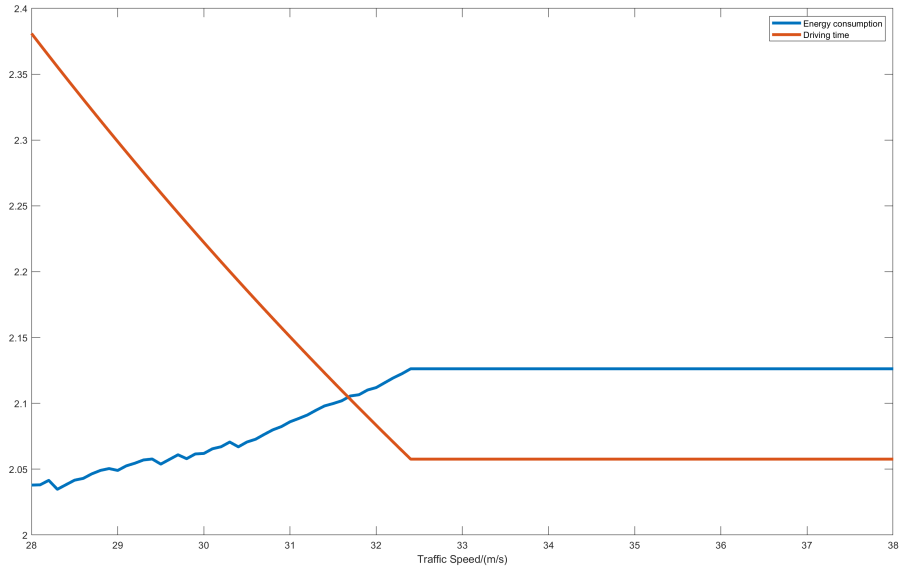

Figure 3.2: Energy consumption and driving time - $V_{traffic}$

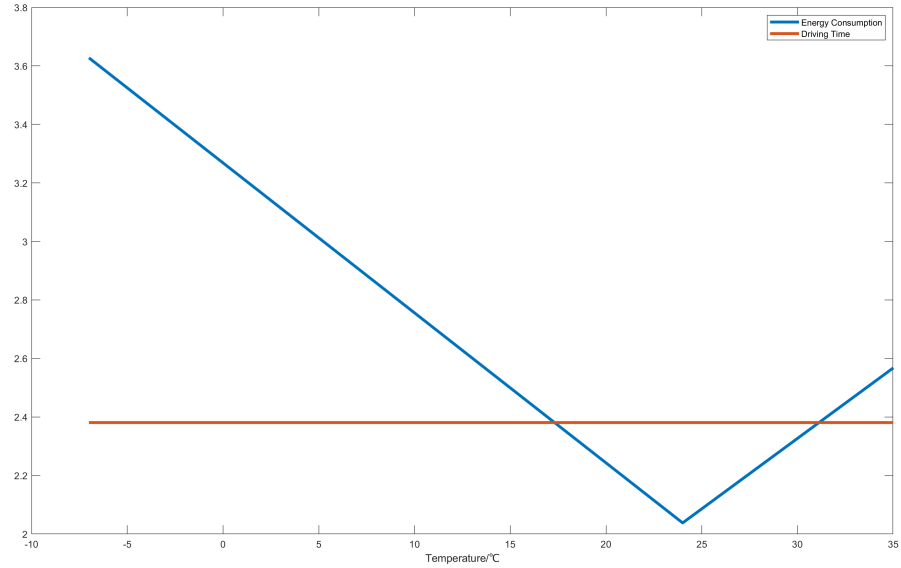

Figure 3.3: Energy consumption and driving time - Temperature

3.2 Map data

In this thesis, a map of northwest Italy edited with SUMO is used. Contains more than 50,000 vertices (or in the word in SUMO 'Junctions'), more than 90,000 edges, and 675 charging stations. There are four basic attributes on the edges of the map:

- **Edge From** The start vertex to which the edge connects
- **Edge To** The end vertex to which the edge connects
- **Distance** The length of the edge
- **Maximum speed** The maximum speed allowed on this edge

At the same time, SUMO also provides data such as latitude and longitude, the number of lanes, the shape of lanes, and the type of lanes. This is convenient for studying traffic conditions, affected by lane capacity; travel costs, affected by road types, and adding some acceleration algorithms later.

The code below provides an example of a string of SUMO map data. For basic research, we need to extract the 'id', 'from', and 'to' of the 'edge' line. The 'speed' and 'length' of the 'lane' line.

```
1  <edge id="-136843408" from="282087873" to="282087871" name="
   Strada Statale 9 Tangenziale di Fiorenzuola" priority="13" type="
   highway.trunk" shape="206198.12,71746.04 206176.65,71797.15
   206158.31,71849.53">
2      <lane id="-136843408_0" index="0" disallow="pedestrian
   bicycle tram rail_urban rail rail_electric rail_fast ship" speed="
   25.00" length="99.59" shape="206199.03,71748.02 206178.14,71797.72
    206163.05,71840.83">
3          <param key="origId" value="136843408 136843407"/>
4      </lane>
5      <param key="ref" value="SS9"/>
6  </edge>
```

This map contains the main traffic network information, but it is missing for some branch roads or rural roads, such as the routing planning algorithm for electric vehicles and its important charging station information. In order to supplement the information on the charging station, it can be added manually in SUMO. Figure **??** is an example of adding a charging station.

3.3 Matlab code implementation

In the current research, many shortest-path algorithms are implemented by using C or Java. It is rarely implemented with Matlab code. However, due to the huge

and excellent simulation toolbox of Matlab, we can easily use the shortest path algorithm implemented on Matlab to jointly simulate with the vehicle dynamics model, vehicle battery model, etc. So it is necessary to implement the shortest path algorithm on Matlab.

SUMO's map data is saved in a .XML file. In order for Matlab to read the map data correctly, we first need to write a small Program [A] to extract useful information from this .XML file. This program saves a .mat file with Matlab struct data which includes:

- **id**: ID of the edge, in the type of string;
- **from**: The start vertex to which the edge connects, in the type of string;
- **to**: The end vertex to which the edge connects, in the type of string;
- **speed**: The maximum speed allowed on this edge, in the type of double;
- **length**: The length of the edge, in the type of double.

After running the program [A] of extract, we can start to find the shortest path by using Dijkstra's algorithm given by Program [B]. The function Dijkstra_sp_000.m will take three inputs **Graph**, **source** and **target** and return three arrays, **dist**[·], **prev**[·] and **S**[·]. The dist[target] will store the distance from the source s to the target t. The prev[·] which is used to generate the path S, is normally not very useful for a point-to-point problem. Path S stores the path from the source to the target.

This program only implements the most basic Dijkstra's algorithm, and its advantage is that it directly accepts strings as input. Both SUMO and OPEN street map use .XML files to store data, so the extracted data must be in the form of strings.

As a test, we can set the source equal to vertex "832544918", and the target equal to vertex "J1". An example of this path can be drawn using SUMO, as shown in Figure 3.4. By running the test with Program [B], we can find the program takes more than 1400 seconds, which is well beyond the allowable range. The reason is that we directly use strings to solve, and simply use arrays to maintain queue Q.

In order to speed up the operation, we can encode edge from and edge to, the simplest is to directly replace the original string ID with sequential decimal numbers as we do in Program [C]. This program will save the map data in the form of forward star representation like Figure 1.3, by using the **edge from index** to behave like a pointer in forward star representation. Also, this program saves all the vertex information, so we can easily transform a vertex string ID to the encoded decimal number or vice versa. The use of binary encoding may further save space and increase speed, but due to the limited time, and for the current amount of

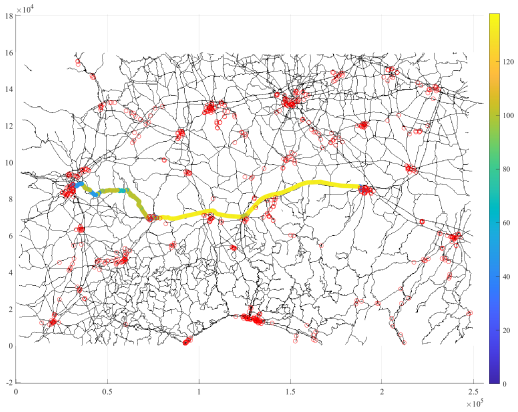

Figure 3.4: Route example

data, decimal encoding is sufficient. Then the algorithm will work like Program [D], which only takes 40 seconds to run the same test, from source '832544918' to target 'J1'. That is 40 times faster than directly operating on strings.

In order to further improve the operation speed, the Fibonacci heap can be used to replace a simple Matlab array. Program [E] and Program [F] respectively construct two objects of the Fibonacci heap node and the Fibonacci heap. Fibonacci heaps are very difficult to construct, and when the amount of data is small, the performance is even worse than binomial heaps due to difficulty in initialization. But as the amount of data increases, the performance of the Fibonacci heap will improve. The Fibonacci heap has two properties:

- **min**: pointer to minimum node
- **n**: the number of nodes currently in heap

The Fibonacci heap also needs to provide some methods, If it is only for Dijkstra's algorithm, it doesn't need to use all the methods, but for the sake of overall understanding and the situation that may be used later, all the methods are provided in Program [F]:

```
1    function H = FibonacciHeap()      %Create the fibonacci heap
2    function ninsert(H,x)             %Insert new node into the heap
3    function insert(H,key,value)      %Make a New node and insert into
    the heap
4    function ele = minimum(H)         %Return the min node
5    function H = union(H1,H2)         %Union two nodes
6    function z = delMin(H)            %Return and delete the min node
    from the heap
```

```
7     function consolidate(H)          %Reorganize Fibonacci heap after
      removing min node
8     function a = D(H)                %Calculate the maximum possible
      degree
9     function Hlink(H,y,x)            %Link nodes inside two Fibonacci
      heaps
10    function key_decreasing(H,x,k)   %Decrease the key of internal
      node x
11    function cut(H,x,y)              %Remove x from the child list of
      y, decrementing y.degree
12    function cascading_cut(H,y)      %Remove all children of y
13    function del_node(H,x)           %delete node
```

The properties required by Fibonacci heap nodes have been mentioned in the first chapter [1], so it won't go into detail here. But to create a Fibonacci heap node, it needs to provide two inputs, a key, and a value. Since Matlab does not provide a default value function, the following code can be used to set the key and value to 0 by default when creating a Fibonacci heap node:

```
1     if(~exist('key','var'))
2         key=0;
3     end
```

Now it's time to update the shortest path algorithm. The program that adds the Fibonacci heap is shown in Program [G]. For the same test, from source '832544918' to target 'J1' which is 2 times faster than the program without Fibonacci heap.

Although the bi-direction search method can be used to speed up the computing speed, since it is impossible to know the total energy consumption after reaching the destination at the beginning, the bi-direction search method cannot be used for the tracking of the electric vehicle battery SoC. Therefore, this method and how to write the corresponding Matlab program will not be studied in this thesis.

We can write label and label set objects to extend Dijkstra's algorithm, so as to realize the limitation of the vehicle battery SoC. The Matlab codes of the label and label set are shown in Program [H] and Program [I]. The corresponding Matlab code of the shortest path algorithm is shown in Program [G]. The Labels object requires 5 properties:

- **distance**: total distance from source s to current vertex v, as one of the criteria to be optimized
- **battery SoC**: Automotive battery SoC at current vertex v, as constraints on feasible paths
- **energy consumption**: total energy consumption from source s to current vertex v, as one of the criteria to be optimized

- **last vertex**: previous vertex of current vertex, used to generate the shortest path
- **travel time**: total travelling time from source s to current vertex v, as one of the criteria to be optimized

The Labels object needs to provide two methods, one is to create a Labels object, and the other is to compare two Labels objects. To create the Labels object method, it needs to provide the default input in the same way as the Fibonacci heap node. Whereas for compare Labels method, an item is considered a better choice only if all criteria are greater than the other. In this way, more Labels can be kept as much as possible in order to select the globally optimal solution.

The Program [G] takes into account the limitations of electric vehicle batteries, but still takes a single shortest distance as the search criteria, and does not include charging stations for vehicles in the map. That means it can only tell how far the vehicle is likely to go within the limits of its current battery power. Obviously, it is possible to approach the destination step by step by searching for the farthest reachable charging station in a segmented manner until the destination is reached. But this obviously cannot provide the overall optimal solution.

In order to simulate the charging process of the vehicle, it is first necessary to add charging stations to the map. in this thesis, we directly add a loop starting and ending at the same vertex on the edge to represent the charging station.

