MUSE: Multimodal Separators for Efficient Route Planning in Transportation Networks

Mohamed Amine Falek, Cristel Pelsser, Sébastien Julien, Fabrice Theoleyre

TRANSPORTATION SCIENCE
Vol. 00, No. 0, Xxxxx 0000, pp. 000–000
ISSN 0041-1655 | EISSN 1526-5447 | 00 | 0000 | 0001

INFORMS
DOI 10.1287/xxxx.0000.0000

MUSE: Multimodal Separators for Efficient Route Planning in Transportation Networks

Amine Mohamed Falek
Technology & Strategy group, 4 rue de Dublin, 67300 Schiltigheim, France.
falek@unistra.fr - a.falek@technologyandstrategy.com

Cristel Pelsser
ICube Lab, CNRS/University of Strasbourg, Pole API, Boulevard Sebastien Brant, 67412 Illkirch, France.
pelsser@unistra.fr

Sebastien Julien
Technology & Strategy group, 4 rue de Dublin, 67300 Schiltigheim, France.
s.julien@technologyandstrategy.com

Fabrice Theoleyre
ICube Lab, CNRS/University of Strasbourg, Pole API, Boulevard Sebastien Brant, 67412 Illkirch, France.
theoleyre@unistra.fr http://www.theoleyre.eu

Many algorithms compute shortest-path queries in mere microseconds on continental-scale networks. Most solutions are, however, tailored to either road or public transit networks in isolation. To fully exploit the transportation infrastructure, multimodal algorithms are sought to compute shortest-paths combining various modes of transportation. Nonetheless, current solutions still lack performance to efficiently handle interactive queries under realistic network conditions where traffic jams, public transit cancelations, or delays often occur. We present MUSE, a new multimodal algorithm based on graph separators to compute shortest travel time paths. It partitions the network into independent, smaller regions, enabling fast and scalable preprocessing. The partition is common to all modes and independent of traffic conditions so that the preprocessing is only executed once. MUSE relies on a state automaton that describes the sequence of modes to constrain the shortest path during the preprocessing and the online phase. The support of new sequences of mobility modes only requires the preprocessing of the cliques, independently for each partition. We also augment our algorithm with heuristics during the query phase to achieve further speedups with minimal effect on correctness. We provide experimental results on France's multimodal network containing the pedestrian, road, bicycle, and public transit networks.

Key words: multimodal shortest path; graph separators, route planning; time-dependent graph

1. Introduction

The well-being of the economy and even social infrastructure is highly reliant on an efficient transportation network. According to the U.S Department of Transportation (DOT 2015), 40% of all traveled passenger-kilometers consists of commuting. While commuting distances are increasing, average speeds are, however, decreasing due to steadily rising congestion. By 2015, the average commuter typically spent 40 hours stuck in traffic, costing $121 billion annually. For decades, personal-vehicle travel has been, and remains, the dominating trend: 105 million American commuters rely mostly on driving while the remaining 32 million depend on all other transportation

modes. Nevertheless, although accounting for only 5% of the overall commute trips, public transit is vital to alleviating congestion. The same report indicates that if public transit users in major metropolitan areas of the united states suddenly reverted to driving, congestion is estimated to increase by 24% and cost an additional $17 billion annually. Fortunately, over the past two decades, public transportation thrived with a registered 25% increase in public transit ridership. Enhanced access to information significantly improved transit networks and steered urban populations to consider better alternatives to their private cars.

Multimodal algorithms are thereby of increasing importance for route planning to exploit the diversity of transportation infrastructure fully. A central problem to route planning is the ability to compute *shortest paths* efficiently. Most commonly, the shortest path denotes that which requires minimal travel time to reach the destination. In that regard, researchers accomplished significant progress, and as a result, the literature is abundant (Bast et al. 2016). Early algorithms such as Dijkstra's (Dijkstra 1959) solve the *one-to-many* shortest path problem by greedily exploring the graph data structure that abstracts the transportation network. However, for large networks with up to several millions of vertices, Dijkstra's algorithm becomes too slow for practical applications. Therefore, newer algorithms tend to split the problem into two parts (i) in a first *offline* phase, known as preprocessing, additional data is extracted from the graph based on some of its unique features, (ii) in a second *online* phase, known as the query, the preprocessed data is used to speed up a variant of Dijkstra's algorithm. Ultimately, efficiency is measured as a tradeoff between query time, preprocessing time, and memory requirements.

The 9^{th} Dimacs Challenge (Demetrescu, Goldberg, and Johnson 2009) was a competition on the shortest path problem and renowned for the significant contribution it brought to the field of route planning in road networks. Algorithms such as *Arc Flags* (Hilger et al. 2009), *Reach* (Goldberg, Kaplan, and Werneck 2006), *Transit Node Routing* (Bast, Funke, and Matijevic 2006) and *Highway Hierarchies* (Delling et al. 2006) layed the foundations for modern algorithms.

For instance, the commercial solution *TomTom* uses a variant of Arc Flags (Schilling 2012) and the Open Source Route Machine (OSRM) (Huber and Rust 2016) runs a faster version of Highway Hierarchies known as *Contraction Hiearchies* (Geisberger et al. 2008). Bast (2009) summarizes the key *ingredients* behind many algorithms that achieved speedups of several orders of magnitude. Techniques such as goal direction, contraction, and hierarchy stem from fundamental properties observed in road networks. For that reason, the same techniques proved to be much less effective when applied to public transit networks, which are time-dependent and lack hierarchy at the intracity level (Bast 2009).

The *time-expanded* model (Pyrga et al. 2004, 2008) is an event-based graph used to model transit networks. Mainly, a single station is abstracted with many vertices, one for each event (arrival, departure, and transfer). Thereby, techniques such as bidirectional search become challenging: although the target station is known, the specific target vertex is not, and the backward search is not trivial. Another approach, the *time-dependent* model (Brodal and Jacob 2004) is more compact and treats each station as a unique vertex in the graph, consequently providing better performance. Nonetheless, the same speedup techniques applied to road networks are orders of magnitude lower when applied to transit networks, as Bauer et al. (Bauer, Delling, and Wagner 2011) experimentally show. Unlike road networks, transit networks are modeled according to timetable information (Müller-Hannemann et al. 2007). Instead of a graph structure, some of the fastest algorithms such as *Round-Based Public Transit Routing* (RAPTOR) (Delling, Pajor, and Werneck 2014) and *Connection Scan Algorithm* (CSA) (Dibbelt et al. 2013) operate directly on the timetable data.

The structural differences between road and public transit networks represent the main challenge for efficient multimodal algorithms. Moreover, modal constraints, which are typically imposed by users, must be dealt with to compute the shortest path with a valid sequence of transportation types. We tackle this problem aiming for a practical solution with fast queries (few milliseconds) for interactive applications and fast preprocessing times (few minutes) to account for traffic, unexpected congestion, and transit delays.

Customizable Route Planning (CRP) (Delling et al. 2011a) is a road network algorithm based on multilevel separators. Initially, the graph is partitioned, aiming to minimize the number of boundary vertices. Then, an overlay is constructed by computing full-cliques across all boundary vertices within each cell in the partition. Finally, during a query, bidirectional-Dijkstra is run on the query graph combining the source and target cells and the overlay, allowing to skip most of the vertices in the underlying graph.

We detail here MUSE, a MUltimodal SEparators-based algorithm, extending CRP to handle the multimodal travel computation. By combining label constraints with a profile label correcting algorithm during preprocessing, we can efficiently handle modal constraints and time-dependency. Our main contributions are the following:

- We present a multimodal graph partitioning approach, with a label correcting algorithm to compute multimodal time-dependent cliques efficiently;
- We associate the multimodal graph to a labeled automaton to reduce the number of vertices in the product graph. It reduces the memory footprint and achieves faster preprocessing times;
- We experimentally evaluate our algorithm on a country-scale network with different heuristics for faster queries.
- We provide an open-source tool to construct multimodal networks and a unified dataset to serve as a benchmark for multimodal route planning algorithms.

We first review in section 2 the relevant multimodal route planning algorithms. We then detail in section 3 the model we used to represent a realistic multimodal network. In section 4, we explain each stage of our solution. Mainly, we discuss partitioning in section 4.1, the process of computing a multimodal overlay in section 4.2, and the algorithm and heuristics used to answer queries in section 4.3. The experimental setup, the results, and associated discussion are available in section 5. Finally, in section 6, we detail our conclusion and possible future work.

2. Related Work

Multimodality involves considering multiple transit networks. The public transit network is typically modeled with a time-dependent graph (Brodal and Jacob 2004): an edge exists in the graph when a shuttle connects both vertices. Inversely, the road and other unrestricted networks (pedestrian and bicycle) are time-independent: the graph structure remains unchanged for very long timescales.

2.1. Road Networks

Route planning in road networks involves computing a path that provides the shortest travel time. However, road congestion is very frequent, and travel duration may vary in time. Thus, Baum et al. (2016) compute a profile search, to compute the minimum travel time for all departure times of the day. They compute a multilevel partition of the graph, constructing an overlay. Then, the approach stores the clique matrix to compute the paths between the different border vertices in the overlay. The preprocessing has to be re-executed only for the affected cells in the partition to handle live traffic. MUSE extends this work to deal with multimodal networks.

2.2. Public Transit networks

Transit (Antsfeld and Walsh 2012) operates on a graph combining different public transit networks and evaluates the shortest path based on the associated risk of transfers (probability to miss one transfer and its impact on travel time) on a combination of transit networks. We believe, however, that *true* multimodal networks must combine both unrestricted networks such as road and pedestrian networks and schedule-based networks such as trains, trams, and buses. Otherwise, a classical time-dependent variant of Dijkstra's algorithm (Bauer, Delling, and Wagner 2011) would suffice to solve the problem.

Bast, Hertel, and Storandt (2016) exploit the natural hierarchy of local and long-distance transports by partitioning the graph. Each stop is assigned to a cluster so that the number of inter-cluster

edges is minimized. Then, an existing algorithm is executed to precompute the paths inside each cluster. Only the local transport connections are considered for clustering since long-distance transportation should rather interconnect the clusters. Delling et al. (2017) also partition the transit network to speed up the computation. They consider a hyper-graph where the vertices correspond to routes (bus/subway) and edges to stops. Typically, multiple public transport agencies are sparsely interconnected and would correspond to different partitions. Some routes are then precomputed in the overlay so that the query can only process the source and target clusters and these overlay routes. In MUSE, we also propose to partition the graph, but we consider rather multimodal networks.

Cionini et al. (2017) exploit a Dynamic Timetable Model (DTM): the graph is updated when a delay is observed in the public transit network. In particular, DTM reduces the number of changes to apply to the graph when a timetable has to be modified. The authors only modify the graph structure to model the public transit network. Thus, a classical algorithm such as ALT (Goldberg and Harrelson 2005) may be directly applied to the modified graph to compute the shortest routes.

2.3. Multimodal route planning

Multimodal networks are usually modeled with labeled graph structures. Vertices belonging to the same modal network are marked with a similar label, and *link* edges (aka. connecting edges) are added to account for transfers from one mode to another. In that context, the shortest path problem comes with additional constraints that restrict the sequence of transportation during query time. Furthermore, path *feasibility* must be ensured: even if the private car is chosen as a valid means, the path is not feasible if it requires using the car after a bus ride.

Minimizing the travel time may not be sufficient in multimodal networks: the number of transfers/modes or the price may also be considered. Three main approaches exist in the literature (Giannakopoulou, Paraskevopoulos, and Zaroliagis 2019):

1. Scalar approaches: multiple metrics are transformed into a single objective, for instance, with a weighted sum. A transfer may typically be transformed into a time penalty to optimize the travel time uniquely. While such an approach reduces the complexity, combining different metrics is often reductive;
2. Pareto sets: the algorithm keeps all the optimal paths for at least one criteria. This set may be quite large, and some heuristics exist to prune the less interesting elements;
3. Label constrained shortest paths: the graph is explored while respecting a sequence of possible modes. These modes are often represented with a Finite State Automata.

2.3.1. Scalar approaches: A few approaches handle only a single objective. Ulloa, Lehoux-Lebacque, and Roulland (2018) propose XTP, an iterative approach to merge the static road network and the public transit network. The shortest path is computed at each iteration from the previously reached bus stops to the next ones. While this approach works well for small networks (*e.g.*, on a metropolitan scale), the complexity becomes intractable for large networks.

The road network subgraph in a multimodal network is usually far bigger (and denser) than the public transit network. *Access-Node Routing* (ANR) (Delling, Pajor, and Wagner 2009) was designed to exploit this property. It borrows ideas from TNR (Bast, Funke, and Matijevic 2006) to perform table lookups on the road network and restrict the search on the public transit network. ANR pre-computes *access-nodes*, a set of vertices in the multimodal graph forming a boundary between the road and the public transit network. For each vertex in the road network, a profile-graph consisting of the shortest paths to all access-nodes is precomputed, requiring significant additional memory. An alternative technique, core-based ANR, hierarchically contracts the road network first. Subsequently, access-nodes are only evaluated for the *core*, i.e., the contracted graph, with significantly fewer vertices.

2.3.2. Pareto sets: *User Constrained Contraction Hierarchies* (UCCH) (Dibbelt, Pajor, and Wagner 2015) is a hierarchical technique originally designed for road networks. The main idea is to contract while preprocessing the graph vertices ordered by their measured *importance*. For each contracted vertex, a shortcut edge is inserted between its neighboring vertices and whose weight preserves the cost of the shortest path containing the contracted vertex. The algorithm solves queries by running a bidirectional Dijkstra, only scanning vertices with higher importance until the forward and the backward search meet. To separate modal constraints from preprocessing, the authors initially split all vertices into different sets based on their labels. They contract each set of vertices separately to make sure that all shortcut edges have a unique label. Thus, UCCH is scalable since it handles each mode of transport separately. However, travel times are often unpredictable in road networks. Thus, a weight change implies that the whole hierarchy has to be recomputed: routes may become suboptimal. We propose instead in MUSE to recompute the paths only for the geographical areas where the weight change occurred.