Adding charging stations in this way can be easily simulated on SUMO, but it will cause problems for Dijkstra's algorithm. This is because Dijkstra's algorithm is label settled, which means it does not perform secondary retrievals for labels that have already been retrieved. The entry vertex and exit vertex of the charging station is the same vertexes, which leads to a dead end once the search enters the charging station. To avoid this situation, we can directly define the charging station at its entry and exit vertices in the map data extracted from SUMO. Program [K] marked the location of the charging stations in the map data, while Program [L] defined the charging stations at the entry and exit vertices.

After the charging model is selected, the final routing plan algorithm of the electric vehicle is shown in Program [M]. In this thesis, it is simply assumed that the charging energy per minute is equal. That is, the charging efficiency will not change with the change of battery SoC. The matching object programs include Program [O] and Program [N].

Chapter 4

Simulation results

4.1 Travelling time and energy consumption

In this section, we will test the variation of the optimal routing for electric vehicles under the same traffic conditions, which means that the average traffic speed of the lane is predefined and fixed when the temperature is different. The range of the experiment is from -7°C to 35°C, and the simulation is performed every 1°C. Road traffic always allows the car to travel at the maximum speed allowed by the traffic regulations, regardless of acceleration and deceleration at the starting vertex and the end vertex. The vehicle will charge its battery every time it travels to a charging station, and there is no loss of speed entering and leaving a charging station, and the health of the battery is good, always charging to 100%. The criterion to be optimized is a linear combination of travel time and energy consumption.

Figure 4.1 shows the variation curve of energy consumption with temperature in a single criterion. The blue line only considers the path with the shortest traveling time, while the red line only considers the path with the least energy consumption. When only the minimum energy consumption is considered, the piece-wise linear relationship between energy consumption and temperature is good. Since the energy consumption and temperature also exhibit a piece-wise linear relationship on a single-segment edge like shown in Figure **??**, this shows that when only energy consumption is considered, the path selection and temperature change little. On the curve that only considers travel time, although a piece-wise linear relationship between energy consumption and temperature can also be distinguished, there are some wrinkles. These wrinkles mean that a different path was chosen, often because battery power constraints make faster paths infeasible.

Figure 4.2 shows how energy consumption changes in the ratio of traveling time and energy consumption from 1:0 to 0:1. As expected, energy consumption is highest when travel time is fully considered, and lowest when energy consumption is

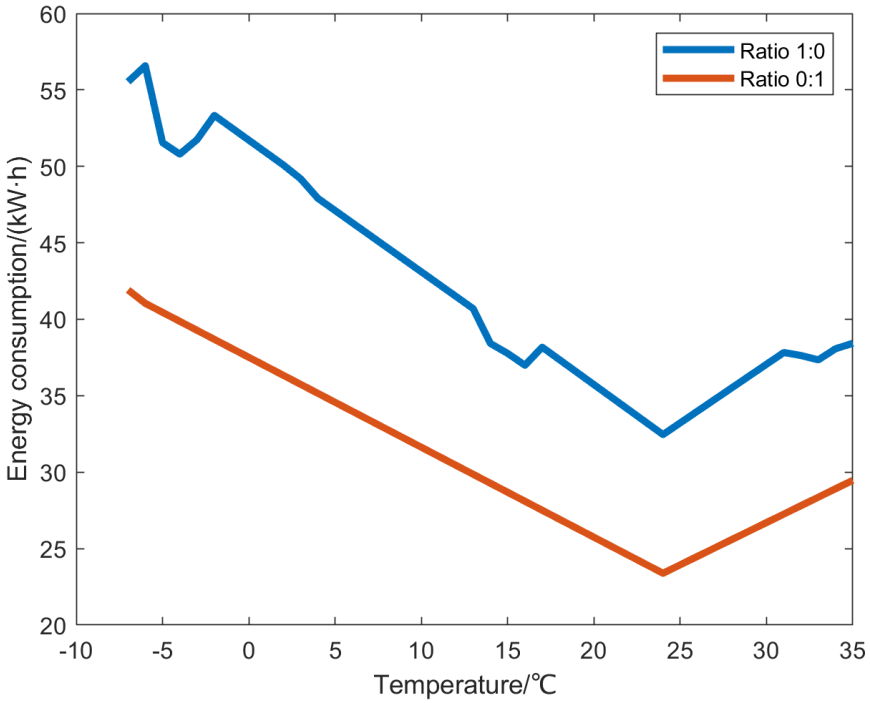

Figure 4.1: Single criterion energy consumption

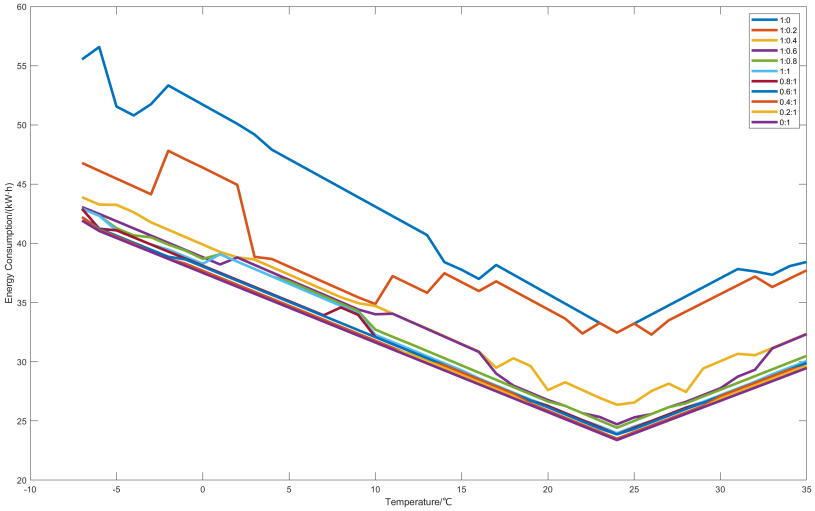

Figure 4.2: Linear combination energy consumption

fully considered. Interestingly, the proportions of different travel times and energy consumption ratios are not consistent at different temperatures. For example, at 5°C, the ratio of 1:0.2 can achieve energy consumption close to the ratio of 0:1, but at 24°C, it completely overlaps with the high energy consumption curve at the ratio of 1:0. Let's examine the details of this figure, which is shown in Figure 4.3. There are two places to notice:

- The closer to the optimum temperature, in this case, 24°C, the less wrinkled the figure will be.

  This may be because, at the optimal temperature, the reduction of energy consumption makes the choice of charging stations for electric vehicles more flexible, which means that electric vehicles can choose a better path regardless of the limitation of battery energy. Clearly, for situations where there is no battery energy limitation, the same optimal path should always be chosen.

- In addition, the reduction of energy consumption and the increase of the proportion of energy consumption criterion are not linear.

  In Figure 4.4, the relationship between energy consumption and energy consumption ratio can be seen more clearly. When the ratio exceeds 0.6:1, the reduction in energy consumption is no longer significant.

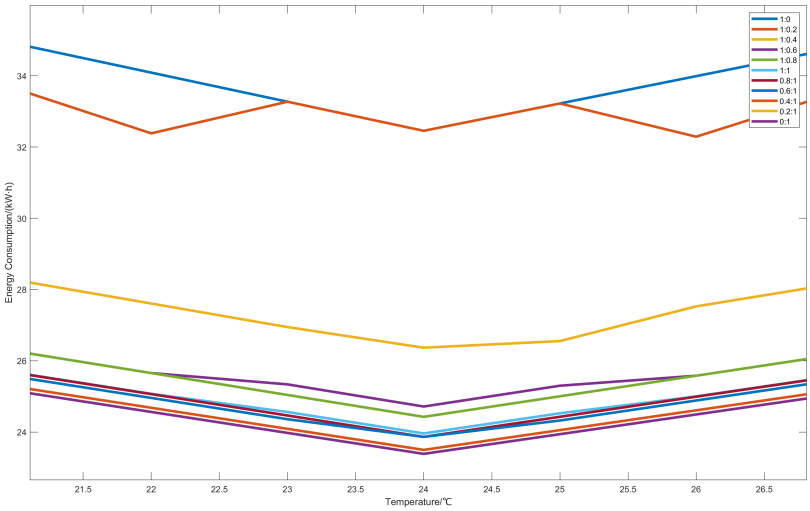

Figure 4.3: Linear combination energy consumption details

As another criterion for linear combinations, the curve of traveling time versus temperature is equally important. Figure 4.5 shows how energy consumption

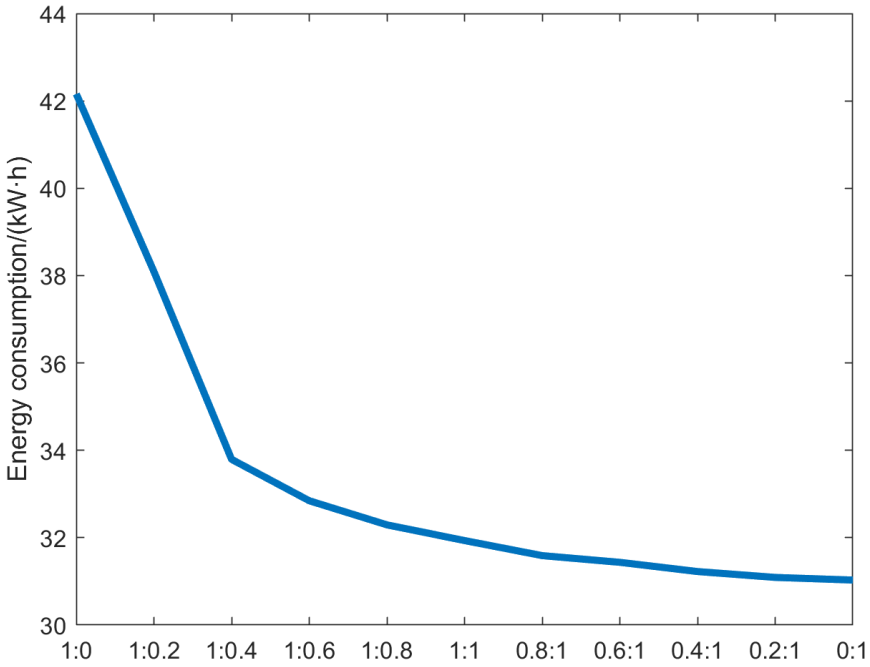

Figure 4.4: Energy consumption

changes in the ratio of traveling time and energy consumption from 1:0 to 0:1. There are many similarities between this figure and Figure 4.1, but by comparing the two figures it can be found that when fully considering energy consumption, just spend less than 0.1kW · h more energy, it can choose a route that saves about 24 minutes of travel time, it takes nearly 24 minutes more to travel. At other temperatures, saving 10kW · h of energy requires about 1 hour of extra travel.

Figure 4.6 shows how traveling time changes in the ratio of traveling time and energy consumption from 1:0 to 0:1. Unlike the energy consumption map, where traveling time contributes the most, traveling time is not necessarily the least at all temperatures. This is because Dijkstra's algorithm is essentially a greedy algorithm, which means that it will always consider the current optimal result in a limited way. This property is good when there is no battery energy constraint, but under battery energy constraints, the globally optimal solution will thus be hidden. But on average, simply considering traveling time does save traveling time. Another conclusion obtained by comparing the two graphs is that under the same energy consumption, the travel time obtained by different travel time-energy consumption ratios is very different. So a reasonable selection ratio is very important. However, in the average value of all temperatures, increasing the proportion of traveling time

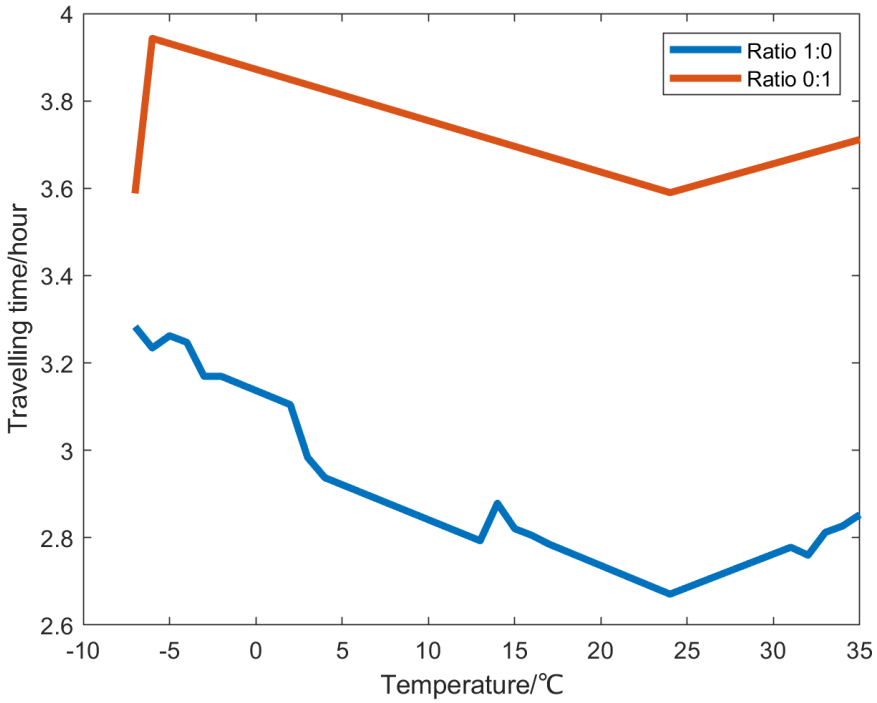

Figure 4.5: Single criterion traveling time

does obtain a smaller traveling time, as shown in Figure 4.7.

Figure 4.8 shows the detail of Figure 4.6 at near 24℃. It can be seen that under different ratios, the interval of traveling time is relatively uniform. Therefore, it may be a good strategy to exclude the proportions whose energy consumption is significantly greater than other groups on the energy consumption figure, then select excellent proportions from the traveling time figure. In addition, due to the characteristics of Dijkstra's algorithm, when the battery energy limitation is considered in the routing selection based on this algorithm, simply increasing the proportion of traveling time does not necessarily result in less traveling time. This is especially important.

Now we look at how the distance varies, although it is not one of the two criteria for the linear combination in the experiment. But the change of distance most directly reflects the change in path selection. Figure 4.6 shows how traveling time changes in the ratio of traveling time and energy consumption from 1:0 to 0:1. It can be seen that when simply considering energy consumption, although it is always necessary to run a long distance, the same path is chosen except at -7°C. And at -7°C, the chosen path is closer than simply considering traveling time. However, at other temperatures, there is a gap of close to 20km. When only traveling time is

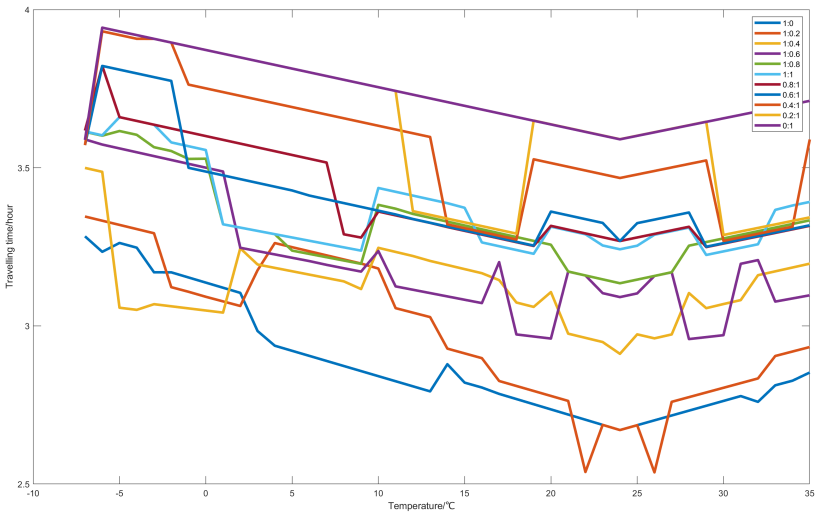

Figure 4.6: Linear combination traveling time

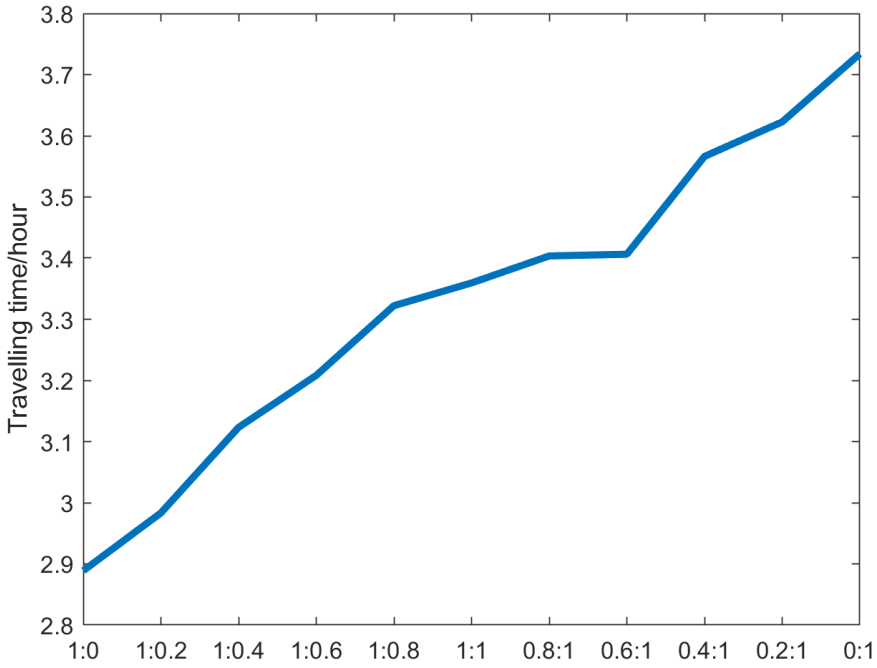

Figure 4.7: Travelling time

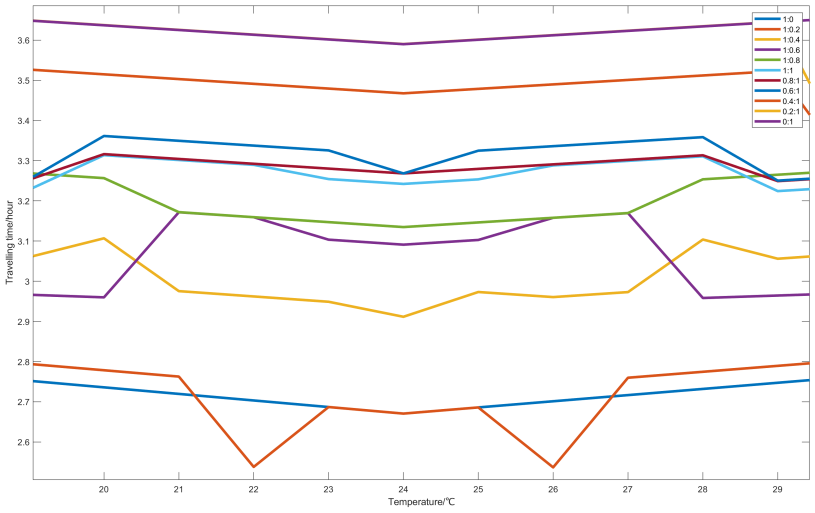

Figure 4.8: Linear combination traveling time details

considered, changes in temperature have a greater impact on path selection.

Figure 4.6 shows how traveling time changes in the ratio of traveling time and energy consumption from 1:0 to 0:1. Since the criteria used are travel time and energy consumption, it is difficult to see patterns in distance versus temperature. Figure 4.11 gives a detailed figure at around 24°C. Even on average, the distance does not absolutely increase with the proportion of energy consumption, but the general trend does increase with the proportion of energy consumption.

Figure 4.13 shows the variation of the minimum value with temperature for each ratio. Because energy consumption is directly proportional to temperature in this thesis, the change of energy consumption is obviously the most obvious with the change of temperature and the proportional relationship is also very clear. In contrast, the distance does not vary substantially with temperature. On the one hand, distance is not one of the two criteria that needs to be optimized, and on the other hand, there are enough and dense enough charging stations in the map used. Of course, in reality, as time goes by, charging stations will be built more densely. Since charging takes time, the trend is the same for travel time and energy consumption. However, since the speed allowed on each road is different, it has a stronger relationship with route selection.

Figure 4.14 shows the change of the minimum value at different ratios. It can be seen that the traveling time and energy consumption have a large change with the ratio, while the distance basically does not change. Except for a sharp rise in the minimum travel time when energy consumption is fully considered, the