Sauer, Wagner, and Zündorf (2020) propose to compute journeys in multimodal networks, with several bike-sharing operators. The operator-dependent variant introduces a new label identifying each operator which can only be changed at the specific bike-sharing stations. Handling labels allow the authors to compute Pareto sets, the bike operator being a criterion. The second variant expands the graph to execute any public transit algorithm: each operator corresponds to a layer. A pruning step helps to reduce the preprocessing time when different operators are far from each other.

ULTRA (Baum et al. 2019) constructs the routes in a public transit network, with transfers using an unrestricted mode (*e.g.*, walking or taxi). More precisely, ULTRA computes shortcuts in the transfer graph (with the unrestricted mode) to connect pairs of stops in the public transit network. ULTRA minimizes the number of shortcuts while still guaranteeing that Pareto-optimal journeys can be extracted. Giannakopoulou, Paraskevopoulos, and Zaroliagis (2019) also adopt a similar approach, extending the Dynamic Timetable Model: they update timetables when a delay is observed. The public transit network is modeled with a time expanded graph, and some additional arcs are inserted to connect two stations through an unrestricted mode (*i.e.*, walking or driving). The authors group vertices in the graph to accelerate the computation. However, the unrestricted mode graph is considered static, while congestion may be frequent in urban environments. Moreover, inserting additional arcs may be expensive in large graphs (*e.g.*, for a whole country).

2.3.3. Label constrained: *State-Dependent ALT* (SDALT) (Kirchler et al. 2011) adapts ALT (Goldberg and Harrelson 2005), a popular goal directed algorithm, for multimodal queries. The intuition is to speed up Dijkstra with a heuristic based on the triangle inequality observed in transportation networks. In essence, it computes during preprocessing a set of landmark vertices. Then, for each landmark, it constructs a constrained shortest-path tree to (and from) all other vertices in the graph. During query time, the preprocessed shortest path costs are used to assign a potential to each vertex representing the tentative distance to reach the target vertex. It relies on D_{RegLC} (Barrett, Jacob, and Marathe 2000), a multimodal version of Dijkstra, constrained with predefined automata. The complexity of the automaton impacts both preprocessing and query times. The main drawback of this approach corresponds to its scalability: computing landmark distances becomes too costly for large graph instances.

3. Model and Assumptions

Transportation networks are usually modeled with *graph* structures for their intuitiveness and the extensive algorithmic toolbox of graph theory (Thomson and Richardson 1995). Mainly, we model a multimodal network using a labeled directed graph with time-dependent edge costs. It consists of multiple *layers* of unimodal networks that are interconnected via *link* edges.

We detail in sections 3.1 through 3.4 each unimodal network followed by the process of computing link edges in section 3.5 to obtain the multimodal graph. For better readability, we summarize recurring notation in table 1. Following are definitions consistently used throughout the paper:

Table 1 keywords and symbol definitions

notation	definition
$G(V,E)$ or G	Directed graph with V vertices and E edges.
$G(V,E,\Sigma)$ or G^{Σ}	Directed graph, labeled with alphabet set Σ.
$E^{road} \subset E$	routes/streets that cars may take
$E^{foot} \subset E$	paths that pedestrians may take
$\mathcal{C}$	set of elementary connections between two public stations (with their time schedule)
$V^{road} \subset V$	crossroads that cars may take
$V^{rentcar} \subset V$	renting stations for shared vehicles
$V^{park} \subset V$	parking spots, where parking a car is authorized
$V^{foot} \subset V$	crossroads that pedestrians may take
$V^{bike} \subset V$	crossroads authorized for bikes
$V^{rentbike} \subset V$	renting stations for shared bikes
$V^{stations} \subset V$	stations of the public transit network
$V^{platform} \subset V$	platforms, each of them attached to a given station
$speed_{walk} \in \mathbb{R}$	average speed of pedestrians
$speed_{bicycle} \in \mathbb{R}$	average speed of bicycles
c, f, b, p	modes: *car*, *foot*, *(private or rental) bicycle*, and *public*
c_r and c_s	to denote when a car is *rented* and *restored*
b_r and b_s	to denote when a bike is *rented* and *restored*
$(v,w) \in E$	Directed edge from vertex v to w.
$c(v,w)$	The cost, represents travel time $[s]$ of (v,w).
$c(v,w,\tau)$	Time dependent cost of (v,w) at time τ.
$len(v,w)$	Physical length $[m]$ of (v,w).
$lab(v,w)$	Label attached to (v,w).
$P = \{v_0, v_1, .., v_k\}$ or $P_{v_0 v_k}$	A path is an ordered set of vertices $v_i \in V(G)$.
$c(P,\tau)$	The cost of path P when departing at time τ.
$d(r,t,\tau)$	The cost of the shortest path P_{rt} departing at τ.
$word(P) = \cup_{i=0}^{k-1} lab(v_i, v_{i+1})$	The sequence of edge labels associated to path P.
$\mathcal{A} = \{S, \Sigma, \delta, s_0, F\}$	Deterministic Finite Automaton (DFA) consists of a set of states S and alphabet Σ, a transition function δ, an initial state s_0, and a set of final states F. See section 4.2.
$G(V^{\times}, E^{\times})$ or $G^{\times} = G^{\Sigma} \times G^{\mathcal{A}}$	Product graph merging graph G^{Σ} and graph automaton $G^{\mathcal{A}}$.
$\langle v, s \rangle \in V^{\times}$	Product vertex is a pair of vertex $v \in V(G^{\Sigma})$ and a state $s \in S(\mathcal{A})$.
$\big(\langle v, s \rangle, \langle w, s' \rangle\big) \in E^{\times}$	Product edge, requires a valid transition $s' \in \delta(s, lab(v,w))$.

A directed Graph $G(V,E)$ consists of a set of vertices $v \in V$, and directed edges $(v,w) \in E$ connecting vertices $v, w \in V$. A vertex is an abstraction of a physical entity in the network: an intersection of road segments in the road network or a public transit network station. Throughout the paper, all graphs are considered directed unless otherwise specified.

The Edge Cost Function $c(v,w,\tau)$ (also written $f_{vw}(\tau)$ in the literature) represents the time required to travel from vertex v to vertex w, when departing at time τ. This variable is referred to as the travel-time. This cost is a periodic positive piece-wise linear function $f : \Pi \to \mathbb{R}^+$ where $\Pi = [0,p] \subset \mathbb{R}$ with a period $p \in \mathbb{N}$. If f is constant, then the edge belongs to a time-independent network such as the foot or bicycle networks; otherwise, the edge is said to be time-dependent. In public transportation, the period is typically one week, and the cost function is typically a non-continuous time function: the travel time increases suddenly when a bus/train leaves a specific

station. In road networks, the cost depends on the congestion and is rather a continuous time function. However, this cost is unknown a priori since making mid or long-term traffic forecasting is still an open problem (Chen et al. 2019).

In any case, the *FIFO* property is maintained to ensure polynomial complexity (Kaufman and Smith 1993). Also known as the non-overtaking property, it holds that for all $\tau_1, \tau_2 \in \Pi$ such that $\tau_1 \leq \tau_2$ then $\tau_1 + f(\tau_1) \leq \tau_2 + f(\tau_2)$. In other words, waiting at a vertex never pays-off.

A Path $P = \{v_0, v_1, .., v_k\}$, also written $P_{v_0 v_k}$, is an ordered sequence of vertices $v_i \in V$. Its associated cost is recursively evaluated with the edge cost of its edges by $c(v_0, v_1, \tau) + c\big(\{v_1, .., v_k\}, \tau + c(v_0, v_1, \tau)\big)$ when departing from v_0 at time τ. For a *query* $q(r, t, \tau)$ with $r, t \in V$, our goal is to compute the *shortest path* P with the smallest cost denoted by $d(r, t, \tau)$.

3.1. Road Network (private cars, taxis, and rental vehicles)

Structurally, road networks consist of intersecting road segments. Each segment is characterized by its length and a specific speed-limit, which can be used to derive the cost function. In its graph representation, each edge $(v, w) \in E^{road}$ represents a road segment, and the vertices $v, w \in V^{road}$ mark the junction of two or more road segments.

For a realistic model, though, dynamic traffic conditions must be taken into account as speed can unpredictably vary. Thus, we rely on speed measurements to construct the cost function. For each segment, we collect a set of speed values over a time window Π sampled at a fine-grained rate Δt (we detail our dataset in the experimental evaluation of section 5). Then, for each edge (v, w) we construct its speed profile as a piece-wise linear function f_{vw}. During query time, we compute the cost $c(v, w, \tau_1)$ of departing from v at time τ_1 by evaluating the area under the speed-curve, adjusting the arrival time τ_2 such that: $\int_{\tau_1}^{\tau_2} f_{vw} dt = len(v, w)$ where $len(v, w)$ is the length of the edge. This is a trivial geometric computation considering the function is piecewise linear. Solving the integral for τ_2, travel-time is given by $c(v, w, \tau_1) = \tau_2 - \tau_1$.

Moreover, getting onto or off of a car is only permissible where parking is possible. Hence, for each vertex belonging to a road segment where parking is authorized (typically excludes highways, tunnels, bridges, and sidewalks), we label it as a suitable parking spot $v \in V^{\text{park}} \subset V^{road}$. Furthermore, instead of restricting the road network to private driving only, we consider alternative options, including rental vehicles and on-demand services such as Uber and taxis. While on-demand vehicles are typically accessible at every vertex $v \in V^{\text{park}}$ (we discuss the additional incurred waiting cost in section 5), rental vehicles are only available at rental stations. Thus, we add a vertex $v \in V^{\text{rentcar}} \subset V^{road}$ for each rental station and insert an edge $(v, w) \in E$ to connect the station to the closest junction w in the road network.

3.2. Foot Network

The foot network is represented by a time-independent graph (V^{foot}, E^{foot}). Vertices V^{foot} represent junctions and edges E^{foot} are added for each footpath including sidewalks, bridges, and stairs. The cost of an edge (v, w) is thereby a constant given by $c(v, w) = len(v, w)/speed_{walk}$ where $speed_{walk}$ represents the average pedestrian walking speed.

3.3. Bicycle Network

Similar to a foot network, the bicycle network is based on a time-independent graph $G(V, E)$ where vertices V represent junctions and edges E depict either cycling lanes or road segments where biking is allowed (typically non-motorway road segments). The cost $c(v, w) = len(v, w)/speed_{bicycle}$ is evaluated based on average cycling speed $speed_{bicycle}$.

Besides private bicycles, bicycle-sharing systems are widespread in urban cities and are efficient for fast transfers between nearby public transit stations. Rental bicycles must, however, often be picked up and returned at specific locations (bicycle stations). Thus, we add a vertex $v \in V_{\text{rentbike} \subseteq V}$ for each rental station and an edge (v, w) between the station and its closest junction in the bicycle network.

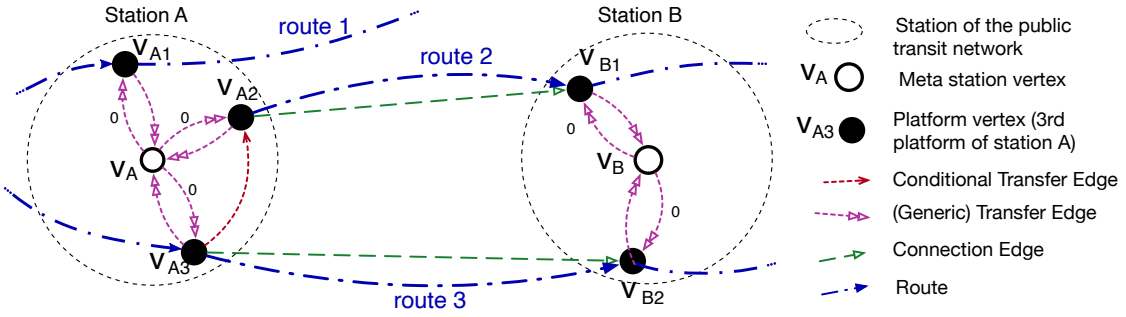


Figure 1 **Time-dependent graph representing the public transit network**

3.4. Public Transit Network

The public transit network is based on a timetable $\mathcal{T} = (\mathcal{Z}, V^{stations}, \mathcal{C})$ which consists of a set of shuttle vehicles $\mathcal{Z}$, a set of stations $V^{stations}$, and a set of *elementary connections* $\mathcal{C}$. An elementary connection is a 5-tuple $c = \{z, v_d, v_a, t_d, t_a\} \in \mathcal{C}$ and represents a unique shuttle $z \in \mathcal{Z}$ departing from a station $v_d \in V^{stations}$ at time t_d and arriving at another station $v_a \in V^{stations}$ at time t_a without a stop at any intermediate station.

We define a *trip* as a sequence of elementary connections $\{c_0, c_1, .., c_k\}$ fulfilled by a single vehicle such that $\forall i = 1, 2, .., k,\ c_i = \{z \in \mathcal{Z}, v_d(i) \in V^{stations}, v_a(i) \in V^{stations}, t_d(i), t_a(i)\}$, where $v_d(i) = v_a(i-1)$. A trip typically denotes the itinerary of a single-vehicle scheduled at a particular time. We group multiple trips into a single *route*, which is time-independent and denotes a fixed ordered sequence of stations. To obtain the set of routes $\mathcal{R}$, we iterate over all trips and extract for each of them, a route $r = \{v_d(i)\}_{i\in[0,k]} \cup \{v_a(k)\}$ that we add to $\mathcal{R}$ *iff* $r \notin \mathcal{R}$. We denote by $R_{sub}(v) \subset \mathcal{R}$ the subset of routes passing through station $v \in V^{stations}$, that is $\forall r \in R_{sub}(v), v \in r$.