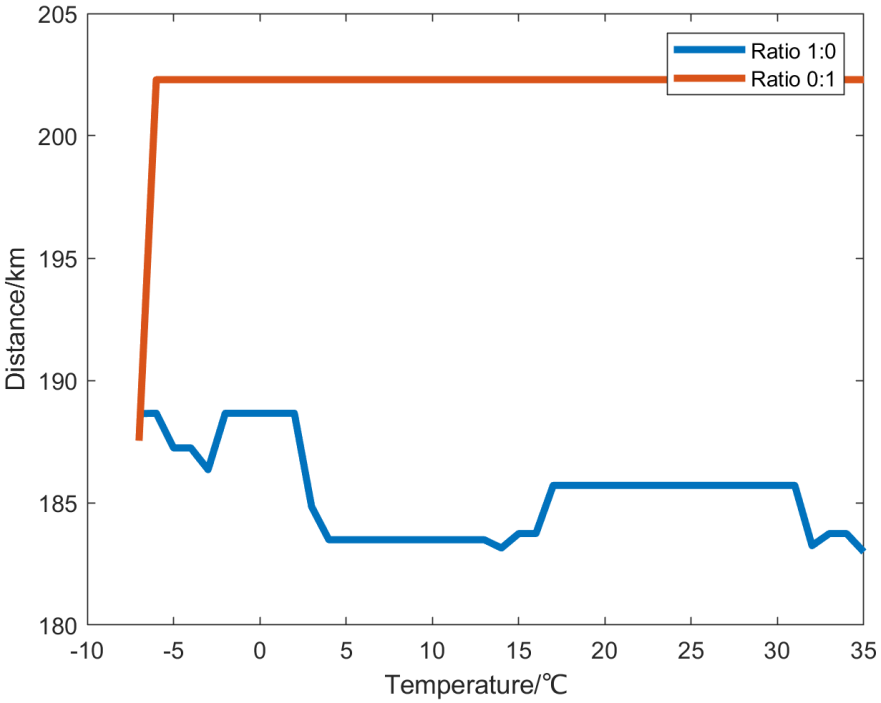

Figure 4.9: Single criterion Distance

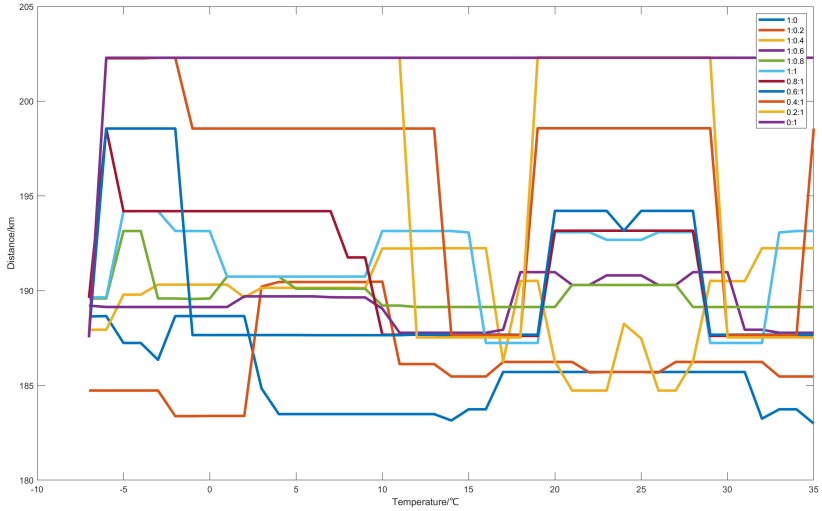

Figure 4.10: Linear combination distance

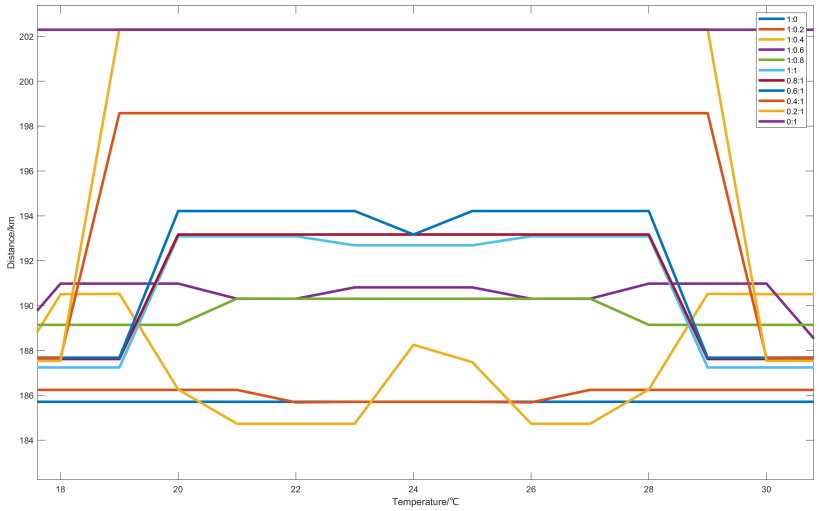

Figure 4.11: Linear combination distance detail

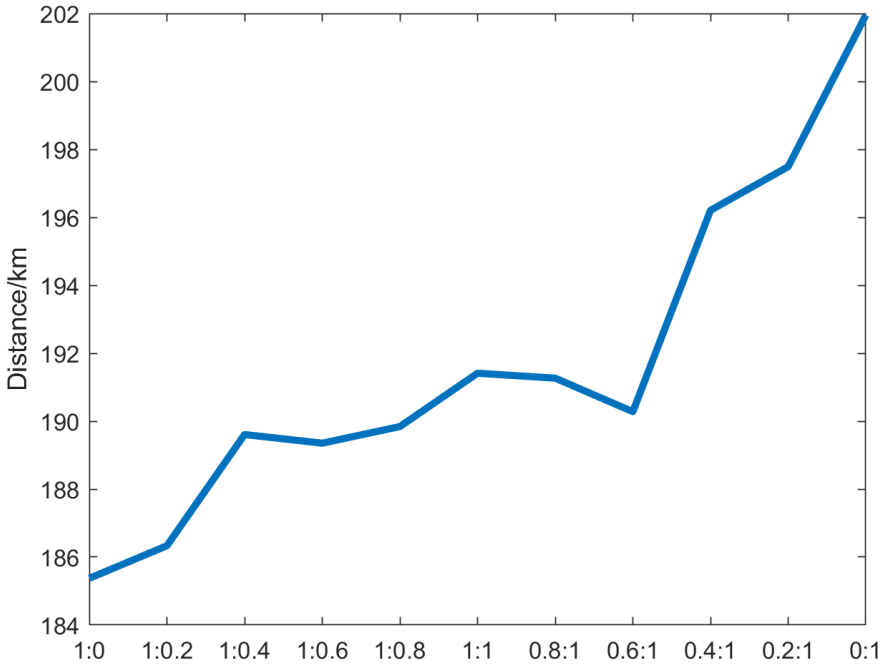

Figure 4.12: Distance

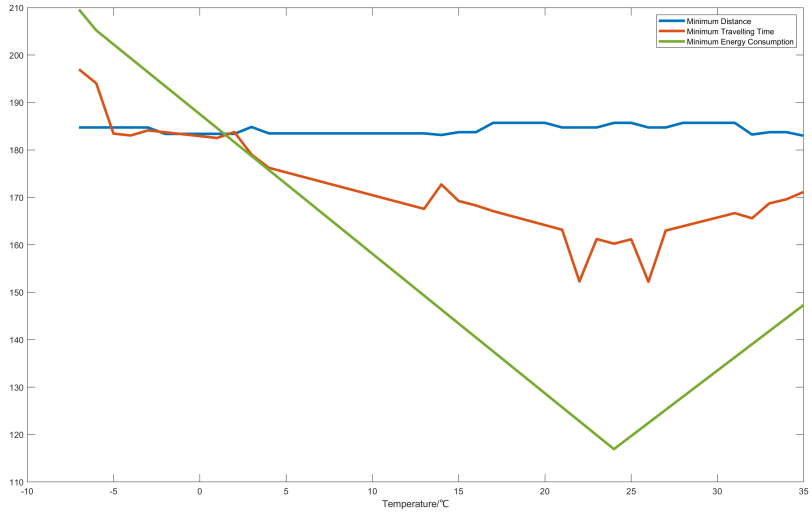

Figure 4.13: Minimum value in different temperature

changes in the other cases are similar. It should be noted that this is the graph of the minimum value, the temperature at which the energy consumption takes the minimum value, and the travel time may not be the minimum value. When using it, you need to refer to the previous graphics for comprehensive consideration.

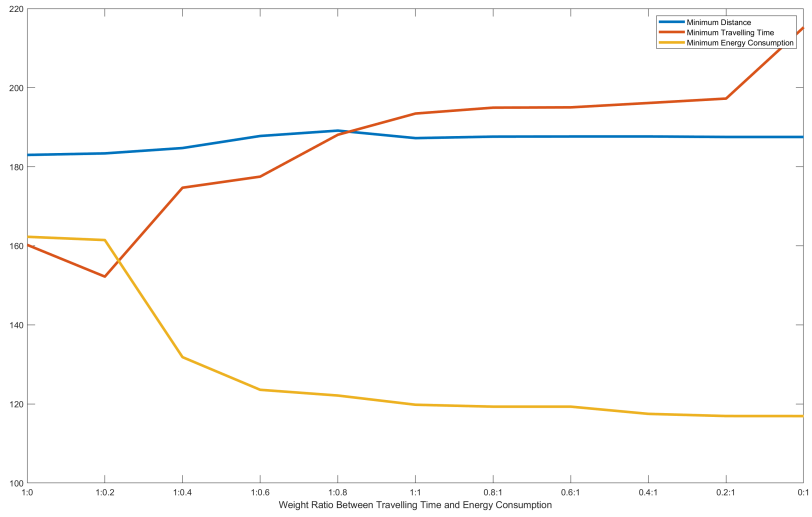

Figure 4.14: Minimum value in different ratios

4.2 Distance and energy consumption

In addition to considering the linear combination of the two criteria of travel time and energy consumption, it is also possible to consider the linear combination of the two criteria of distance and energy consumption. Since charging time and energy consumption are very strongly correlated, and the speed allowed by the road will also affect travel time and energy consumption at the same time. So directly considering the linear combination of travel time and energy consumption may not be a very appropriate choice. As for distance and energy consumption, although there is also a certain relationship, obviously under the same speed, temperature and other conditions, the longer the distance traveled, the higher the energy consumption of electric vehicles. But the correlation will not be as strong as travel time and energy consumption.

Figure 4.15, Figure 4.16 and Figure 4.17 are the energy consumption curves under this linear combination. Unlike the linear combination of travel time-energy consumption, the obtained energy consumption diagram does not appear to cross at all under different linear combination ratios, and even many curves overlap completely.

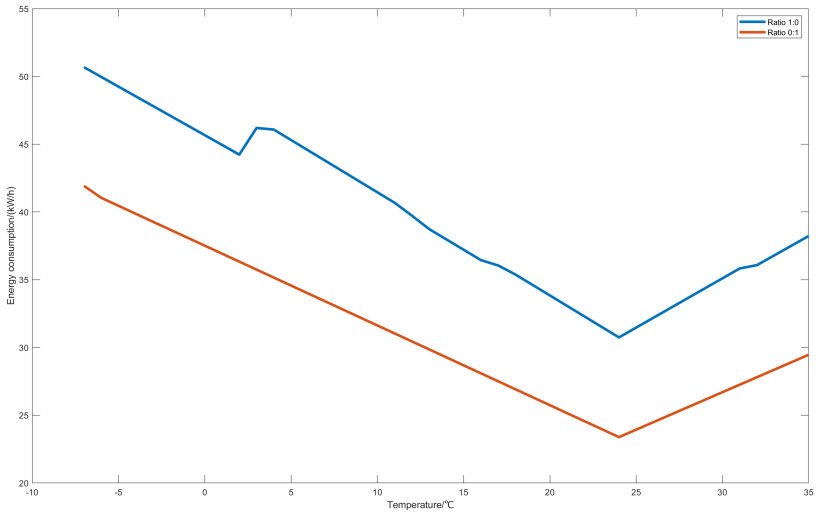

Figure 4.15: Single criterion energy

Figure 4.18, Figure 4.19 and Figure 4.20 are the traveling time curves under this linear combination. The curves of travel time and temperature still have many intersection points, and the curve considering only distance cannot achieve the least travel time at all temperatures.

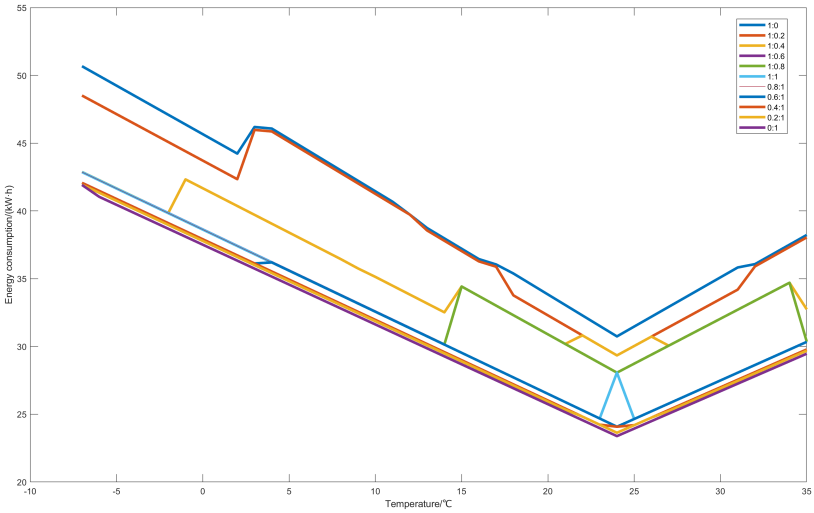

Figure 4.16: Linear combination energy consumption

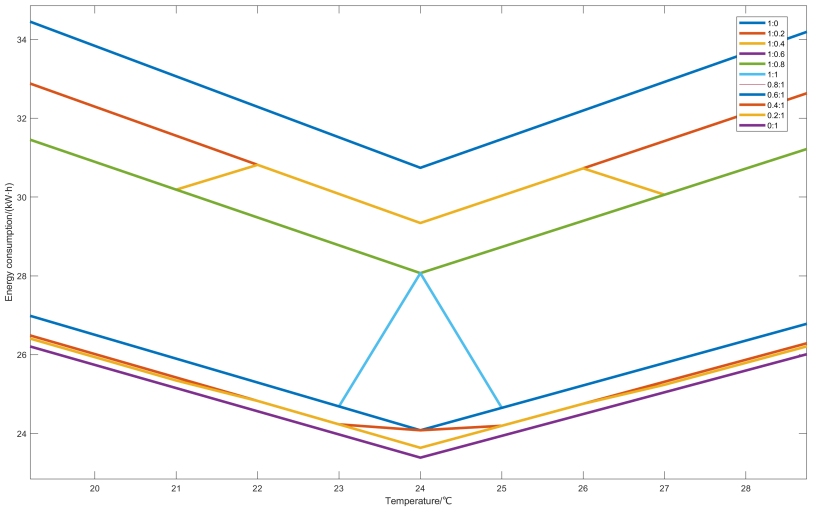

Figure 4.17: Linear combination energy consumption detail

Figure 4.21, Figure 4.22 and Figure 4.23 are the distance curves under this linear combination. It can be seen that the phenomenon of overlapping curves is very obvious, and the difference between other curves is within 10km except for the complete consideration of energy consumption. From here we can find that

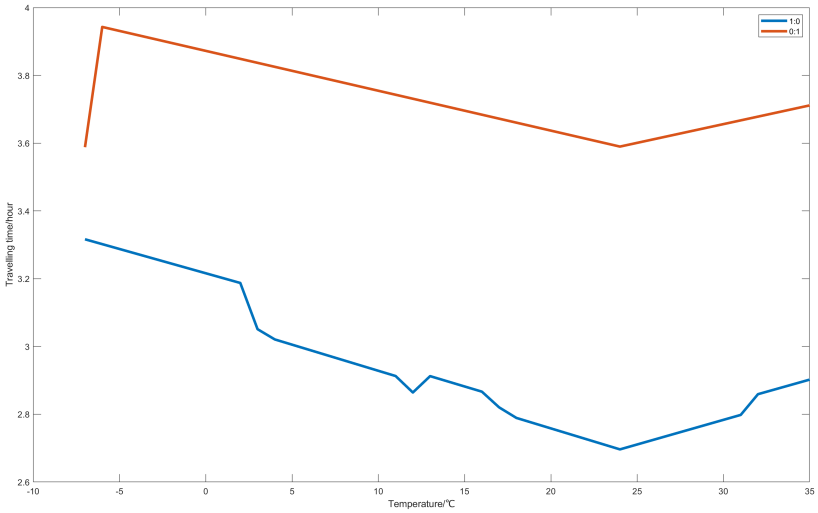

Figure 4.18: Single criterion traveling time

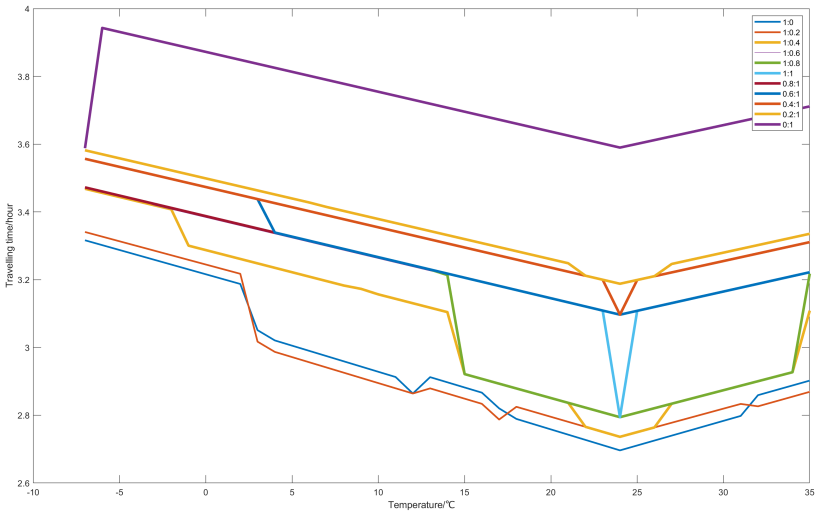

Figure 4.19: Linear combination traveling time

we need to make a trade-off in the selection of the criterion to be optimized. To optimize travel time, it will show a strong correlation with energy consumption, while the effect of optimizing distance is very insignificant.

Figure 4.24 and 4.25 show the change diagram of the minimum value. Compared

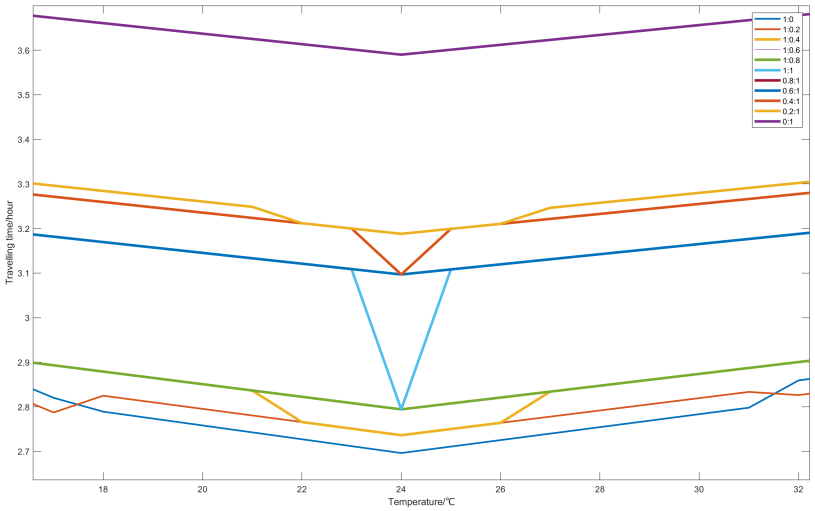

Figure 4.20: Linear combination traveling time detail

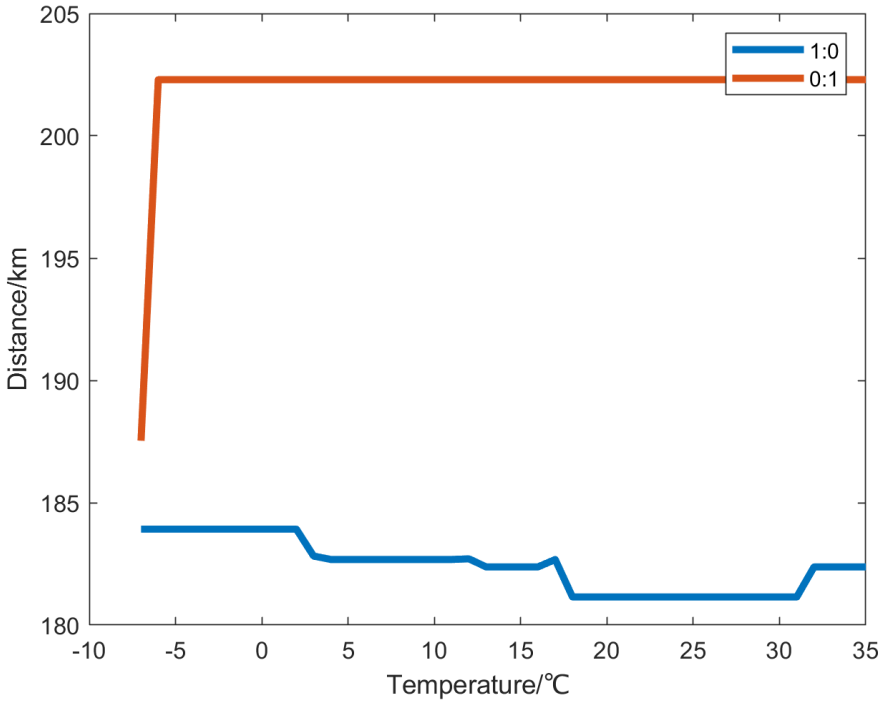

Figure 4.21: Single criterion distance

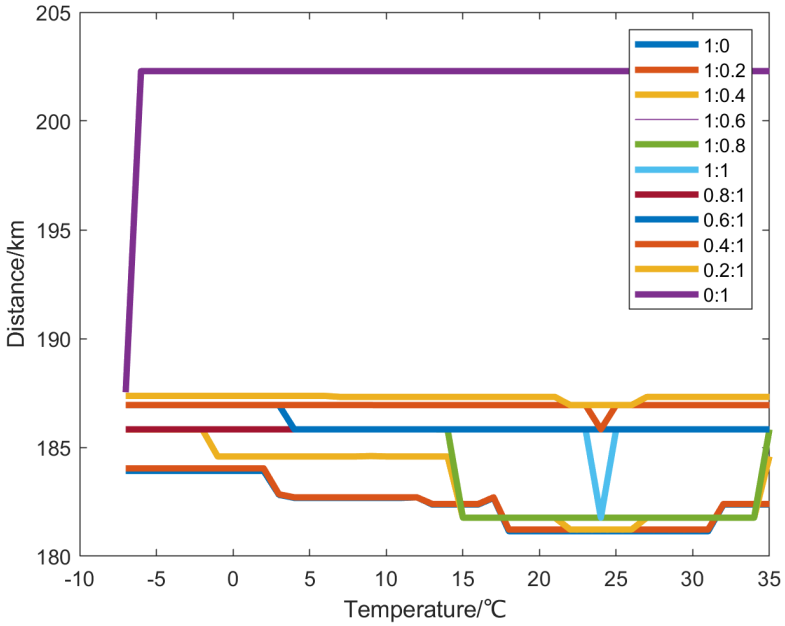

Figure 4.22: Linear combination distance

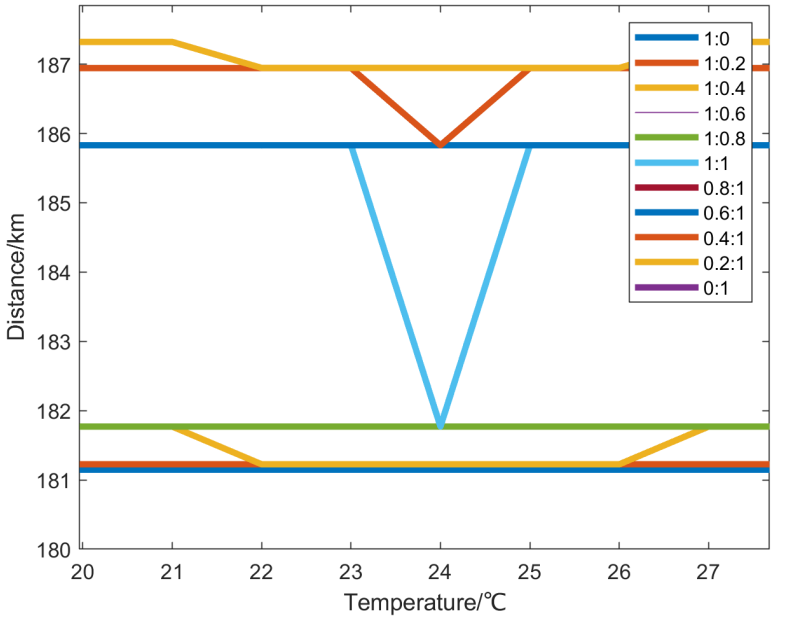

Figure 4.23: Linear combination distance detail

with the linear combination of traveling time and energy consumption, the change of traveling time will be more obvious.

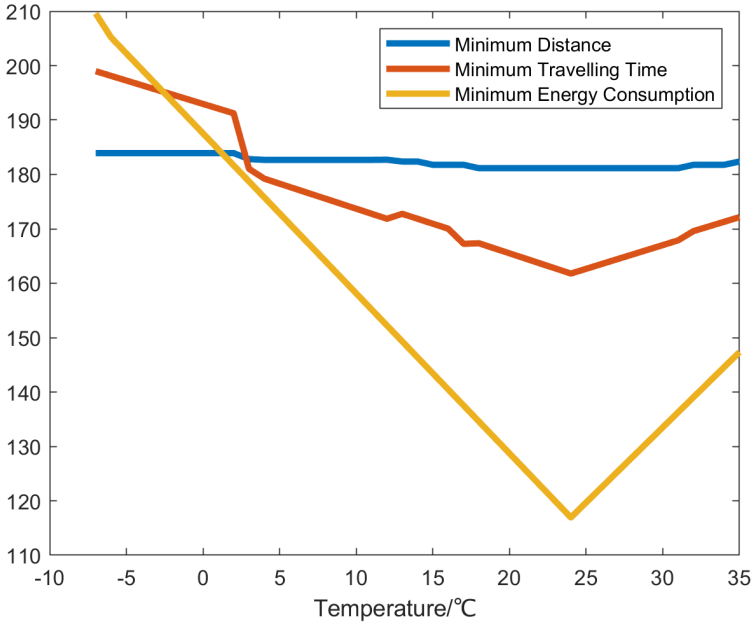

Figure 4.24: Minimum value in different temperature

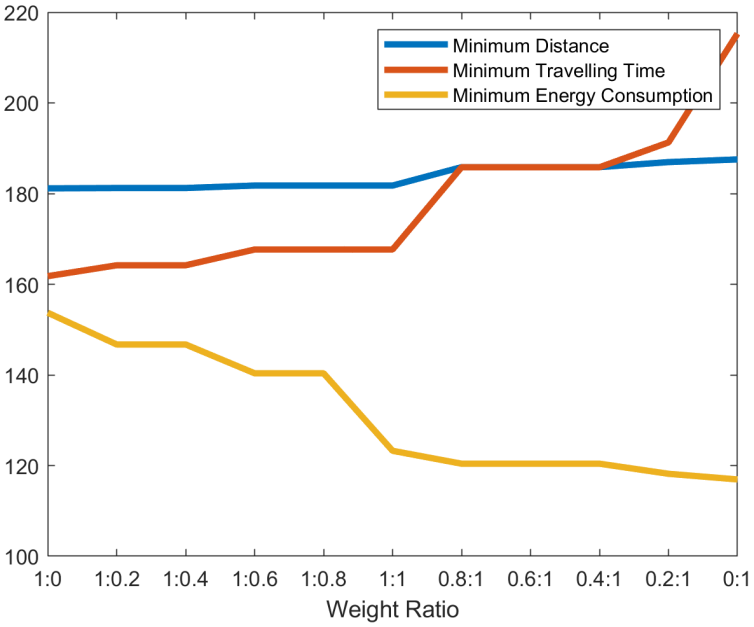

Figure 4.25: Minimum value in different ratios

Chapter 5

Conclusions and future works

5.1 Conclusions

The shortest path algorithm has always been one of the more important research projects in the field of computer algorithms, and many mature navigation software have been developed. At the same time, it is also widely used in unmanned driving, robotic arms, robots, network traffic and other technologies.

With more and more electric vehicles on the market and on the transportation network, people's awareness of environmental protection is getting stronger. How to save energy is an unavoidable topic. This makes it necessary to develop a multi-criteria routing planning algorithm for electric vehicles to provide energy-saving and convenient travel routes.

This thesis develops a multi-criteria routing planning algorithm for electric vehicles on Matlab. Then built the vehicle model, extract useful parts from raw map data, testing and applying the algorithm. Since temperature has a greater impact on the energy consumption of electric vehicles, this thesis also compares the impact on routing at different temperatures in different criteria ratio. It can be seen that at different temperatures, routing is very sensitive to the ratio of linear combinations of multiple criteria. Although changes in temperature and criteria did not simultaneously change the distance of much of the route, only about 10 km varied over a route approaching 200 km. But it has a big impact on energy consumption and travel time. So using the same criteria and the same linear combination ratio of that criteria will not give the same ideal results when the temperature changes. When it is necessary to change the ratio of linear combination, energy consumption should be given priority, because it is nonlinear with the change of ratio, which makes the energy consumption in one or two ratios may be far

greater than that in other cases, while in other ratios, there is little difference in energy consumption. In this way, unsuitable ratios can be easily eliminated. Then consider the traveling time, the change of the average traveling time is almost linear, so the traveling time is reduced while the energy consumption is almost the same.