We have to model the transfers between different trips and routes to construct a realistic time-dependent graph $G(V, E)$ from the timetable $\mathcal{T}$. We proceed as follows:

- $\forall v \in V^{stations}$, $\forall r \in R_{sub}(v)$, we add a *platform vertex* $v_p \in V^{platforms}$, modeling the *platform* in the station for the corresponding route. Let us denote by $f_{station} : (V^{stations}, R_{sub}(v)) \rightarrow V^{platforms}$ the bijective function connecting each pair of station and route to its *platform vertex*.
- if we know exactly the transfer time between two platforms, $\forall v \in V^{stations}$, we add *conditional transfer edges* $(v_p, v_{p'})$ between platform vertices $v_p, v_{p'} \in V^{platforms}$ of the same station, i.e., $f_{station}(v_p) = f_{station}(v_{p'})$. Their cost is the transfer time (walking time) from one platform to another;
- If the transfer time is not known exactly, we insert *generic transfer edges*. We compute the average walking time to go to a central point in the station, denoting a fixed transfer cost $c(v_p, f_{station}(v_p))$ toward any other platform of the same station $f_{station}(v_p)$. As $c(v_p, f_{station}(v_p))$ includes the full transfer cost, $\forall v'_p \in V^{platforms} | f_{station}(v_p) = f_{station}(v'_p)$, we add an edge $(f_{station}(v_p), v'_p)$ with $c(f_{station}(v_p), v'_p) = 0$ to preserve the connectivity: we consider that the transfer time has already been considered when the passenger arrives at the station vertex.
  These transfer edges are also used when a passenger exits the transit network.
- $\forall c \in \mathcal{C}$, we add a *connection edge* representing one "edge" for each edge of a route followed by a collection of vehicles. A *connection edge* $(v_p, v_{p'})$ is inserted between the platform vertices $v_p \in V^{platforms}$ and $v_{p'} \in V^{platforms}$ where $\exists c_i \in \mathcal{R} | v_p = v_d(i)$ and $v_{p'} = v_a(i)$ (a vehicle leaves $v_d(i)$ and arrives at $v_a(i)$). Its cost $c(v_p, v_{p'}, \tau) = t_a(i) - t_d(i)$ is time dependent, and depends on the timetable. It is worth noting that we consider non-multi-edge graphs: a connection edge is inserted between two consecutive platforms of a route. Since different routes correspond to different platforms, we have at most one edge between any pair of platforms (either a (conditional) transfer edge or a connection edge depending on whether the platforms correspond to the same station or not).

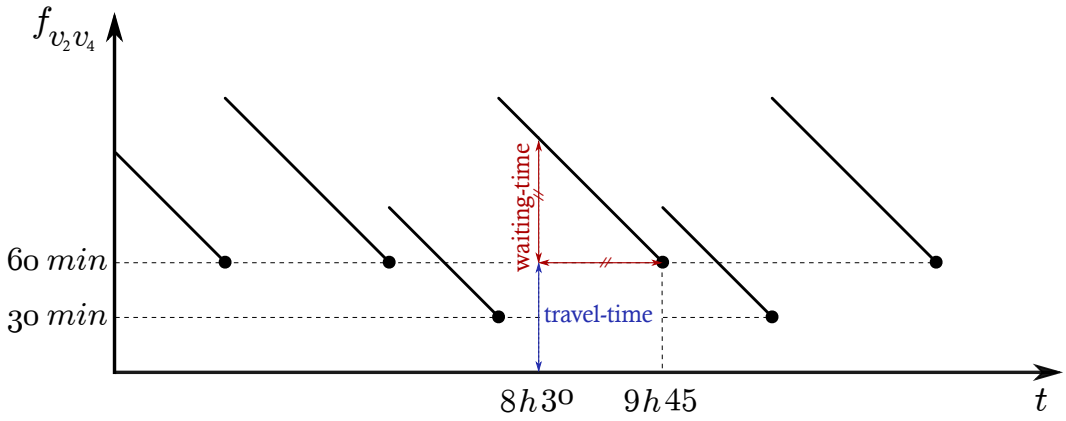


Figure 2 **Piecewise linear function describing the cost of edge (v_{A2}, v_{B1}) from figure 1. Six trains are scheduled: four regular trains of 60 min travel-time and two fast trains of 30 min travel-time. Upon arriving to the station at 8:30, the next shuttle is a regular train scheduled to depart at 9:45.**

The resulting graph is illustrated in figure 1. A hypothetical journey may begin at station v_A at $\tau =$ 8:30. There, we wait for the next train departing from the second platform (vertex v_{A2}) towards station v_B via route 2, reaching the first platform v_{B1}. The cost function of edge (v_{A2}, v_{B1}) denoted $f_{v_{A2},v_{B1}}$ is depicted in figure 2. The total cost for the traversal of (v_{A2}, v_{B1}) at $t = \tau =$ 8:30 is $c(v_{A2}, v_{B1}, \tau) = 135min$, comprising both the waiting and the travel time. Because the slope $df_{v_{A2},v_{B1}}(t)/dt = -1$, waiting time is given by the elapsed time between arrival and the next departure (8:30 → 9:45). Upon arriving to station v_B at 10:45, we disembark on the platform denoted by vertex v_{B2} and proceed to transfer to another train traveling along route 3, costing us $c\big(v_{B1}, v_B, \tau + c(v_{A2}, v_{B1}, \tau)\big)$ additional time.

3.5. Assembling the Multi-modal Network

The multimodal network combines all of the road, foot, bicycle, and public transit networks within a single data structure: a labeled directed graph $G(V, E, \Sigma)$. To distinguish the networks, we attach a unique *label* $\sigma \in \Sigma = \{c, f, b, rb, p\}$ to each edge, where c, f, b, rb, and p stand for *car*, *foot*, *(private) bicycle*, *rental bicycles*, and *public* respectively. Let us denote by $G_\sigma(V_\sigma, E_\sigma)$ the uni-modal graph labeled $\sigma \in \Sigma$. The vertex-set of the multi-modal graph G is given by $V = \cup_{\sigma\in\Sigma} V_\sigma$ and its edge-set $E = \cup_{\sigma\in\Sigma} E^\sigma \cup E_{\text{link}}$ where E_{link} contains the set of *link* edges allowing modal changes in G, such as taking a bus after a short walk, by linking the different uni-modal networks together.

Intuitively, any transition from one network to another should be mediated via the foot network, as *some* walking is usually required for any modal change. Depending on the network however, transitions to and from the foot network are only allowed at specific locations, and thus, we select a subset of *link* vertices $V^\sigma_{\text{link}} \subseteq V^\sigma$ from each graph for the mode σ:

Foot* ↔ *Road: The road network is accessible everywhere a car is allowed to park. Furthermore, rental vehicles are accessible at rental stations. Thus, all parking spots and rental stations are marked as link vertices $V^{road}_{\text{link}} = V^{\text{park}} \cup V^{\text{rentcar}}$.

Foot* ← *Private Bicycle: Considering that bicycles can be used almost everywhere walking is possible (except on stairways, for instance), every vertex $v \in V^{bike}$ is a link vertex from which we can access the foot network. Thus, $V^{bike}_{\text{link}} = V^{bike}$.

Foot* ↔ *Rental Bicycle: when renting a bicycle, we can 'enter' and 'exit' the bicycle mode only when entering a rental station. Thus, $V^{rentbike}_{\text{link}} = V^{rentbike}$, where $V^{rentbike}$ represents the rental stations only.

We assume here that a bicycle can be returned to any rental station. If different companies are colocated, we should force the user to return its bike to the right rental station. To support such a

feature, we would need to use a different label for each company, similarly to (Sauer, Wagner, and Zündorf 2020). The automaton would, in that case, introduce a new state denoting the company id to force to return the bicycle to the right location.

Foot ↔ Public Transit: Public transit stations are accessible via station vertices. Hence, $V_{\text{link}}^{stations} = V^{stations}$.

Then, for each vertex $v \in V_{\text{link}}^{\sigma \in \Sigma}$, we must compute the closest vertex $w \in V^{foot}$ in the foot network and add the link edges $(v,w),(w,v) \in E_{\text{link}}$. Additional labels are added to Σ as we label link edges according to the type of transfer:

- link edges (v,w), $(w,v) | v \in V^{foot} \wedge w \in V^{rentbike}$ are labeled with $lab(v,w) = b_r$ and $lab(w,v) = b_s$ which imply *renting* and *restoring* the bike respectively.
- link edges $(v,w) | v \in V^{bike}$ and $w \in V^{foot}$ are labeled with $lab(v,w) = b$. Indeed, we can only transition from the bike to the foot network (i.e., the bike is only available at the point of departure).
- link edges (v,w), $(w,v) | v \in V^{foot}$ and $w \in V^{\text{rentcar}}$ denote transfers to car-rental stations, and are labeled with the labels c_r and c_s similarly to the bike-rental case.

Relying on a brute force approach to compute link edges is costly: we have to scan the whole foot network to identify the closest foot-vertex for each vertex in the bicycle network, leading to a quadratic complexity of $\mathcal{O}(V_b \times V_f)$. A better approach relies on clustering the foot vertices using a 2d-tree (Bhatia et al. 2010) based on latitude and longitude, which is a suitable data structure for solving the *nearest neighbor* problem in logarithmic time.

It is worth noting that this preprocessing is executed once, even if the traffic congestion evolves later: multimodal links are not time-dependent. Thus, the cost of a link edge $c(v,w)$ is fixed and depends on the transfer type. It includes the required walking-time to transfer to or from the foot network and an additional cost to consider either parking-time, processing at a rental station, or for instance, the time it takes to secure a bicycle. Nonetheless, we must also ensure path *feasibility*. That is, if the private car (resp. bicycle) is left behind at some point during the trip in favor of using the bus, we would not be able to use our private car (resp. bicycle) again. Similarly, a scenario in which the private car is used after taking a train is not valid. However, the road network remains accessible via other means such as a taxi or from a rental station. Such constraints are not embedded within the graph but instead dealt with using an automaton, as detailed in the upcoming section.

We may extend our multimodal graph to handle passengers carrying their private bikes on public transport. In that case, we would accept a direct transition between the bicycle and public networks. The link vertices correspond to the subway/bus stations that accept bicycles. The label associated with the link edges are different when transitioning from bicycles to the public network, and inversely. That way, we can force in the automaton the sequence *bicycle* → *public* → *bicycle* (cf. section 4.3).

4. MUSE: The Algorithm

We now detail MUSE, a speedup technique to *Regular Language Constrained Dijkstra* (D_{RegLC}) based on graph separators to solve the multimodal shortest path problem. By dividing the multimodal graph G^{Σ} into multiple smaller regions, we can precompute an overlay graph H significantly smaller than G^{Σ} and use it to compute queries much faster.

MUSE runs in three stages:

1. **partitioning the graph:** we partition the graph G^{Σ} to split it into k cells $\{C_0, C_1, .., C_k\}$ so that we minimize the number of boundary vertices (section 4.1). Regardless of the time-dependent nature of G^{Σ}, partitioning is run only once, as it only depends on the topology of the graph.
2. **computing the overlay:** we compute for each cell a *full clique*, essentially adding a directed edge for each pair of boundary vertices (section 4.2). It is worth noting that the full clique has to

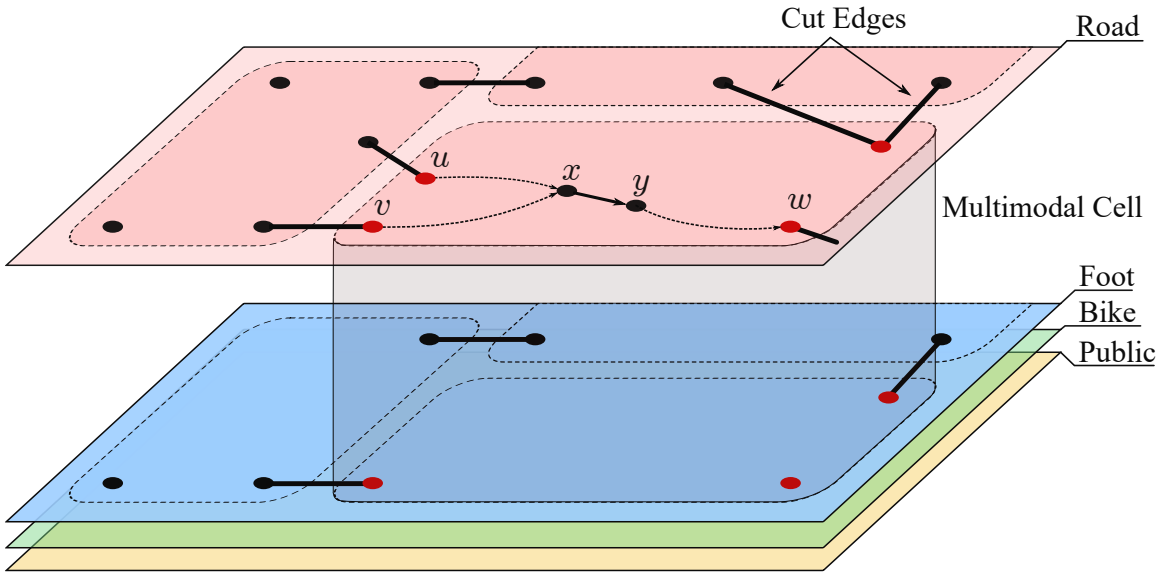


Figure 3 **Multimodal graph partition with cut edges shown in bold segments. The multimodal cell spans across all unimodal networks. The vertices u, v, and w are boundary vertices and the shortest paths P_{uw} and P_{vw} share the subpath $\{x, y, ..w\}$.**

be recomputed when the weights change. The overlay graph consists of all cliques and *cut edges*, i.e., edges connecting two different cells. The cliques are constrained by an automaton that defines the sequences of modes allowed by the user. Furthermore, considering that the graph contains time-dependent edges, the weight of clique edges can also, possibly, be time-dependent. Therefore, we run *profile queries* to compute the cost function of each clique edge. Even though profile queries are costly, cliques are independent, and the computations are parallelized, allowing to compute the whole overlay H for large instances in a few minutes only.

3. **solving queries:** we compute the shortest path $P = q(r, t, \tau)$ by running D_{RegLC} on the query graph $G^q = G_r \cup H \cup G_t$ which consists of the subgraphs G_r and G_t, induced by the cells containing the source and target vertices r and t, respectively and the overlay H. We can improve query times with a goal-directed version of the algorithm called MUSE*. Better speed-ups are possible using heuristics (MUSE^{SV}) with a tradeoff on correctness (section 4.3).

4.1. Stage 1: Partitioning The Graph

Planar graphs (i.e., graphs that can be drawn on a flat surface without any intersecting edges) belong to a class of sparse graphs with valuable topological properties. Most importantly, planar graphs can be partitioned in linear time with small separators (Djidjev 1982).

Given an undirected graph $G(V, E)$, a partition on G is a collection $\{C_1, C_2, .., C_k\}$ where each element $C_i \subseteq V$ is referred to as a *cell*. Partitioning breaks apart the vertex-set V into k cells $V = \cup_{i=1}^{k} C_i$ with no overlap $C_i \cap C_j = \varnothing | i \neq j$. Most importantly, the goal is to compute a partition such that the number of boundary vertices is minimized. An edge (v, w) is a *cut edge* if both v and w are boundary vertices belonging to two different cells.

Although not planar (due to overpasses and tunnels), road networks can also be efficiently partitioned (Eppstein and Goodrich 2008, Sanders and Schulz 2012). In a multimodal graph, the vast majority of the vertices belong to the road network. In fact, most foot and bicycle vertices are duplicates of road vertices (think of sidewalks along a road segment, for instance); therefore, partitioning remains viable.

Computing ideal partitions is, however, NP-hard (Garey and Johnson 2002); thus, we recourse to approximations that have exhibited good results for transportation networks. METIS (Karypis

and Kumar 1998) is a Multilevel Graph Partitioning algorithm that runs in three stages. The first, *coarsening* stage, aims to reduce the graph's size by repeatedly merging neighboring vertices and collapsing edges. Each iteration i, produces a coarser graph G_i than its predecessor G_{i-1}. The second stage, *partitioning*, is run on the coarsest graph. Finally, during the *uncoarsening and refining* stage, the graph is expanded at each iteration i, and the partition is refined until the graph reaches its initial size.

Setup: our objective is to minimize the overall number of boundary vertices, regardless of the number of incoming and outgoing cut edges at each cell. Therefore, we transform the directed labeled graph G^{Σ} into an undirected graph $G(V, E)$ without labels. Furthermore, considering that the public transit network is modeled as a time-dependent graph, the partitioning process might break apart the platform vertices of a single station across multiple cells. This would mean that the different platforms of a station would all be border vertices with many transfer edges, which would be suboptimal. To speed-up the partitioning, we contract all the platforms ($V^{platforms}$) attached to the same station into a single vertex $v_s \in V^{station}$, corresponding to a station. We also keep an edge $(v_s, v_{s'}) \in E$ if there exists at least one connection edge between any substation of v_s and any substation of v'_s (*i.e.*, the two stations are connected by at least one route).
Coarsening is an iterative process where pairs of vertices are contracted together to reduce the size of the graph. At each iteration i we obtain a coarser graph $G_i(V_i, E_i)$, so that $|V_i| < |V_{i-1}|$. Initially, $\forall v \in G$ we attribute a size label $s(v) = 1$. Contracting two vertices v, w means replacing them with a new vertex x such that $s(x) = s(v) + s(w)$. To contract multiple vertices simultaneously, a set of edges $M \subset E_i$ is selected, such that no two edges share the same vertex, also known as a *matching*. Thus, at each iteration, we compute the maximal matching, i.e., a matching such that no additional edge can be added, and contract all vertices $v, w | (v, w) \in M$. Coarsening is halted when the size of the graph is sufficiently small. Most importantly, we must ensure that $|V_i| \geq k$ where k represents the number of desired cells in the partition.
Partitioning: the coarsest graph $G_c(V_c, E_c)$ can be partitioned using Breadth-First-Search (BFS) starting from a random vertex $v \in V_c$ and growing a tree $T \subset V_c$ until $|T| \simeq 1/2|V_c|$. To obtain k partitions, the initial partitions are then recursively partitioned $log_2(k)$ times. Kernighan and Lin (Kernighan and Lin 1970) use a heuristic to polish the partitioning by exchanging vertices between cells producing a smaller cut-size.
Uncoarsening and Refining: at each iteration i, a less coarse graph G_i is obtained by expanding G_{i-1}. For all vertices $x \in V_{i-1}$, we expand x to obtain vertices $v, w \in V_i$, and assign them to the same cell x belonged to. Kernighan Lin's heuristic is then run again to refine the partition. After the last iteration, we expand the station vertices to retrieve the public transit network and transform the undirected partitioned graph G back to the multimodal graph G^{Σ}.

We obtain a partitioned multimodal graph, as illustrated in figure 3, where each layer corresponds to a specific type of transportation. The cells of the partition are, therefore, multimodal cells with boundary vertices located in different layers.

4.2. Stage 2: Computing The Overlay

The partition produces a set of cells, where each cell C_i contains a set of boundary vertices $V_i^b \subset C_i$. The overlay graph $H(V^H, E^H)$ of G^{Σ} has the vertex-set $V^H = \cup_{i=1}^{k} V_i^b$ which consists of all boundary vertices. The goal is to compute for each pair of boundary vertices $v, w \in V_i^b$ belonging to the same cell C_i, a shortcut edge (v, w) denoting the shortest path P_{vw}.

Computing all shortcuts within a cell produces a *clique*. It is worth noting that MUSE is naturally parallelizable. Indeed, each cell is independent, and the computation of the shortest paths for each clique can be executed on a different processor/core. After computing all cliques, the edge-set of the overlay $E^H = E_{cut} \cup_{i=1}^{k} (v, w) \mid v, w \in V_i^b$ consists of all edge cuts E_{cut} together with the clique edges. This explains why we minimize the number of boundary vertices when partitioning the graph. Computing a clique edge (v, w) requires, however, solving the two following problems:

- The shortest path P_{vw} between v and w is multimodal. Therefore, we must solve the Label Constrained Shortest Path Problem (Barrett, Jacob, and Marathe 2000) (LCSPP) to restrict the modal sequence of P_{vw}.
- Edges can be time-dependent; hence, the cost of P_{vw} varies based on departure time. Therefore, we must compute the *profile* of (v, w) denoting its associated cost for all departure times.

We solve both problems simultaneously using a label correcting algorithm, which produces a Constrained Profile Clique (CPC) for each cell in the partition. We first detail each problem separately, then delve into the details of the algorithm.

Solving the LCSPP: Using formal languages, we can define modal constraints using *regular expressions* to prune prohibited edge transitions while computing clique edges. A regular expression defines a rule for combining a set of letters of an alphabet using the *or*, *and*, and *kleene* operators. Moreover, any regular expression can be written in the form of a Deterministic finite-state automaton (DFA) (Brüggemann-Klein 1993). Formally, an DFA is a 5-tuple $\mathcal{A} = \{S, \Sigma, \delta, s_0, F\}$ which consists of a finite number of states S, an alphabet Σ, a transition function $\delta : S \times \Sigma \to 2^S$, an initial state s_0, and a set of accepting states $F \subseteq S$. Therefore, given a directed labeled graph G^Σ and an DFA $\mathcal{A}$, MUSE solves the LCSPP by computing a path in G^Σ such that (i) its cost is minimum and (ii) the *word* obtained by concatenating its edge labels (sequence of modes) is accepted by $\mathcal{A}$ (*i.e.*, it corresponds to an acceptable sequence of modes).

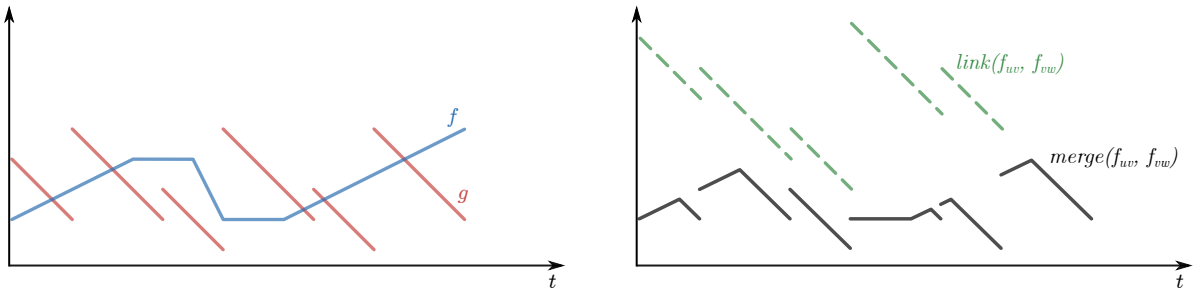
Conveniently, $\mathcal{A}$ can be implemented as a directed labeled graph $G^\mathcal{A}$. Thus, we can solve the LCSPP with D_{RegLC} (Barrett, Jacob, and Marathe 2000), a regular language constrained Dijkstra algorithm, deployed on the *product graph* $G(V^\times, E^\times)$ where $V^\times$ are vertices and $E^\times$ edges. The product graph $G^\times = G^\Sigma \times G^\mathcal{A}$ is a composition of the underlying graph G^Σ and the automaton graph $G^\mathcal{A}$. A product vertex $\langle v, s\rangle \in V^\times$ is a pair of a vertex $v \in V(G^\Sigma)$ and a state $s \in S(\mathcal{A})$. A product edge $(\langle v, s_i\rangle, \langle w, s_j\rangle) \in E^\times$ is added *iff* there exists an edge $(v, w) \in E(G^\Sigma)$ such that $s_i \times lab(v, w) \to s_j$ is a valid transition of $\delta \in A$.

Computing a shortest path profile (D_p): In this section we use the term *distance label* of a vertex v, written $l(v)$, to designate the tentative cost to reach v from a source vertex r (not to be confused with multimodal labels). In a unimodal time-dependent graph G, we call D_p a *label correcting* version of Dijkstra's algorithm (Orda and Rom 1990) allowing to compute the *profile* function $d_*(r, t)$ denoting the minimum travel time of the shortest path P_{rt} for all departure times τ. The algorithm does not necessarily settle a vertex v after it is extracted from the queue because of time-dependency. Rather, it propagates its profile $d_*(r, v)$ to its neighbors and potentially, re-inserts v into the queue (label correction) if $d_*(r, v)$ is improved for some departure time τ.

In that context, performing edge relaxations requires additional operations to manipulate functions rather than scalars. Therefore, we define the following operations:

- **evaluate:** given a piece-wise linear function f and a time τ, evaluation returns $f(\tau)$ in $\mathcal{O}(log|f|)$.
- **link:** given two edges (v, w) and (w, x) with cost functions f and g, respectively, linking returns the cost function $f * g = f + g \circ (f + \tau)$ of the path (v, w, x) with complexity $\mathcal{O}(|f| + |g|)$. The operator $\circ$ designates the composition of two functions.
- **merge:** given two parallel edges (v, w) and $(v, w)'$, with cost functions f and g, respectively, merging returns the cost function $h = min(f, g)$, with minimal travel time from u to v for any time τ. That is, $merge(f, g) = min\{f(\tau), g(\tau) \mid \forall \tau \in \Pi\}$ with complexity $\mathcal{O}(|f| + |g|)$.

Performing the merge operation, in particular, is computationally expensive because we have to compute all segment intersections of the input functions f and g. We use a line sweep algorithm (Bentley and Ottmann 1979) over the breakpoints of f and g. Computing intersections is a requirement because the merged functions are not necessarily homogeneous; that is, the slope of the piecewise linear functions f and g can be different, rendering the merge operation non-trivial. Figure 4 illustrates the outcome of the link and merge operations. In this example, we are merging (and linking) the cost function f of a road network edge with the cost function g of a public transit



(a) cost functions f_{uv} of a road network edge, and f_{vw} of a public transit edge.

(b) the resulting functions after linking or merging f_{uv} and f_{vw}.

Figure 4 Illustrating the outcome of the linking and merging non-homogeneous edge cost functions f and g.

edge. As illustrated, the merged function in figure 4b contains breakpoints belonging to either f or g (for instance the first and last breakpoints) but also breakpoints corresponding to the segment-intersections of f and g. Furthermore, the resulting function has more breakpoints than either f or g, and thus, memory requirement becomes a concern after successive merge-operations.

Nonetheless, merging can be avoided if one of the functions dominates the other; that is, if $\underline{f} \geq \overline{g}$, then g is kept as the result of the merge. Similarly, explicit linking can be avoided if either function is constant. If both f and g are constant, linking is a trivial addition. If only one function is constant, we distinguish two cases: if f is constant, we simply translate all g segments upwards by the constant. If otherwise, g is constant, we additionally translate all f segments left by the constant. Only if f and g are non-constant functions, we scan their breakpoints and perform the link operation.

Algorithm 1 details the steps of D_p, which is very analogous to the classic label setting Dijkstra algorithm with a few adaptations to handle vertex labels which are piecewise linear functions. At each iteration, the vertex v with the smallest key $key(v) = \underline{d_*}(r, v)$ (*i.e.*, it has the highest priority) is extracted from the queue (line 4) and all of its neighboring vertices are scanned (line 7). For each outgoing edge (v, w), we compute the distance label $l(w) = d_*(r, v) \ * \ f_{vw}$ (link operation) and check if it improves $d_*(r, w)$ for some departure time. In that case, we update the distance label, with the merge operation: $d_*(r, w) = merge(d_*(r, w), l(w))$ and compute $key(w) = \underline{d_*}(r, w)$ (line 8-10). If w is scanned for the first time, we insert it into the queue; otherwise, we just update its key inside the queue (11-15). This whole process is repeated until a vertex v is extracted from the queue such that $\underline{d_*}(r, v) > \overline{d_*}(r, t)$ holds (line 5). Therefore, the target's distance label cannot be improved anymore, the algorithm is stopped, and $d_*(r, t)$ is correct. Similarly, for a set of targets T, the same condition must hold for each $t \in T$ for the algorithm to stop.

Constrained Profile Clique Algorithm (lcD_{pRegLC}): Given a cell C_i, we have to compute the shortest path profiles from each boundary vertex $v \in V_i^b$ toward all other boundary vertices. Since G^Σ is a labeled graph, we must also ensure that all shortest paths abide by the regular language of automaton $\mathcal{A}$. Let us call D_{pRegLC} the regular language constrained version of D_p, which computes profiles of constrained shortest paths. Hence, one approach is to run D_{pRegLC}, $|V_i^b| \times |S|$ times, from each product vertex $\langle v, s \rangle$ where $v \in V_i^b$ and $s \in S(\mathcal{A})$. In other words, the algorithm would compute a path from any boundary vertex: the path is possibly different for any state of the DFA. However, this approach is wasteful in terms of memory usage since all pairs of vertex v and state s of the DFA are always possible.