5.2 Future works

In this thesis, the acceleration technology is not considered. In the follow-up work, the calculation speed can be improved by adding latitude and longitude information and preprocessing the map such as layering. In addition, the vehicle model used is relatively rudimentary. Although the more accurate model has nothing to do with the algorithm itself, it will obviously affect the output results. Finally, more criteria that need to be optimized can be considered, such as the money spent on highways and charging stations.

Appendix A

extract.m

```
clc
clear

edge_data_file = fopen("nordovest.net.xml.txt");
edge_count=0;

while ~feof(edge_data_file)
    tline = fgetl(edge_data_file);
    if contains(tline,"<edge")&&~contains(tline,"internal")
        edge_count=edge_count+1;
    end
end

frewind(edge_data_file)
edge_id = strings([edge_count,1]);
edge_from = strings([edge_count,1]);
edge_to = strings([edge_count,1]);
edge_speed = zeros(edge_count,1);
edge_length = zeros(edge_count,1);

edge_count=0;

while ~feof(edge_data_file)
    tline = fgetl(edge_data_file);
    if contains(tline,"<edge")&&~contains(tline,"internal")
        edge_count=edge_count+1;
        idstart = strfind(tline,'id=')+4;
        idend = strfind(tline,' from=')-2;
        edge_id(edge_count) = string(tline(idstart:idend));
        fromstart = strfind(tline,'from=')+6;
        fromend = strfind(tline,' to=')-2;
```

```
32          edge_from(edge_count) = string(tline(fromstart:fromend));
33          tostart = strfind(tline,'to=')+4;
34          if contains(tline,' name=')
35              toend = strfind(tline,' name=')-2;
36          else
37              toend = strfind(tline,' priority=')-2;
38          end
39          edge_to(edge_count) = string(tline(tostart:toend));
40          tline = fgetl(edge_data_file);
41          speedstart = strfind(tline,'speed=')+7;
42          speedend = strfind(tline,' length=')-2;
43          edge_speed(edge_count) = str2double(tline(speedstart:speedend
    ));
44
45          lengthstart = strfind(tline,'length=')+8;
46          if contains(tline,' acceleration=')
47              lengthend = strfind(tline,' acceleration=')-2;
48          elseif contains(tline,' width=')
49              lengthend = strfind(tline,' width=')-2;
50          elseif contains(tline,' customShape=')
51              lengthend = strfind(tline,' customShape=')-2;
52          else
53              lengthend = strfind(tline,' shape=')-2;
54          end
55          edge_length(edge_count) = str2double(tline(lengthstart:
    lengthend));
56      end
57  end
58
59  fclose(edge_data_file);
60
61
62  edge_struct = struct('id',edge_id,'from',edge_from,'to',edge_to,'
    speed',edge_speed,'length',edge_length);
63
64  save("edge.mat","edge_struct")
```

Appendix B

Dijkstra_sp_000.m

```
function [dist,prev,S]=Dijkstra_sp_000(Graph,source,target)
    Nodes = Graph.Nodes.Name;
    dist = ones(height(Nodes),1)*inf;
    prev = strings(height(Nodes),1);
    Q = Nodes;
    dist(ismember(Nodes,source)) = 0;
    [~,I]=min(dist);
    while ~isempty(Q)
        u = Q(I);
        if u==target
            S=strings();
            if ~isempty(prev(ismember(Nodes,u)))||u==source
                while u~=""
                    S(end+1)=u;
                    u=prev(ismember(Nodes,u));
                end
            end
            S=fliplr(S);
            S(end)=[];
            return
        end
        Q(I)=[];
        length(Q)
        i = find(strcmp(Graph.Edges.EndNodes(:,1),u));
        i_len = length(i);
        set_junc_to=zeros(i_len,1);
        for j=1:i_len
            alt = dist(strcmp(Nodes,u)) + Graph.Edges.Weight(i(j));
            junc_to = find(strcmp(Nodes,Graph.Edges.EndNodes(i(j),2))
    );
            if alt < dist(junc_to)
```

```
31                      dist(junc_to)=alt;
32                      prev(junc_to)=u;
33                  end
34                  set_junc_to(j) = junc_to;
35              end
36              [~,I]=min(dist(ismember(Nodes,Q)));
37          end
38      end
```

Appendix C

reencode000.m

```
clear
clc

load("edge.mat")
node_name = unique([edge_struct.from;edge_struct.to],"stable");
encode_edge_struct = ones(size([edge_struct.from,edge_struct.to,
    edge_struct.length]));

edge_from_index = ones(length(unique(edge_struct.from)),1);

for i = 1:length(edge_struct.id)
    encode_edge_struct(i,1)=find(node_name==edge_struct.from(i));
    encode_edge_struct(i,2)=find(node_name==edge_struct.to(i));
    encode_edge_struct(i,3)=edge_struct.length(i);
end

encode_edge_struct=sortrows(encode_edge_struct,1);

for i = 1:length(edge_from_index)
    x=find(encode_edge_struct(:,1)==i);
    edge_from_index(i)=x(1);
end

save('Node_Name','node_name')
save('encode_edge',"encode_edge_struct","edge_from_index");
```

Appendix D

Dijkstra_sp_001.m

```
function [dist,prev,S]=Dijkstra_sp_001(Graph,Graph_index,source,
    target)
    [source,target] = Node_traslate(source,target);
    Nodes = unique([Graph(:,1);Graph(:,2)]);
    dist = ones(length(Nodes),1)*inf;
    prev = zeros(length(Nodes),1);
    Q = Nodes;
    dist(source) = 0;
    [~,I]=min(dist);
    while ~isempty(Q)
        u = Q(I);
        if u==target
            S=[];
            if prev(u)~=0||u==source
                while u~=0
                    S(end+1)=u;
                    u=prev(u);
                end
            end
            S=fliplr(S);
            S=decode000(S);
            return
        end
        Q(I)=[];
        while u>length(Graph_index)
            [~,I]=min(dist(ismember(Nodes,Q)));
            u = Q(I);
            Q(I)=[];
        end
        i_s = Graph_index(u);
        if u==length(Graph_index)
```

```
31              i_e = i_s;
32          else
33              i_e = Graph_index(u+1)-1;
34          end
35
36          for i=i_s:i_e
37              alt = dist(u) + Graph(i,3);
38              junc_to = Graph(i,2);
39              if alt < dist(junc_to)
40                  dist(junc_to)=alt;
41                  prev(junc_to)=u;
42              end
43          end
44          [~,I]=min(dist(ismember(Nodes,Q)));
45      end
46  end
```

Appendix E

FibonacciHeapNode.m

```
classdef FibonacciHeapNode < handle
    properties
        p       %   pointer to parent
        child   %   pointer to children
        left    %   pointer to left siblings
        right   %   pointer to right sliblings
        degree  %   number of children
        mark    %   indicates whether node x has lost a child since
    the
                %   last time x was made the child of another node
        key     %   key of node
        value   %   value of the key
        nil
    end

    methods
        %Create Node
        function x=FibonacciHeapNode(key,value)
            if(~exist('key','var'))
                key=0;
            end
            if(~exist('value','var'))
                value=0;
            end
            x.key=key;
            x.value=value;
            x.left=x;
            x.right=x;
            x.mark=false;
            x.degree=0;
            x.p=[];
```

```
31                  x.child=[];
32                  x.nil=false;
33              end
34          end
35      end
```

Appendix F

FibonacciHeap.m

```
classdef FibonacciHeap < handle

    properties
        min     %pointer to minimum node
        n       %the number of nodes currently in heap
    end

    methods

        %Create the fibonacci heap
        function H = FibonacciHeap()
            H.n = 0;
            H.min = [];
        end

        %Insert new node into heap
        function ninsert(H,x)
            x.right=x;
            x.left=x;
            if H.n == 0
                H.min = x;
            else
                % insert x to the root
                x.left = H.min.left;
                x.right = H.min;
                H.min.left.right = x;
                H.min.left = x;
                if x.key < H.min.key
                    H.min = x;
                end
            end
```

```
32              H.n = H.n + 1;
33          end
34
35          %Make a New node and insert into heap
36          function insert(H,key,value)
37              if(~exist('value','var'))
38                  value=0;
39              end
40              x=FibonacciHeapNode(key,value);
41              if H.n == 0
42                  H.min = x;
43              else
44                  % insert x to the root
45                  x.left = H.min.left;
46                  x.right = H.min;
47                  H.min.left.right = x;
48                  H.min.left = x;
49                  if x.key < H.min.key
50                      H.min = x;
51                  end
52              end
53              H.n = H.n + 1;
54          end
55
56          function ele = minimum(H)
57              ele=H.min;
58          end
59
60          function H = union(H1,H2)
61              H = FibonacciHeap();
62              H.min = H1.min;
63              H.min.left.right = H2.min.right;
64              H2.min.right.left = H.min.left;
65              H.min.left = H2.min;
66              H2.min.right = H.min;
67              if isempty(H1.min) || ((~isempty(H2.min))&&H2.min.key<H1.
    min.key)
68                  H.min=H2.min;
69              end
70              H.n = H1.n + H2.n;
71          end
72
73          function z = delMin(H)
74              z = H.min;
75              while ~isempty(z.child)
76                  x = z.child;
77                  x.right.left = x.left;
78                  x.left.right = x.right;
79                  if x.left ~= x
```

```
80                      x.p.child = x.left;
81                  else
82                      x.p.child = [];
83                  end
84                  x.left = H.min.left;
85                  x.right = H.min;
86                  H.min.left.right = x;
87                  H.min.left = x;
88                  x.p = [];
89              end
90              z.right.left = z.left;
91              z.left.right = z.right;
92              if z == z.right
93                  H.min = [];
94              else
95                  H.min = z.right;
96                  H.consolidate();
97              end
98              H.n = H.n - 1;
99          end
100
101         function consolidate(H)
102             A=H.D();
103             for i=1:length(A)
104                 A(i).nil = true;
105             end
106             rawNode=H.min;
107             nowNode=rawNode;
108             counter = 0;
109             while true
110                 counter=counter+1;
111                 nowNode=nowNode.right;
112                 if nowNode==rawNode
113                     break
114                 end
115             end
116             for i=1:counter
117                 x=nowNode;
118                 nowNode=nowNode.right;
119                 d=x.degree+1; %d should be the degree, but in matlab,
120                               %index starts with 1, so d is degree+1
    now
121                 while A(d).nil~=true
122                     y=A(d);
123                     if x.key>y.key
124                         a=x;
125                         x=y;
126                         y=a;
127                     end
```

```
128                     H.Hlink(y,x)
129                     A(d)=FibonacciHeapNode;
130                     A(d).nil=true;
131                     d=d+1;
132                 end
133                 A(d)=x;
134             end
135             H.min=FibonacciHeapNode;
136             H.min.nil=true;
137             for i=1:length(A)
138                 if A(i).nil~=true
139                     if H.min.nil==true
140                         A(i).right=A(i);
141                         A(i).left=A(i);
142                         H.min = A(i);
143                     else
144                         %insert A(i) into H's root list
145                         A(i).left=H.min.left;
146                         A(i).right=H.min;
147                         H.min.left.right=A(i);
148                         H.min.left=A(i);
149                         if A(i).key<H.min.key
150                             H.min=A(i);
151                         end
152                     end
153                 end
154             end
155         end
156
157         function a = D(H)
158             a=log(H.n)/log((1+5^0.5)/2);
159             a=floor(a);
160             obj.Array(1,a+1)=FibonacciHeapNode;
161             a=obj.Array;
162         end
163
164         function Hlink(H,y,x)
165             % remove y from the root list of H
166             y.right.left=y.left;
167             y.left.right=y.right;
168             % make y a child of x, incrementing x.degree
169             y.p=x;
170             if isempty(x.child)
171                 x.child = y;
172                 y.left=y;
173                 y.right=y;
174             else
175                 x.child.left.right = y;
176                 y.left = x.child.left;
```

```
177                 x.child.left=y;
178                 y.right=x.child;
179             end
180             x.degree = x.degree+1;
181             % y.mark = false
182             y.mark = false;
183         end
184
185         function key_decreasing(H,x,k)
186             if k>x.key
187                 print('error, new key is greater than current key')
188                 return
189             end
190             x.key=k;
191             y=x.p;
192             if ~isempty(y) && x.key<y.key
193                 H.cut(x,y)
194                 H.cascading_cut(y)
195             end
196             if x.key<H.min.key
197                 H.min=x;
198             end
199         end
200
201         function cut(H,x,y)
202             %remove x from the child list of y, decrementing y.degree
203             if y.child==x
204                 if x.right==x
205                     y.child=[];
206                 else
207                     y.child=x.right;
208                     x.right.left=x.left;
209                     x.left.right=x.right;
210                 end
211             else
212                 x.right.left=x.left;
213                 x.left.right=x.right;
214             end
215             %add x to the root list of H
216             H.min.left.right=x;
217             x.left=H.min.left;
218             H.min.left=x;
219             x.right=H.min;
220             %x.p=NIL
221             x.p=[];
222             %x.mark=false
223             x.mark=false;
224         end
225
```

```
226         function cascading_cut(H,y)
227             z=y.p;
228             if ~isempty(z)
229                 if y.mark == false
230                     y.mark = true;
231                 else
232                     H.cut(y,z)
233                     H.cascading_cut(z)
234                 end
235             end
236         end
237 
238         function del_node(H,x)
239             H.key_decreasing(x,-inf);
240             H.delMin();
241         end
242 
243     end %end of the methods
244 
245 end %end of the classdef
```