Consider $\mathcal{A}_5$, the automaton shown in figure 5e. A vertex v whose label $lab(v) = c$ (corresponds to the road network) is not compliant with the state s_2 that corresponds to the public transit network. Besides, this state is only reachable from either itself or s_1 (the foot network). Therefore, we design the automaton such that each state is attributed a unique label $\{f, c, b, p\}$ designating a specific type of network. Hence, the product vertex $\langle v, s \rangle$ is *valid* only if $lab(v) = lab(s)$. This way, we obtain a smaller set $V_i^{\times b}$ of boundary vertices in the product graph.

Algorithm 1: D_p (Profile Dijkstra)

Input : graph $G(V,E)$, source vertex r, set of target vertices T
Output: profile function $d_*(r,t)$ for all $t \in T$

1 $d_*(r,v) \equiv \infty \mid \forall v \in V$ and $scanned(v) =$ false;
2 $d_*(r,r) \equiv 0$, set $key(r) = 0$ and insert r into priority queue Q;
3 **while** Q *not empty* **do**
4 extract vertex v with smallest key from Q;
5 **if** $\underline{d_*}(r,v) > \overline{d_*}(r,t) \forall t \in T$ **then** // stop when v cannot improve any target $t \in T$
6 **return**
7 **foreach** *outgoing edge* (v,w) **do**
8 **if** $d_*(r,v) * f_{vw} < d_*(r,w)$ **then** // (u,w) yields an improvement for some departure τ
9 $d_*(r,w) = merge(d_*(r,w),\ d_*(r,v) * f_{vw})$; // update profile of w
10 $key(w) = \underline{d_*}(r,v)$; // set key to global minimum of the profile function
11 **if** *not scanned*(w) **then**
12 insert w into Q;
13 $scanned(w) = true$;
14 **else**
15 update $key(w)$ inside Q;
16 **end**
17 **end**

Moreover, instead of running D_{pRegLC} a number of $|V_i^{\times b}|$ times (once for each valid product boundary vertex), we are better off using a *multiple source* approach to compute the whole clique in a single run. To understand the appeal of using a multiple-source approach over multiple calls to the single-source shortest path algorithm, consider the diagram of figure 3. Vertices u, v, and w are boundary vertices belonging to the same cell and the edge (x,y) is common to both shortest paths P_{uw} and P_{vw}. With a multiple source approach, each vertex is attributed one distance label for each source vertex. Hence, when scanning vertex y, we can update both $d_*(u,y)$ and $d_*(v,y)$ at once. This approach is favored when the set of source vertices are close to one another, increasing the likelihood of overlapping shortest paths. Thereby, particularly effective to compute our constrained cliques considering that boundary vertices are tightly packed in practice, outlining the boundary of the cell they belong to, as illustrated in figure 6b.

We call our multiple-source algorithm lcD_{pRegLC}, which stands for *label-correcting Regular Language Constrained profile Dijkstra*. As shown in Algorithm 2, lcD_{pRegLC} takes as input the multimodal graph G^Σ, the automaton $\mathcal{A}$ and a set of source vertices R. Since we are manipulating product vertices, the distance label of vertex $\langle v, s' \rangle$ with respect to a source vertex $\langle r, s \rangle$ is written $d_*^{r,s}(v,s')$.

The first block (line 1-5) initializes, for each source vertex $\langle r, s \rangle$, its own distance label to $d_*^{r,s}(r,s) \equiv 0$ and that of all other vertices $d_*^{r,s}(v,s') \equiv \infty$ (with respect to the source $\langle r, s \rangle$). Then, all source vertices are added to the queue. At each iteration, the vertex $\langle v, s' \rangle$ with the smallest key is extracted from the queue (line 7). Then, for each outgoing product edge $(\langle v, s' \rangle, \langle w, s'' \rangle)$ (line 8-9), we check if the distance label of $\langle w, s'' \rangle$ can be improved with respect to each source vertex $\langle r, s \rangle$. That is, if $d_*^{r,s}(v,s') * f_{vw} < d_*^{r,s}(w,s'')$ holds for any departure time τ, then the edge is relaxed (line 13). In that case, we update the key $key(w,s'')$ to the smallest $\underline{d_*^{r,s}}(w,s'')$ among all source vertices $\langle r, s \rangle$ and add $\langle w, s'' \rangle$ to the queue (line 16-19).

To compute the clique of a cell C_i, we define the set of source vertices as the boundary vertices $R = V_i^b$. We also define a set of targets $T = V_i^b$ and stop the algorithm when the next extracted vertex $\langle v, s' \rangle$ respects rule $key(v,s') > \overline{d_*^{r,s}}(t,s'')$ for all targets $\langle t, s'' \rangle \in T \times S$. This means that we cannot find a shorter path for the most distant boundary vertex.

Algorithm 2: lcD_{pRegLC} (label correcting D_{pRegLC})

Input : Graph $G(V, E, \Sigma)$, Automaton $\mathcal{A}(S, \Sigma, \delta, s_0, F)$, and Source set $R \subseteq V$
Output: Constrained shortest path profiles $d_*^{r,s}(v, s') \mid \forall \langle r, s\rangle \in R \times S$ and $\forall \langle v, s'\rangle \in V \times S$

foreach *source* $\langle r, s\rangle \in R \times S$ **do**
  $d_*^{r,s}(r, s) \equiv 0$; // set distance label of $\langle r, s\rangle$ to 0
  $d_*^{r,s}(v, s') \equiv \infty \mid \forall \langle v, s'\rangle \neq \langle r, s\rangle$; // set distance label of all other vertices to infinity
  set $key(r, s) = 0$ and add $\langle r, s\rangle$ to priority queue Q;
end
while Q *not empty* **do**
  extract product vertex $\langle v, s'\rangle$ with smallest key from Q;
  foreach *outgoing edge* (v, w) **do**
    foreach *transition* $s' \times lab(v, w) \to s''$ **do**
      $updated = false$;
      foreach *source* $\langle r, s\rangle$ **do** // compute distance labels of $\langle w, s''\rangle$ for each source $\langle r, s\rangle$
        if $d_*^{r,s}(v, s') * f_{vw} < d_*^{r,s}(w, s'')$ **then** // relax product edge $(\langle v, s'\rangle, \langle w, s''\rangle)$
          $d_*^{r,s}(w, s'') = merge(d_*^{r,s}(w, s''),\ d_*^{r,s}(v, s') * f_{vw})$;
          $updated = true$;
      end
      if *updated* **then**
        // set key to the global minimum among all distance labels of $\langle w, s''\rangle$
        $key(w, s'') = min\{\underline{d_*^{r,s}}(w, s'') \mid \forall \langle r, s\rangle \in R \times S\}$;
        add $\langle w, s'\rangle$ to Q if not in Q, otherwise update its key;
    end
  end
end

4.3. Stage 3: Solving Queries

During the online phase, MUSE solves a query $q(r, t, \tau)$ by running D_{RegLC} on the query graph $G^q = G_r \cup H \cup G_t$ which consists of the overlay H (precomputed in the previous stage) and the subgraphs G_r and G_t, induced by the cells containing the source and target vertices r and t, respectively. As G^q is significantly smaller than the original multimodal graph, MUSE computes the shortest path $P_{rt}(\tau)$ fast enough for interactive queries. Algorithm 3 details the flow of execution.

Nonetheless, knowing the target's location allows for making informed decisions while searching for the shortest path. Instead of "blindly" expanding from the source, the intuition behind goal direction is to use heuristics to guide the search by prioritizing vertices closer to the target. We call MUSE* (pronounced 'MUSE-star') the goal directed version of the algorithm (analogous to $A^\star$ (Hart, Nilsson, and Raphael 1968)) which uses Euclidean distances to compute the *potential function*. The potential function $\pi(v)$ of a vertex v represents an estimate of the remaining cost to reach the target vertex t. To guarantee correctness however, the potential function must be admissible, which means that it must underestimates the true cost, that is, $\pi(v) \leq cost(P_{vt})$ where P_{vt} denotes the shortest path between v and t.

Another variant called MUSE^{SV} applies the same strategy but relies on the *Sedgewick-Vitter* heuristic (Sedgewick and Vitter 1986), which trades off correctness for the sake of speed. The potential function is evaluated as $\pi(v) = \alpha \times d_{Euc}(v, t)\ /\ Speed_{max}$ where $d_{Euc}(v, t)$ denotes the Euclidean distance between v and the target t, $Speed_{max}$ the highest speed in the network and α a tuning parameter. Setting $\alpha > 1$ may potentially overestimate the shortest path's cost, which impacts correctness but results in a smaller search space overall.

Algorithm 3: MUSE (query algorithm)

Input : graph $G(V,E,\Sigma)$, partition $\{C_1,C_2,..,C_k\}$, overlay H, DFA $\mathcal{A}=\{S,\Sigma,\delta,s_0,F\}$, source $r\in V$, target $t\in V$, departure time τ
Output: cost of shortest path $P_{rt}(\tau)$

Let $G^q(V^q,E^q)=C_r\cup H\cup C_t$; // G^q is the query graph and C_r and C_t are the source and target subgraphs induced by their respective cells
$d(v,s)=\infty \mid \forall\langle v,s\rangle\in V^q\times S$; // set distance label of all vertices to infinity
$d(r,s_0)=0$; // set distance label of source vertex to 0
add source $\langle r,s_0\rangle$ to priority queue Q
while Q *not empty* **do**
  extract product vertex $\langle v,s\rangle$ with smallest key $d(v,s)$ from Q;
  if $v=t$ *and* $s\in F(\mathcal{A})$ **then**
    return $d(v,s)$; // v is the target and s is a final state
  foreach *outgoing edge* $(v,w)\in E^q$ **do**
    foreach *transition* $s\times lab(v,w)\to s'$ **do**
      if $d(v,s)+f_{vw}(\tau+d(v,s))<d(w,s')$ **then**
        $d(w,s')=d(v,s)+f_{vw}(\tau+d(v,s))$; // relax the product edge $(\langle v,s\rangle,\langle w,s'\rangle)$
        $key(w,s')=d(w,s')+\pi(w)$;
        add $\langle w,s'\rangle$ to Q if not in Q, otherwise update its key;
    end
  end
end

It is worth noting that different modes may be used inside the same cell. Indeed, Algorithm 3 does not restrict the number of time a specific re-enters a specific cell. Only the Finite State Automaton constrains the exploration and will possibly prevent re-using the same mode several times.

4.4. Handling time-dependent weights and unpredictable evolutions

In multimodal transportation, we cannot assume weights are static: the user decides according to the current network conditions. However, dynamic weights do not have the same meaning and are not handled similarly for all transportation modes.

For **Public Transit Networks**, the weight of each edge is a non-continuous function: the travel time increases discretely after a station has been deserved, including the waiting time before the next vehicle's arrival. Thus, merging two paths corresponds to merging the corresponding end-to-end cost functions (section 4.2). These linking and merging operations are expensive, but the weights are predictive and known a priori.

For **Foot and Bike Networks**, we can safely neglect the variations. Thus, these modes don't impact significantly the computation time.

On the contrary, **Road Networks** also have time-dependent weights. However, these travel times for each road segment are highly complex to estimate accurately (Yang, Vereshchaka, and Dong 2018). Thus, we cannot update the routes with the linking - merging functions. We must rather recompute the routes for the road network, taking into account the new weights.

Typically, UCCH (Dibbelt, Pajor, and Wagner 2015) considers each mode independently. Thus, only the road network mode has to be reprocessed. However, the routes have to be recomputed as soon as one of the routes becomes sub-optimal. MUSE has been rather designed to cope efficiently with this kind of situation: each cell is independent, and the routes have to be recomputed only for the cells which undergo a change.

Table 2 Graph characteristics. For the road and bicycle networks, the stations column designates the number of rental stations.

Transportation Network	Vertices		edges		stations	
	G_{idf}	G_{fr}	G_{idf}	G_{fr}	G_{idf}	G_{fr}
Foot (G_f)	519,558	8,048,695	1,363,995	20,157,922	-	-
Road (G_r)	457,406	7,252,489	1,018,814	16,712,036	249	899
Bicycle (G_b)	406,711	7,354,519	1,000,591	18,292,964	800	3,324
Public Transit (G_p)	60,959	255,872	351,551	1,089,911	18,836	251,685
Foot-Road edges (E_r^{link})	-	-	44,270	431,470	-	-
Foot-Bicycle edges (E_b^{link})	-	-	813,422	14,665,980	-	-
Foot-Public edges (E_p^{link})	-	-	37,672	192,886	-	-
Multimodal (G^{Σ})	1,444,634	22,911,575	4,630,315	71,543,169	-	-

5. Experimental Evaluation

We report in this section the performance evaluation of MUSE and its associated heuristic-based variants, namely MUSE* and MUSE^{SV}. Furthermore, we evaluate State-Dependent ALT (SDALT) (Kirchler et al. 2011) on the same dataset to compare both algorithms. SDALT is a landmark-based approach applied to multimodal networks. Its preprocessing consists of computing all shortest paths in the network from and to a set of landmarks vertices, as described in section 2.3.3.

Our experimental setup is fully available:

1. a tool to transform OSM data into a directed graph with adjacency lists (Falek 2019);
2. our dataset (GTFS, map, NFA, graph overlay) is available online (Falek 2021a). In particular, it can be reused to execute other algorithms in the same conditions;
3. our implementation of our algorithms (Falek 2021b).