Appendix G

Dijkstra_sp_002.m

```
function [dist,prev,S]=Dijkstra_sp_002(Nodes,edges,Graph_index,source
    ,target)
    [source,target]=Node_traslate(source,target);
    obj.Array(length(Nodes),1) = FibonacciHeapNode;
    Fnodes = obj.Array;
    for i=1:length(Nodes)
        Fnodes(i).key = inf;
        Fnodes(i).value = i;
    end
    dist = ones(length(Nodes),1)*inf;
    prev = zeros(length(Nodes),1);
    dist(source)=0;
    Fnodes(source).key=0;
    Q=FibonacciHeap;
    Q.ninsert(Fnodes(source));

    while ~isempty(Q.min)
        u=Q.delMin();
        if u==Fnodes(target)
            S=[];
            while u~=Fnodes(source)
                S(end+1)=u.value;
                u=Fnodes(prev(u.value));
            end
            S(end+1)=u.value;
            S=fliplr(S);
            S=decode000(S);
            break
        end
        while u.value>length(Graph_index)
            u=Q.delMin();
```

```
31          end
32
33          i_s = Graph_index(u.value);
34          if u.value==length(Graph_index)
35              i_e = i_s;
36          else
37              i_e = Graph_index(u.value+1)-1;
38          end
39          for i=i_s:i_e
40              alt = u.key + edges(i,3);
41              junc_to = edges(i,2);
42              if alt < dist(junc_to)
43                  dist(junc_to)=alt;
44                  prev(junc_to)=u.value;
45                  if Fnodes(junc_to).key==inf
46                      Fnodes(junc_to).key=alt;
47                      Q.ninsert(Fnodes(junc_to));
48                  else
49                      Q.key_decreasing(Fnodes(junc_to),alt);
50                  end
51              end
52          end
53      end
54
55  end
```

Appendix H

Labels.m

```
classdef Labels < handle
    properties
        distance
        battery_SoC
        last_vertex
    end
    methods
        function x=Labels(distance,battery_SoC,last_vertex)
            if(~exist('distance','var'))
                distance=inf;
            end
            if(~exist('battery_SoC','var'))
                battery_SoC=0;
            end
            if(~exist('last_vertex','var'))
                last_vertex=0;
            end
            x.distance=distance;
            x.battery_SoC=battery_SoC;
            x.last_vertex=last_vertex;
        end
        function bo = label_le(x,other_label)
            if x.distance<=other_label.distance&&...
               x.battery_SoC>=other_label.battery_SoC
                bo=true;
            else
                bo=false;
            end
        end
    end
end
```

Appendix I

LabelSet.m

```
classdef LabelSet < handle

    properties
        sets
        distances
        battery_SoC
        travel_time
        n
    end

    methods

        function x=LabelSet()
            x.sets=Labels.empty;
            x.distances=[];
            x.battery_SoC=[];
            x.travel_time=[];
            x.n=0;
            x.add_label(Labels(inf,0,0))
        end

        function add_label(x,label)
            x.n=x.n+1;
            x.sets(x.n)=label;
            x.distances(x.n)=label.distance;
            x.battery_SoC(x.n)=label.battery_SoC;
            x.travel_time(x.n)=label.travel_time;
        end

        function key=key_label(x)
            key=min(x.travel_time);
```

```
32          end
33
34          function y=min_label(x)
35              [~,idx] = min(x.travel_time);
36              idx=idx(1);
37              y=x.sets(idx);
38          end
39
40          function y=settle_label(x)
41              [~,idx] = min(x.travel_time);
42              idx=idx(1);
43              y=x.sets(idx);
44              x.sets(idx)=[];
45              x.distances(idx)=[];
46              x.battery_SoC(idx)=[];
47              x.travel_time(idx)=[];
48              x.n=x.n-1;
49          end
50
51          function check_label(x)
52              while 1
53                  [~,idx_dis]=max(x.distances);
54                  [~,idx_ene]=min(x.battery_SoC);
55                  if (idx_dis==idx_ene)&&(x.n~=1)
56                      x.sets(idx_dis)=[];
57                      x.distances(idx_dis)=[];
58                      x.battery_SoC(idx_dis)=[];
59                      x.travel_time(idx_dis)=[];
60                      x.n=x.n-1;
61                  else
62                      break
63                  end
64              end
65          end
66
67          function t=has_unsettled_labels(x)
68              if x.n==0
69                  t=false;
70              else
71                  t=true;
72              end
73          end
74      end
75  end
```

Appendix J

Dijkstra_sp_003.m

```
function label_set_settled=Dijkstra_sp_003(Nodes,Charge,B_range,
    B_source, ...
    edges,Graph_index,source,target)
    %*********************************************************
    %****Calculate the shortest path with battery constrains****
    %*********************************************************


    %%
    %*********************************************************
    %**First step:
    %****Trans the source and target vertex from str to num*****
    %*********************************************************
    [source,target]=Node_traslate(source,target);

    %%
    %*********************************************************
    %**Second step:
    %***********Initail labelset for every vertex*************
    %*********************************************************
    label_set.array(1,length(Nodes))=LabelSet;
    label_set_settled.array(1,length(Nodes))=LabelSet;
    label_set=label_set.array;
    label_set_settled=label_set_settled.array;

    %%
    %*********************************************************
    %**Third step:
    %*******Set the labelset of source vertex to (0,0,0)*******
    %**Which means from source vertex to source vertex:
    %**Have 0 meter distance, 0 Kw/h energy consumption*********
```

```
31      %**And the previce vertex to source is nothing*************
32      %***********************************************************
33      label_set(source).add_label(Labels(0,B_source,0));
34      label_set(source).check_label();
35
36      %%
37      %***********************************************************
38      %**Fourth step:
39      %*****Initail the Fibonacci Heap Node for every vertex******
40      %***********************************************************
41      Fnodes.Array(length(Nodes),1) = FibonacciHeapNode;
42      Fnodes = Fnodes.Array;
43      for i=1:length(Nodes)
44          Fnodes(i).key = inf;
45          Fnodes(i).value = i;
46      end
47
48      %%
49      %***********************************************************
50      %**Fifth step:
51      %*****And the value from source vertex label set to
52      % Fibonacci Heap*****************************************
53      %***********************************************************
54      Q=FibonacciHeap;
55      Fnodes(source).key=label_set(source).key_label();
56      Q.ninsert(Fnodes(source));
57
58      %%
59      %***********************************************************
60      %**Sixth step:
61      %************************Main Loop**************************
62      %***********************************************************
63      while ~isempty(Q.min)
64          %%
65          %*******************************************************
66          %**Seventh step:
67          %***From Fibonacci Heap get the present search vertex***
68          %*******************************************************
69          u=Q.delMin();
70          while u.value>length(Graph_index)
71              u=Q.delMin();
72          end
73
74          %%
75          %*******************************************************
76          %**Extra step
77          %************************Charge*************************
78          %*******************************************************
79          if find(Charge==u.value)
```

```
80              for i=1:length(label_set(u.value))
81                  label_set(u.value).battery_SoC(i)=B_range(2);
82                  label_set(u.value).sets(i).battery_SoC=B_range(2);
83              end
84          end
85
86          %%
87          %*******************************************************
88          %**Eighth step:
89          %**When present search vertex equal to target vertex,
90          % stop*****************************************************
91          %*******************************************************
92  %           if u==Fnodes(target)
93  %               the_label = label_set(u.value).settle_label();
94  %               label_set_settled(u.value).add_label(the_label);
95  %               label_set_settled(u.value).check_label();
96  %               break
97  %           end
98
99          %%
100         %*******************************************************
101         %**Nineth step:
102         %**************Settle the current label***************
103         %*******************************************************
104         the_label = label_set(u.value).settle_label();
105         label_set_settled(u.value).add_label(the_label);
106         label_set_settled(u.value).check_label();
107         if label_set(u.value).has_unsettled_labels()
108             Fnodes(u.value)=FibonacciHeapNode();
109             Fnodes(u.value).value=u.value;
110             Fnodes(u.value).key=label_set(u.value).key_label();
111             Q.ninsert(Fnodes(u.value));
112         end
113
114         %%
115         %*******************************************************
116         %**Tenth step:
117         %********Check all edges connected to vertex u**********
118         %*******************************************************
119
120         i_s = Graph_index(u.value);
121         if u.value==length(Graph_index)
122             i_e = i_s;
123         else
124             i_e = Graph_index(u.value+1)-1;
125         end
126         for i=i_s:i_e
127             alt = the_label.distance + edges(i,3);
128             junc_to = edges(i,2);
```

```
129             battery_SoC = the_label.battery_SoC-edges(i,4);
130             if battery_SoC>B_range(1)&&label_set(junc_to).
    has_unsettled_labels()
131                 new_label = Labels(alt,battery_SoC,u.value);
132                 mini_label = label_set(junc_to).min_label;
133                 if ~mini_label.label_le(new_label)
134                     label_set(junc_to).add_label(new_label);
135                     label_set(junc_to).check_label();
136                     if Fnodes(junc_to).key==inf
137                         Fnodes(junc_to).key=alt;
138                         Q.ninsert(Fnodes(junc_to));
139                     elseif Fnodes(junc_to).key>alt
140                         Q.key_decreasing(Fnodes(junc_to),alt);
141                     end
142                 end
143             end
144         end
145     end
```

Appendix K

reencode002.m

```
clear
clc

load('CSFile2.mat')
load("edge.mat")

node_name = unique([edge_struct.from;edge_struct.to],"stable");
edge_size = size([edge_struct.from,edge_struct.to,edge_struct.length
    ]);
edge_size(2)=edge_size(2)+1;
encode_edge_struct = zeros(edge_size);

Charge=ones(length(chargingStation),2);
for i = 1:length(Charge)
    b=chargingStation(i);
    Charge(i,1)=find(node_name==edge_struct.from(edge_struct.id==b.
    edge));
    Charge(i,2)=find(node_name==edge_struct.to(edge_struct.id==b.edge
    ));
    if encode_edge_struct(edge_struct.id==b.edge,4)==0
        encode_edge_struct(edge_struct.id==b.edge,4)=1;
    else
        b.edge
    end
end

edge_from_index = ones(length(unique(edge_struct.from)),1);
edge_to_index = ones(length(unique(edge_struct.to)),1);

for i = 1:length(edge_struct.id)
    encode_edge_struct(i,1)=find(node_name==edge_struct.from(i));
```

```
29      encode_edge_struct(i,2)=find(node_name==edge_struct.to(i));
30      encode_edge_struct(i,3)=edge_struct.length(i);
31  end
32  encode_edge_struct=sortrows(encode_edge_struct,1);
33
34  for i = 1:length(edge_from_index)
35      x=find(encode_edge_struct(:,1)==i);
36      edge_from_index(i)=x(1);
37  end
38
39  for i = 1:length(edge_to_index)
40      x=find(encode_edge_struct(:,2)==i);
41      if isempty(x)
42          continue
43      else
44          edge_to_index(i)=x(1);
45      end
46  end
47
48  save('Node_Name','node_name')
49  save('edge2',"encode_edge_struct","edge_from_index","edge_to_index","
        Charge");
```

Appendix L

findingchargingstation.m

```
clear
clc
%Find charging stations edge
load edge2.mat
newCharge=zeros(length(Charge),1);
for i=1:length(Charge)
    N_from = Charge(i,1);
    i_s = edge_from_index(N_from);
    if N_from==length(edge_from_index)
        i_e=i_s;
    else
        i_e = edge_from_index(N_from+1)-1;
    end
    while i_s==i_e
        N_from = encode_edge_struct(encode_edge_struct(:,2)==N_from
    ,1);
        N_from = N_from(1);
        i_s = edge_from_index(N_from);
        if N_from==length(edge_from_index)
            i_e=i_s;
        else
            i_e = edge_from_index(N_from+1)-1;
        end
%         if i==373
%             break
%         end
    end
    newCharge(i) = N_from;
end
Charge=unique(newCharge);
```

```
30 save('edge3',"encode_edge_struct","edge_from_index","edge_to_index","
       Charge");
```