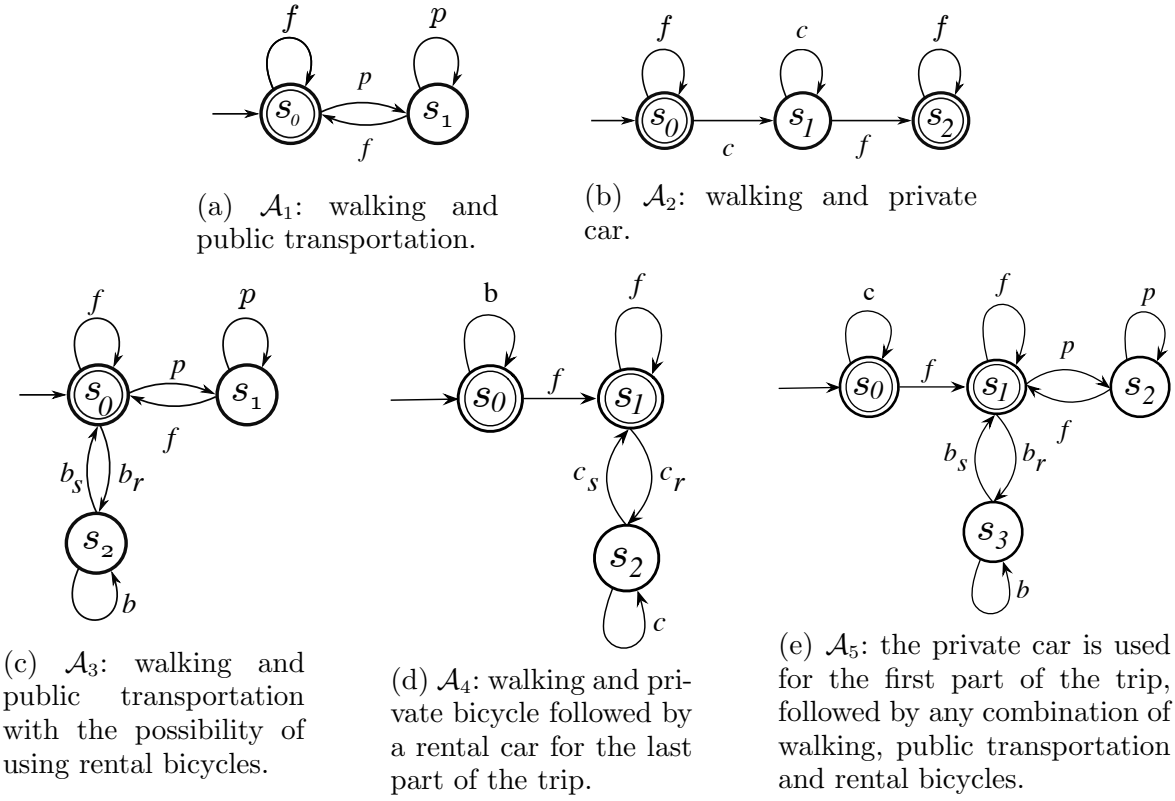
5.1. Evaluation Setup

We run all algorithms on two graph instances to assess scalability: the Ile-de-France region denoted G_{idf} and a country-size graph G_{fr} representing France. Table 2 summarizes the graph characteristics based on each transportation layer. The graphs were modeled by combining:

1. a topological dataset from **OpenstreetMap** (Mooney, Minghini et al. 2017), an open-access dataset built through the effort of crowd-sensing and accessible via the GeoFabric (Geofabrik 2019) online platform. We use it to construct the road, cycling, and pedestrian graphs. It consists of latitude and longitude coordinates denoting roads, parking spots, and even bicycle and car rental service stations.
2. a *General Transit Feed Specification* (**GTFS**) dataset that represents the standard format used to encode public transit schedule information. For the Ile-de-France graph instance, we rely on the *Ile-de-France mobilités* (IdFm) dataset (Ile de France Mobilités 2020), an online platform providing up-to-date data in GTFS format combining train, RER, subway, tramways, and bus networks in the region. For France's public transit network, we combine four GTFS datasets covering the whole country and obtained via the open-source platform Navitia (Navitia Open Data 2020).

To model common use case multimodal trips, we define several automata (Fig. 5):

$\mathcal{A}_1$ depicts the combination of walking and all types of public transportation.
$\mathcal{A}_2$ depicts trips that rely on the private car only, for the whole trip;
$\mathcal{A}_3$ extends $\mathcal{A}_1$ with faster transfers, using rental bicycles, mostly available in the city center at specific locations. An additional state s_2 is reachable via link edges labeled b_r and b_s to either rent or return the rental bicycle before pursuing the trip;



(a) $\mathcal{A}_1$: walking and public transportation.

(b) $\mathcal{A}_2$: walking and private car.

(c) $\mathcal{A}_3$: walking and public transportation with the possibility of using rental bicycles.

(d) $\mathcal{A}_4$: walking and private bicycle followed by a rental car for the last part of the trip.

(e) $\mathcal{A}_5$: the private car is used for the first part of the trip, followed by any combination of walking, public transportation and rental bicycles.

Figure 5 Set of Automata depicting four scenarios used to constrain preprocessing and queries.

$\mathcal{A}_4$ excludes the public transit network. It allows using the private bicycle initially followed by a rental car for the remaining part of the trip;
$\mathcal{A}_5$ combines all means of transportation. The private car is used only initially, followed by any combination of walking, public transportation, and rental bicycles.

We ran all our experiments on the High-Performance Computing (HPC) of the University of Strasbourg. We generated a unique *job* for each partition and automaton that was executed on an Intel Haswell node totaling 24 cores and 32 GB of RAM.

5.2. Preprocessing

5.2.1. MUSE: Graph partitioning results are reported in table 3. We tested various partitions ranging from 100 - 500 cells for G_{idf} and 800 - 6,000 cells for G_{fr}. The partition size, i.e., the number of cells, impacts both preprocessing and query times. Increasing the number of cells yields smaller cells with fewer border vertices but produces a larger overlay graph, which slows the queries. In contrast, a smaller partitioning results in large cells, which increases the preprocessing times. Consequently, selecting the *adequate* partition is a tradeoff between preprocessing and query times.

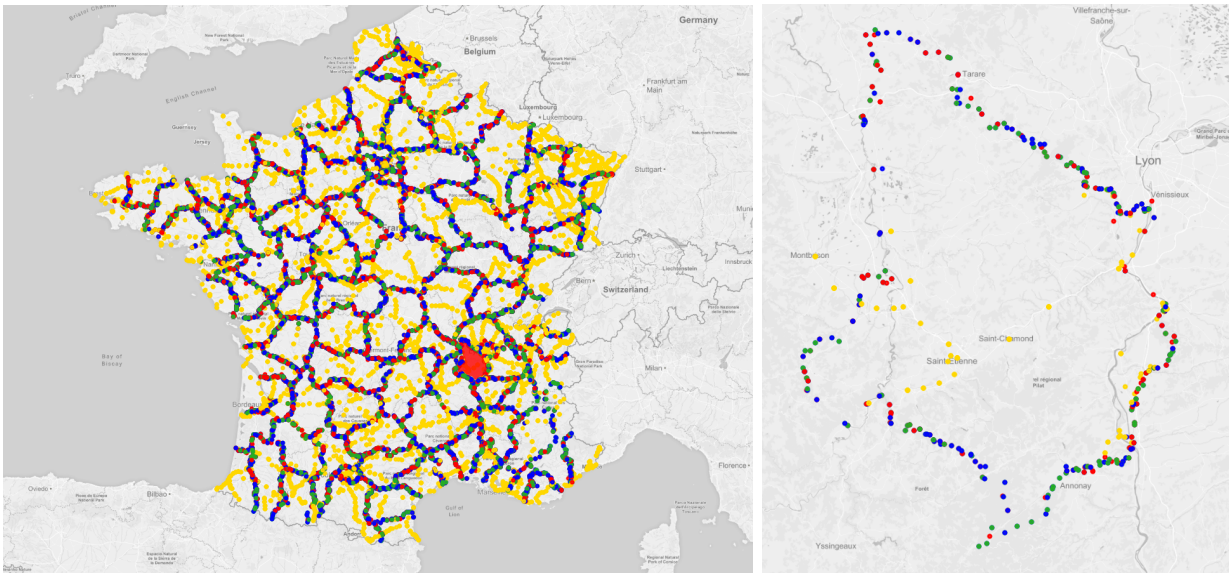
Once the graph is partitioned, the cells are independent of each other, and therefore preprocessing benefits from parallelism. For instance, figure 6a depicts a partitioning of the G_{fr} graph into 100 cells. As illustrated, the number of nodes of each cell varies according to the density of the graph. Given a cell with N border product vertices, we evaluate two strategies to compute its clique:

One-to-Many Strategy: runs a profile regular language constrained Dijkstra D_{pRegLC} algorithm from each border product vertex. The algorithm is run N times to compute the clique.

Many-to-Many Strategy: runs lcD_{pRegLC} only once. It simultaneously computes all the shortest path profiles of the clique in a single call.

Table 3 **Computational time for partitioning the graph with different number of cells and the associated number of border vertices.**

Graph	Number of cells	# Border vertices				Time [s]
		min	med	max	tot	
G_{idf}	100	124	239	428	24,518	1.023
	200	65	180	344	36,975	1.278
	300	52	159	407	49,102	1.474
	400	56	134	263	55,303	1.673
	500	29	122	308	63,081	1.874
G_{fr}	800	13	91	317	94,897	16.471
	1,000	13	91	317	94,897	18.351
	2,000	3	67	331	141,104	19.742
	4,000	4	47	191	194,352	24.598
	6,000	2	38	183	239,639	33.366



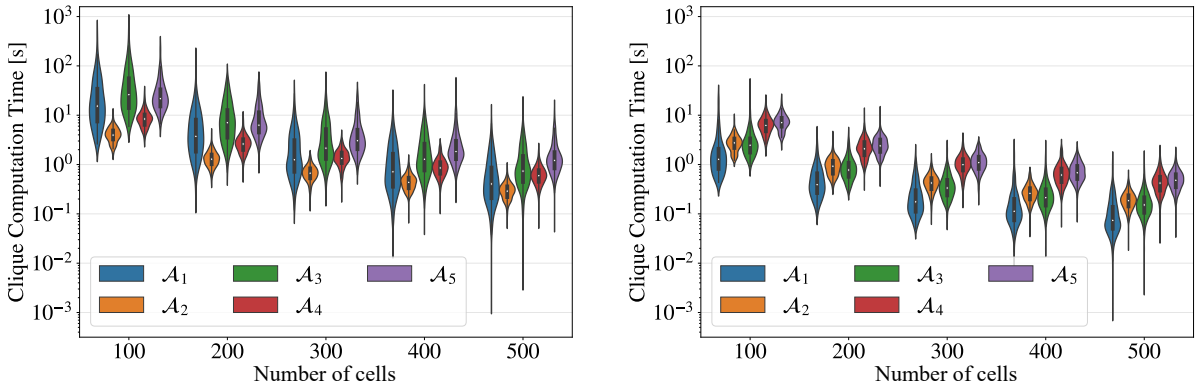
(a) France multimodal network partition (b) boundary vertices of a single cell

Figure 6 **A multimodal partition of France separating the underlying graph into 100 cells. (a) Boundary vertices belonging to the foot, bicycle, road and public transit networks are displayed in blue, green, red, and yellow dots, respectively. (b) boundary vertices of the area highlighted in red in figure (a)**

Given that the graph's density is not uniform, preprocessing time varies significantly (few milliseconds to seconds) from one cell to another within the same partition. Using the one-to-many strategy, we make the computation in parallel for each border vertex, *i.e.*, we do not need to allocate all the border vertices of a cell to the same CPU. However, the one-to-many strategy does not leverage the fact that the shortest paths of a clique are likely to share common subpaths. Therefore, vertices that are shared by several shortest paths are processed several times.

In contrast, the many-to-many strategy is more efficient as it simultaneously processes all the clique's shortest paths. However, it prohibits load balancing because it allocates a whole cell to a single CPU. In terms of memory, the largest overlays we obtained are 289 MB and 3062 MB for G_{idf} and G_{fr} respectively.

Figure 7 depicts the clique computation time distribution in the Ile-de-France region using the one-to-many (left) and many-to-many (right) strategies for each partition and automaton. As



(a) one-to-many algorithm (D_{pRegLC}) (b) many-to-many algorithm (lcD_{pRegLC})

Figure 7 **Computational time for the profile cliques of G_{idf} as a function of number of cells and automaton.**

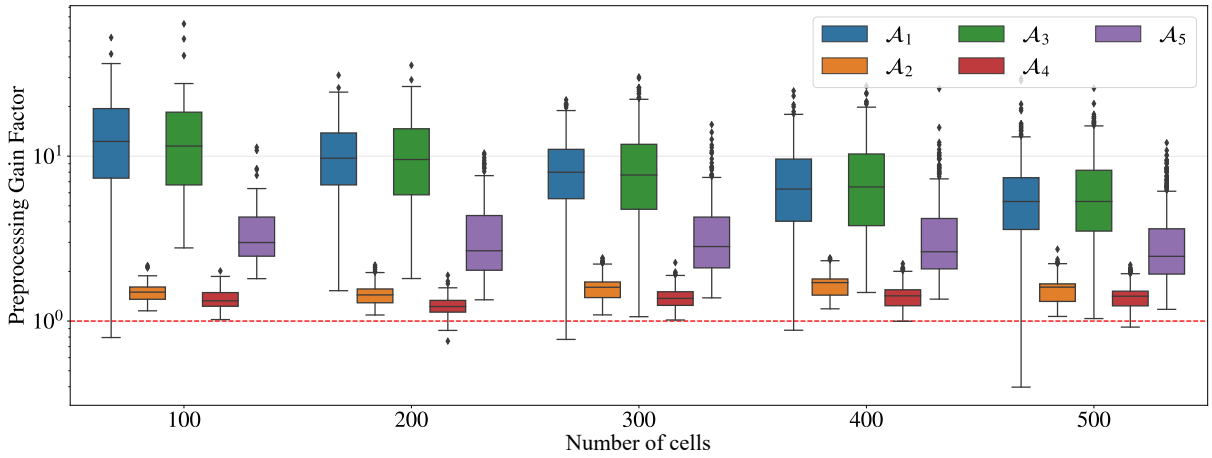


Figure 8 **The preprocessing Gain Factor denoting the speedup of the many-to-many (lcD_{pRegLC}) over the one-to-many (D_{pRegLC}) strategy for each automaton and number of cells. The horizontal red line corresponds to a gain of 1 and imply no speedup.**

expected, preprocessing time decreases when the number of cells increases. Furthermore, larger automata require more processing time (for instance, $\mathcal{A}_5$ compared to $\mathcal{A}_1$) because it increases the size of the product graph in addition to computing one shortest path for each available state. An interesting result is that lcD_{pRegLC} is an order of magnitude faster compared to running D_{pRegLC} multiple times. This confirms our previous hypothesis and shows that within a cell, there is indeed a significant number of shortest paths of the clique that do share common subpaths.

For a clearer comparison, we compute the *Preprocessing Gain Factor* (PGF) for each clique, denoting the ratio of computation time using the one-to-many strategy to that of using the many-to-many strategy. The results are shown in figure 8 and suggest that the gain depends mainly on the type of automaton used to constrain the preprocessing. The gain is smaller on automata that rely, partially, on using the road network ($\mathcal{A}_5$) and smallest for automata that exclude the public transit network ($\mathcal{A}_2$ and $\mathcal{A}_4$). In our setup, the road and public transit networks are the dominating means of transportation; that is, if all means are allowed, the shortest path between two locations is comprised, mostly, of the road network and public transit edges (fastest alternatives). Furthermore, since the road network is much denser and larger than the public transit network, the shortest

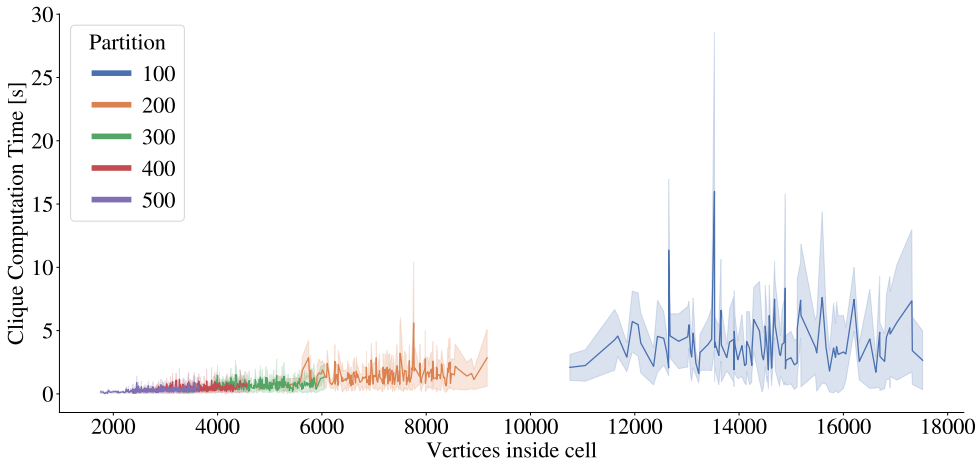


Figure 9 **MUSE clique computation time for Ile-de-France based on the number of vertices contained in each cell of the partition.**

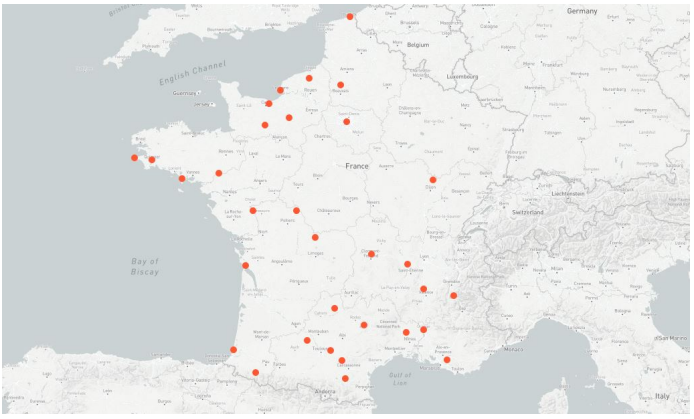


Figure 10 **Distribution of 32 landmarks on the France graph based on the Avoid algorithm (Goldberg and Harrelson 2005).**

paths that make up a clique overlap less when constrained by the road network compared to the public transit network. Hence, the gain factor is significant for automata $\mathcal{A}_1$ and $\mathcal{A}_3$.

For a better insight, figure 9 shows the preprocessing time (using the many-to-many strategy) of each cell in the partition as a function of the cell size, i.e., the number of vertices inside the cell. Even for the worst-case scenario, the computation time is less than 30 sec and corresponds to a large cell of the 100-cell partition with 14000 vertices. The overall processing time is usually less than 5 min, even without considering the benefit of parallel computing.

For the France region, we experimentally selected the partitions such that the cell sizes are equivalent to those of the Ile-de-France region, and therefore, preprocessing times are similar when one CPU is dedicated to each cell in both cases.

5.2.2. SDALT: The multimodal goal-directed algorithm known as SDALT improves upon the A* algorithm heuristics to achieve fast queries. SDALT relies on the *triangle inequality* principle (Goldberg and Harrelson 2005) to produce better bounds than Euclidean distances. To do so, it meticulously precomputes a small set of vertices L called *landmarks* spread across the network. Then, for each landmark $l \in L$, it precomputes a forward and reverse *constrained shortest path tree* rooted at l. We implement SDALT as described in (Kirchler et al. 2011). Preprocessing consists of computing the landmark-set and their associated shortest-path trees.

Table 4 Preprocessing space and time of SDALT (computing landmark distances) based on 32 landmarks for each automaton in the Ile-de-France region.

	$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{A}_3$	$\mathcal{A}_4$	$\mathcal{A}_5$
space [MB]	634	450	611	438	423
runtime [s]	64.92	83.54	79.92	108.04	106.07

Landmarks selection: we select 32 landmarks using the *avoid* algorithm (Goldberg and Harrelson 2005). The first landmark $l_0 \in L$ is chosen uniformly at random from the vertex-set V_f of the foot network. We compute all shortest paths from l_0 to every other vertex $v \in V_f$. Note that the reverse shortest paths have the same cost as the foot network is bidirectional. Then, we pick a random vertex $r \in V_f$ and compute the shortest path tree rooted at r, denoted T_r. For every other vertex $v \in V_f$, we compute its *weight* given by $weight(v) = d(r,v) - max\{d(l,v) \mid \forall l \in L\}$. The weight of a vertex v consequently represents the error in the distance estimation for r when using the set of landmarks L, including v.

To select a new landmark, we compute the *size* of every vertex $v \in V_f$. First we define the subtree $T_v \subset T_r$ rooted at v. In particular, we set $size(v) = 0$ if T_v contains a landmark $l \in L$: v is useless since another landmark exists that is already well located. Otherwise, $size(v)$ is evaluated as the sum of all the weights of the vertices in T_v. Finally, we select the vertex w greedily with the largest size, and we walk in its subtree T_w, traversing recursively, the child with the largest subtree until a leaf l is reached. That leaf l is then added to the set of landmarks L. This strategy produces good coverage landmarks that yield good lower bounds. This preprocessing phase is run only once.

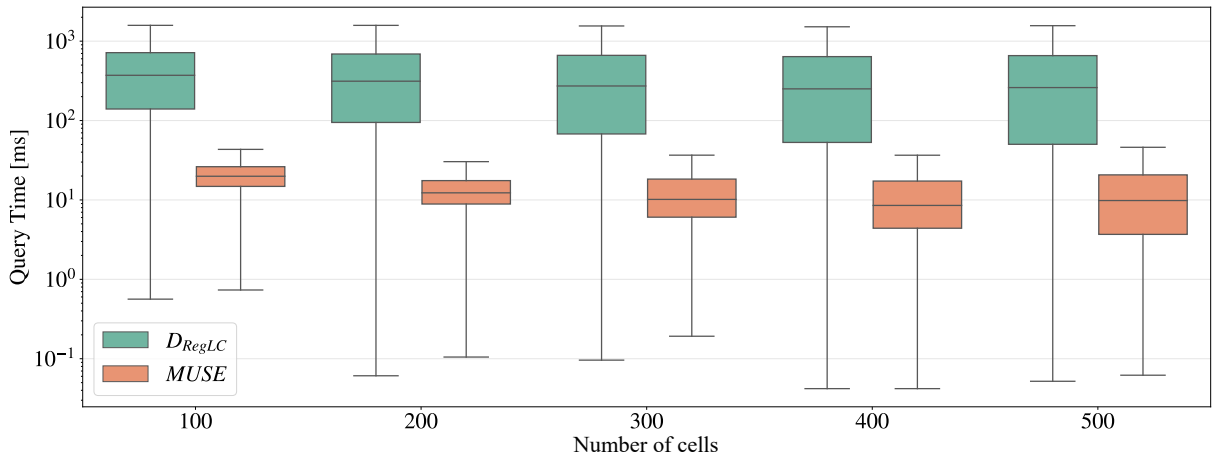
Landmark Distances: From each landmark $l \in L$, we run D_{RegLC} on the regular and the reverse multimodal graphs (we reverse the graph by inverting the direction of all its edges). Therefore, we obtain $|L|$ forward and backward constrained shortest-path trees rooted at each landmark l, referred to as the *landmark distances*. In the original SDALT implementation (Kirchler et al. 2011), the authors defined four strategies to precompute landmark distances: standard, basic, advanced, and specific. Based on the author's conclusions, the advanced strategy provides only mild speedups compared to the basic strategy and requires more memory as the number of calculated bounds is a factor $|L|$ higher in the worst case. Hence, we favor the basic implementation, which uses D_{RegLC} in conjunction with a single-state automaton with a loop transition labeled with all the allowed transportation modes.

The computational time for selecting 32 landmarks using the Avoid algorithm was 29 seconds for G_{idf} and 8.75 minutes for G_{fr}. The additional preprocessing time and space required to compute landmark distances is summarized in table 4 for G_{idf}. However, SDALT is not scalable: computing all shortest paths between the 32 landmarks and all the vertices of the France graph proved to be impractical, even with the resources available on the HPC.

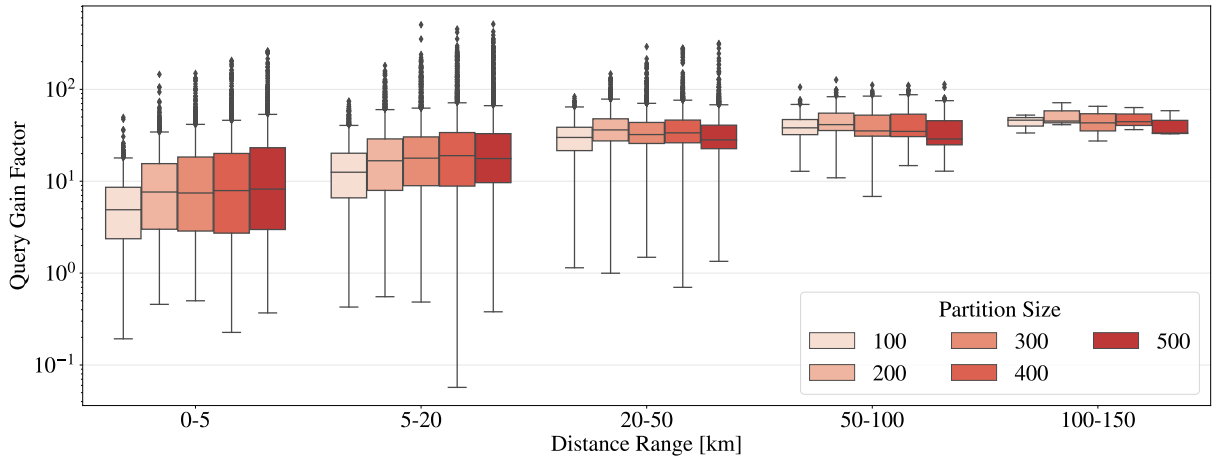
5.3. Queries

After precomputing an overlay graph will all the partitions and automata, we generate 10,000 random queries $q(r,t,\tau)$ where r,t are the source and target vertices respectively, and τ is the departure time. We compare MUSE with both D_{RegLC} (multimodal Regular Language Constrained Dijkstra (Barrett, Jacob, and Marathe 2000)) and SDALT (State-Dependent ALT (Kirchler et al. 2011)) (Table 5). Figure 11a depicts the query times of MUSE compared to D_{RegLC} on the Ile-de-France region, for all queries, regardless of the constraining automaton. MUSE achieves a speedup of almost two orders of magnitude over D_{RegLC}.

By organizing the queries according to their range (Euclidean distance between the source and the target), we observe that the speedup increases with the distance range (figure 11b). Indeed, the query time not only depends on the number of cells but also the location of the source and target vertices. Recalling that the query graph $G^q = G_r \cup H \cup G_t$ combines the subgraphs G_r and G_t, induced by the cells containing the source and target vertices r and t, respectively and the



(a) Query times distribution of D_{RegLC} and MUSE for all automata.



(b) The Query Gain Factor represents the achieved speedup of MUSE over D_{RegLC} for different partitions and query distance ranges (Euclidean distance).

Figure 11 Performance comparison of D_{RegLC} and MUSE on the Ile-de-France region graph.

overlay H, MUSE always explores G_r and G_t. Therefore, if the cells are large or if the source and target are either located inside the same cell or within adjacent cells, MUSE does not benefit from the overlay to skip over the rest of the graph. Hence, MUSE performs best for long-range queries, as shown in figure 11b.

In contrast, figure 12 shows the query times distribution of all of D_{RegLC}, SDALT, and MUSE for each automaton in the Ile-de-France region. We notice that the speedup of our own implementation of SDALT over D_{RegLC} is very close to the original results published by Kirchler et al. MUSE is overall an order of magnitude faster than SDALT and achieves better stability across all the automata. By comparing the query times distribution of SDALT constrained by $\mathcal{A}_1$ and $\mathcal{A}_5$, we observe that the automaton's complexity has a significant impact on the performance of SDALT. In contrast, MUSE is more robust to the modal constraints.