Appendix M

Dijkstra_sp_007.m

```
function label_set_settled=Dijkstra_sp_007(Nodes,Charge,B_range,
    B_source, ...
    edges,Graph_index,source,target,ratio)
    %*************************************************************
    %****Calculate the shortest path with battery constrains****
    %*************************************************************
    %******And the simulate with temperature and traffic********

    %%
    %*************************************************************
    %**First step:
    %****Trans the source and target vertex from str to num*****
    %*************************************************************
    [source,target]=Node_traslate(source,target);

    %%
    %*************************************************************
    %**Second step:
    %***********Initail labelset for every vertex*************
    %*************************************************************
    label_set.array(1,length(Nodes))=LabelSet_5;
    label_set_settled.array(1,length(Nodes))=LabelSet_5;
    label_set=label_set.array;
    label_set_settled=label_set_settled.array;
    for i=1:length(label_set)
        label_set(i).ratio=ratio;
    end
    for i=1:length(label_set_settled)
        label_set_settled(i).ratio=ratio;
    end

```

```
31      %%
32      %*************************************************************
33      %**Third step:
34      %*******Set the labelset of source vertex to (0,0,0)*******
35      %**Which means from source vertex to source vertex:
36      %**Have 0 meter distance, 0 Kw/h energy consumption*********
37      %**And the previce vertex to source is nothing**************
38      %*************************************************************
39      label_set(source).add_label(Labels_3(0,B_source,0,0,0));
40      label_set(source).check_label();
41
42      %%
43      %*************************************************************
44      %**Fourth step:
45      %******Initail the Fibonacci Heap Node for every vertex******
46      %*************************************************************
47      Fnodes.Array(length(Nodes),1) = FibonacciHeapNode;
48      Fnodes = Fnodes.Array;
49      for i=1:length(Nodes)
50          Fnodes(i).key = inf;
51          Fnodes(i).value = i;
52      end
53
54      %%
55      %*************************************************************
56      %**Fifth step:
57      %******And the value from source vertex label set to
58      % Fibonacci Heap*************************************************
59      %*************************************************************
60      Q=FibonacciHeap;
61      Fnodes(source).key=label_set(source).key_label();
62      Q.ninsert(Fnodes(source));
63
64      %%
65      %*************************************************************
66      %**Sixth step:
67      %************************Main Loop****************************
68      %*************************************************************
69      while ~isempty(Q.min)
70          %%
71          %*********************************************************
72          %**Seventh step:
73          %***From Fibonacci Heap get the present search vertex***
74          %*********************************************************
75          u=Q.delMin();
76          while u.value>length(Graph_index)
77              u=Q.delMin();
78          end
79
```

```
80          %%
81          %*********************************************************
82          %**Extra step
83          %**********************Charge*****************************
84          %*********************************************************
85          if find(Charge==u.value)
86              for i=1:length(label_set(u.value))
87                  label_set(u.value).travel_time(i)=...
88                  label_set(u.value).travel_time(i)+(B_range(2)-
    label_set(u.value).battery_SoC(i))/100e3*3600;
89                  label_set(u.value).sets(i).travel_time=...
90                  label_set(u.value).travel_time(i)+(B_range(2)-
    label_set(u.value).battery_SoC(i))/100e3*3600;
91                  label_set(u.value).battery_SoC(i)=B_range(2);
92                  label_set(u.value).sets(i).battery_SoC=B_range(2);
93              end
94          end
95
96          %%
97          %*********************************************************
98          %**Eighth step:
99          %**When present search vertex equal to target vertex,
100         % stop*****************************************************
101         %*********************************************************
102         if u==Fnodes(target)
103             the_label = label_set(u.value).settle_label();
104             label_set_settled(u.value).add_label(the_label);
105             label_set_settled(u.value).check_label();
106             break
107         end
108
109         %%
110         %*********************************************************
111         %**Nineth step:
112         %***************Settle the current label******************
113         %*********************************************************
114         the_label = label_set(u.value).settle_label();
115         label_set_settled(u.value).add_label(the_label);
116         label_set_settled(u.value).check_label();
117         if label_set(u.value).has_unsettled_labels()
118             Fnodes(u.value)=FibonacciHeapNode();
119             Fnodes(u.value).value=u.value;
120             Fnodes(u.value).key=label_set(u.value).key_label();
121             Q.ninsert(Fnodes(u.value));
122         end
123
124         %%
125         %*********************************************************
126         %**Tenth step:
```

```
127         %********Check all edges connected to vertex u**********
128         %*****************************************************
129 
130         i_s = Graph_index(u.value);
131         if u.value==length(Graph_index)
132             i_e = i_s;
133         else
134             i_e = Graph_index(u.value+1)-1;
135         end
136         for i=i_s:i_e
137 
138             junc_to = edges(i,2);
139             [driving_time,energy_consumption]=speed_curve(edges(i,4),
    edges(i,5),edges(i,3),edges(i,6));
140             new_distance = the_label.distance + edges(i,3);
141             battery_SoC = the_label.battery_SoC-energy_consumption;
142             travel_time = the_label.travel_time + driving_time;
143             if battery_SoC>B_range(1)&&label_set(junc_to).
    has_unsettled_labels()
144                 new_label = Labels_3(new_distance,battery_SoC,u.value
    ,energy_consumption,travel_time);
145                 mini_label = label_set(junc_to).min_label;
146                 if ~mini_label.label_le(new_label)
147                     label_set(junc_to).add_label(new_label);
148                     label_set(junc_to).check_label();
149                     if Fnodes(junc_to).key==inf
150                         Fnodes(junc_to).key=travel_time*ratio(1)+
    energy_consumption*ratio(2);
151                         Q.ninsert(Fnodes(junc_to));
152                     elseif Fnodes(junc_to).key>new_distance*ratio(1)+
    energy_consumption*ratio(2)
153                         Q.key_decreasing(Fnodes(junc_to),travel_time*
    ratio(1)+energy_consumption*ratio(2));
154                     end
155 
156                 end
157             end
158         end
159     end
```

Appendix N

LabelSet_5.m

```
classdef LabelSet_5 < handle

    properties
        sets
        distances
        battery_SoC
        energy_consumption
        ratio   % wieght of distance and energy_consumption, default
    =[1,0];
        travel_time
        n
    end

    methods

        function x=LabelSet_5(ratio)
            x.sets=Labels_3.empty;
            x.distances=[];
            x.battery_SoC=[];
            x.energy_consumption=[];
            x.travel_time=[];
            x.n=0;
            if(~exist('ratio','var'))
                ratio=[1,0];
            end
            x.ratio = ratio;
            x.add_label(Labels_3(inf,0,0,inf))
        end

        function add_label(x,label)
            x.n=x.n+1;
```

```
31              x.sets(x.n)=label;
32              x.distances(x.n)=label.distance;
33              x.battery_SoC(x.n)=label.battery_SoC;
34              x.energy_consumption(x.n)=label.energy_consumption;
35              x.travel_time(x.n)=label.travel_time;
36          end
37
38          function key=key_label(x)
39              key=min(x.travel_time*x.ratio(1)+x.energy_consumption*x.
    ratio(2));
40          end
41
42          function y=min_label(x)
43              [~,idx] = min(x.travel_time*x.ratio(1)+x.
    energy_consumption*x.ratio(2));
44              idx=idx(1);
45              y=x.sets(idx);
46          end
47
48          function y=settle_label(x)
49              [~,idx] = min(x.travel_time*x.ratio(1)+x.
    energy_consumption*x.ratio(2));
50              idx=idx(1);
51              y=x.sets(idx);
52              x.sets(idx)=[];
53              x.distances(idx)=[];
54              x.battery_SoC(idx)=[];
55              x.energy_consumption(idx)=[];
56              x.travel_time(idx)=[];
57              x.n=x.n-1;
58          end
59
60          function check_label(x)
61              while 1
62                  [~,idx_travel]=max(x.travel_time);
63                  [~,idx_ene]=max(x.energy_consumption);
64                  [~,idx_soc]=min(x.battery_SoC);
65                  if (idx_travel==idx_ene)&&(x.n~=1)&&(idx_ene==idx_soc
    )
66                      x.sets(idx_travel)=[];
67                      x.distances(idx_travel)=[];
68                      x.battery_SoC(idx_travel)=[];
69                      x.energy_consumption(idx_travel)=[];
70                      x.travel_time(idx_travel)=[];
71                      x.n=x.n-1;
72                  else
73                      break
74                  end
75              end
```

```
76          end
77
78          function t=has_unsettled_labels(x)
79              if x.n==0
80                  t=false;
81              else
82                  t=true;
83              end
84          end
85
86      end
87  end
```

Appendix O

Labels_3.m

```
classdef Labels_3 < handle
    properties
        distance
        battery_SoC
        energy_consumption
        last_vertex
        travel_time
    end
    methods
        function x=Labels_3(distance,battery_SoC,last_vertex,
    energy_consumption,travel_time)
            if(~exist('distance','var'))
                distance=inf;
            end
            if(~exist('battery_SoC','var'))
                battery_SoC=0;
            end
            if(~exist('last_vertex','var'))
                last_vertex=0;
            end
            if(~exist('energy_consumption','var'))
                energy_consumption=inf;
            end
            if(~exist('travel_time','var'))
                travel_time=inf;
            end
            x.distance=distance;
            x.battery_SoC=battery_SoC;
            x.last_vertex=last_vertex;
            x.energy_consumption=energy_consumption;
            x.travel_time=travel_time;
```

```
31          end
32          function bo = label_le(x,other_label)
33              if x.travel_time<=other_label.travel_time&&...
34                 x.energy_consumption<=other_label.energy_consumption
35                  bo=true;
36              else
37                  bo=false;
38              end
39          end
40      end
41  end
```

Bibliography

[1] E.W. DIJKSTRA. «A Note on Two Problems in Connexion with Graphs.» In: *Numerische Mathematik* 1 (1959), pp. 269–271. URL: http://eudml.org/doc/131436 (cit. on p. 2).

[2] F. Zhan and Charles Noon. «Shortest Path Algorithms: An Evaluation Using Real Road Networks». In: *Transportation Science* 32 (Feb. 1998), pp. 65–73. DOI: 10.1287/trsc.32.1.65 (cit. on p. 2).

[3] Martin Sachenbacher, Martin Leucker, Andreas Artmeier, and Julian Haselmayr. «Efficient energy-optimal routing for electric vehicles». In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 25. 1. 2011, pp. 1402–1407 (cit. on p. 2).

[4] Michael T. Goodrich and Paweł Pszona. *Two-Phase Bicriterion Search for Finding Fast and Efficient Electric Vehicle Routes*. 2014. arXiv: 1409.3192 [cs.DS] (cit. on p. 2).

[5] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*. 2015. arXiv: 1504.05140 [cs.DS] (cit. on p. 2).

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844 (cit. on p. 2).

[7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. «A Formal Basis for the Heuristic Determination of Minimum Cost Paths». In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (cit. on p. 3).

[8] Moritz Hilger, Ekkehard Köhler, Rolf Möhring, and Heiko Schilling. «Fast Point-to-Point Shortest Path Computations with Arc-Flags». In: vol. 74. July 2009, pp. 41–72. ISBN: 9780821843833. DOI: 10.1090/dimacs/074/03 (cit. on p. 3).

[9] Alexandros Efentakis, Dieter Pfoser, and Agnès Voisard. «Efficient Data Management in Support of Shortest-Path Computation». In: *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Computational Transportation Science*. CTS '11. Chicago, Illinois: Association for Computing Machinery, 2011, pp. 28–33. ISBN: 9781450310345. DOI: 10.1145/2068984.2068990. URL: https://doi.org/10.1145/2068984.2068990 (cit. on p. 3).

[10] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. «PHAST: Hardware-accelerated shortest path trees». In: *Journal of Parallel and Distributed Computing* 73.7 (2013). Best Papers: International Parallel and Distributed Processing Symposium (IPDPS) 2010, 2011 and 2012, pp. 940–952. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2012.02.007. URL: https://www.sciencedirect.com/science/article/pii/S074373151200041X (cit. on p. 3).

[11] Michael L. Fredman and Robert Endre Tarjan. «Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms». In: *J. ACM* 34.3 (July 1987), pp. 596–615. ISSN: 0004-5411. DOI: 10.1145/28869.28874. URL: https://doi.org/10.1145/28869.28874 (cit. on p. 13).

[12] Mar. 2019. URL: https://www.sae.org/site/news/2019/03/aaa-ev-range-thermal-effect-study (cit. on p. 26).