MUSE can be augmented using any literature technique that does not require preprocessing to achieve further speedups during the query phase. Bi-directional search has been proved to be inefficient (and often results in speed-downs) on time-dependent graphs (Nannicini et al. 2008). Instead, we relied on goal direction to implement MUSE* and MUSE^{SV} (Table 5), as already

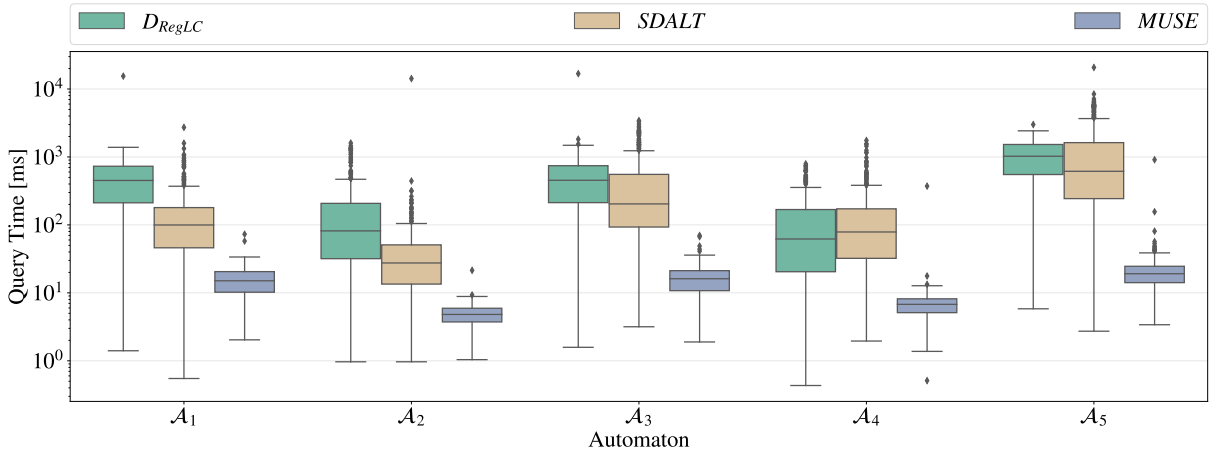


Figure 12 **Query times distribution of D_{RegLC}, SDALT, and the 300-based partition size MUSE for each automaton on the Ile-de-France region graph.**

Table 5 **List of algorithms used in the query phase of the experimental evaluation.**

D_{RegLC}	Regular Language Constrained Dijkstra (Barrett, Jacob, and Marathe 2000).
SDALT	State-Dependent ALT (Kirchler et al. 2011).
MUSE	similar to D_{RegLC} but runs on the query graph $G^q = G_r \cup H \cup G_t$, formed by the overlay H and the source and target induced subgraphs G_r and G_t, respectively.
MUSE*	augments MUSE to obey Goal-Direction principles, similar to A* (Hart, Nilsson, and Raphael 1968).
MUSE^{SV}	similar to MUSE* but implements the *Sedgewick-Vitter* heuristic (Sedgewick and Vitter 1986) to tradeoff correctness for the sake of speed.

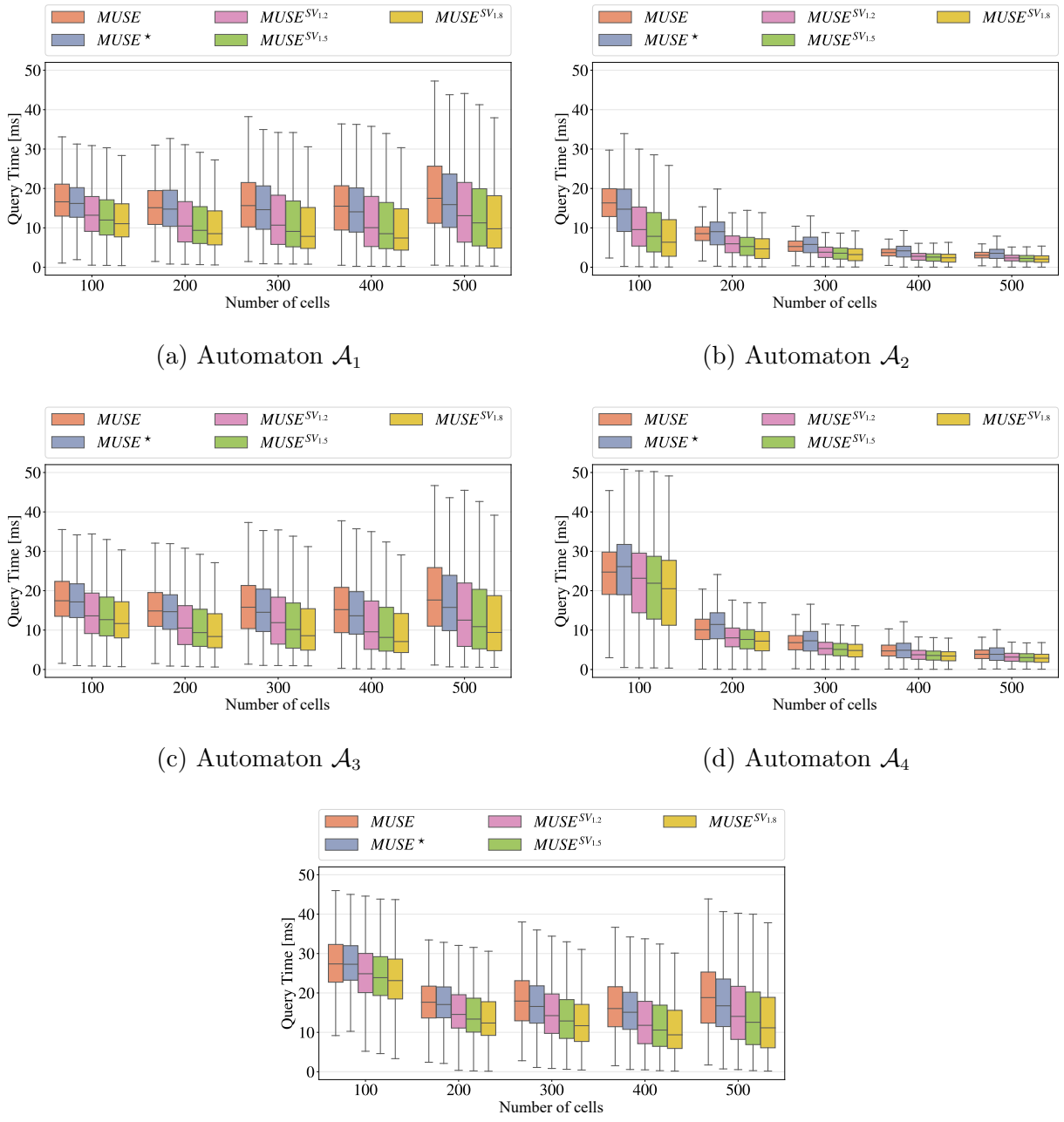
discussed in section 4.3. For the MUSE^{SV} variant, we tested three values for the heuristic tuning parameter $\alpha_1 = 1.2$, $\alpha_2 = 1.5$, and $\alpha_3 = 1.8$. We provide the results in figure 13, depicting the query time distribution of all MUSE variants for each automaton and number of cells.

We notice that query times on $\mathcal{A}_2$ and $\mathcal{A}_4$ are faster and keep decreasing when the number of cells increases. In contrast, an optimal number of cells (200 cells) exists for the automata $\mathcal{A}_1$, $\mathcal{A}_3$, and $\mathcal{A}_5$. Ultimately, query times are bound to increase again beyond a certain threshold of the number of cells. The largest possible partition is, in fact, $|V|$ (the size of the graph) with each cell containing a single vertex. Hence, the query graph is the whole graph, and MUSE degrades to D_{RegLC}. Considering that $\mathcal{A}_2$ and $\mathcal{A}_4$ exclude the public transit network, the link and merge operations, which are the bottleneck of the algorithm, are trivial, which explains why the queries are significantly faster.

Goal direction does not seem to be a good strategy to accelerate queries in the Ile-de-France region. In fact, Delling et al. (2011a) seem to obtain similar results for small partitions on unimodal road networks using Precomputed Cluster Distances (PCD) (Maue, Sanders, and Matijevic 2010), another goal directed algorithm.

In the France region, we fix the number of cells to 1,000 and provide the query times in figure 14. As query the distance range increases significantly (up to 1,000 km), the computational gain of MUSE over D_{RegLC} reaches up to 3 orders of magnitude, especially on automata that include the public transit network ($\mathcal{A}_1$, $\mathcal{A}_3$, and $\mathcal{A}_5$). Furthermore, on the same automata, MUSE* provides a significant speedup.

Using the *Sedgewick-Vitter* heuristic allows us to reduce query times by half in Ile-de-France and up to an order of magnitude in France depending on the automaton and number of cells (Fig. 14).



(a) Automaton $\mathcal{A}_1$

(b) Automaton $\mathcal{A}_2$

(c) Automaton $\mathcal{A}_3$

(d) Automaton $\mathcal{A}_4$

(e) Automaton $\mathcal{A}_5$

Figure 13 **Query times distribution for the Ile-de-France region for each automaton of figure 5. Each plot depicts the running times of MUSE, MUSE*, and MUSE^{SV} (with a heuristic factor of 1.2, 1.5, and 1.8) for each number of cell of the preprocessing.**

This is, however, at the cost of sacrificing the travel time correctness. We report in figure 15 the Travel Time Error (in minutes) for each MUSE^{SV} variant across all automata. MUSE$^{SV_{1.2}}$, the less aggressive overestimating variant, provides paths that are 99% of the time only a few seconds longer than the shortest paths. In the worst case, we measure a travel time error of 5 minutes. Interestingly, even MUSE$^{SV_{1.5}}$ and MUSE$^{SV_{1.8}}$ behave well if the automata do not allow using the public transit network ($\mathcal{A}_2$ and $\mathcal{A}_4$). Since the travel time function of public transit graph edges is

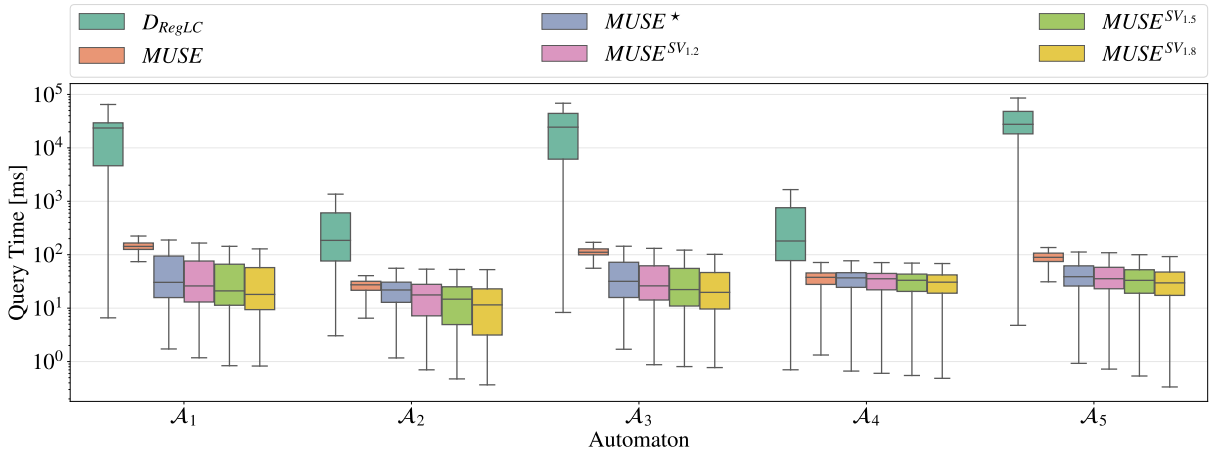
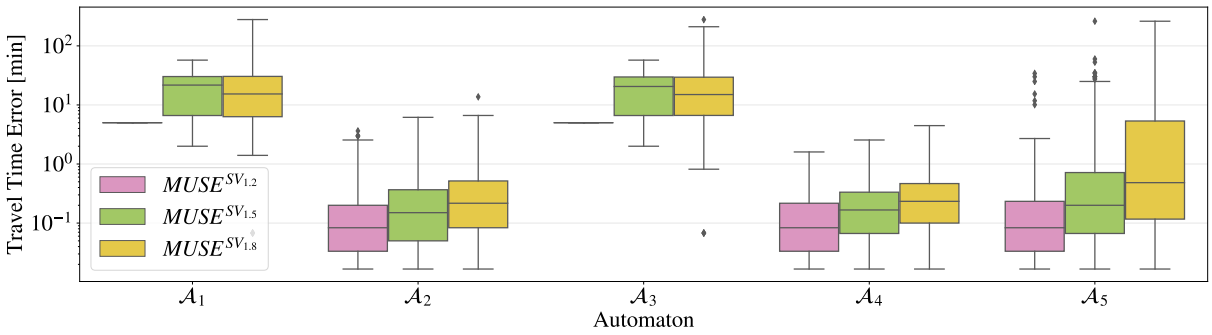
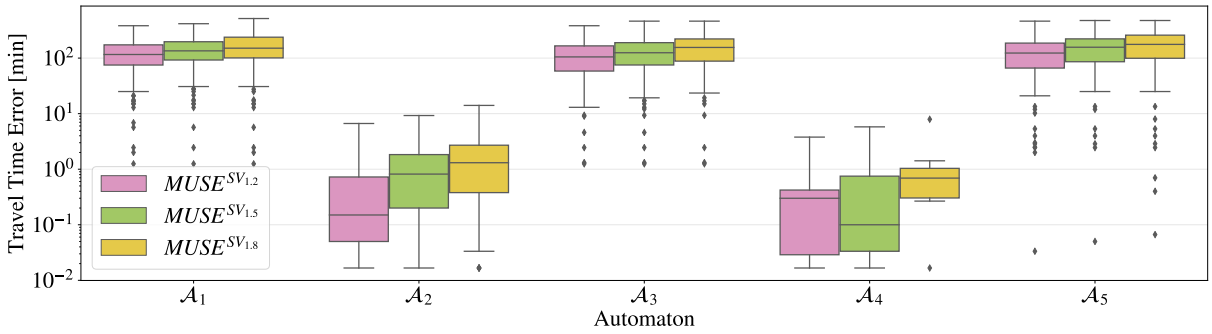


Figure 14 **On the France region, we fix the number of cells to 1,000 cells and report the query times distribution of D_{RegLC} versus MUSE and all of its variants, for each automaton.**



(a) Ile-de-France region



(b) France region

Figure 15 **Travel Time Error distribution on the Ile-de-France and France regions for each automaton based on the heuristic factor setting of MUSE^{SV}.**

non-continuous (figure 2), missing a connection results in a sudden increase of travel time, which explains the large errors observed on the remaining automata. Thus, this variant should be carefully exploited.

6. Conclusion and Future Work

We presented MUSE, a new approach to solve the multimodal shortest path problem via graph partitioning and finite state automaton. Its main advantage is scalability, as it splits large graph instances into several independent cells that can be processed in parallel. We also provide different strategies to improve the performance of both the preprocessing and the query phases. We achieve preprocessing and query times that are fast enough to process a cell and resolve a query in only a few milliseconds. Hence, MUSE is a viable solution for real-time multimodal route planning. We experimentally tested our implementation and achieved a speedup of up to two orders of magnitude compared to D_{RegLC}, the label constrained version of Dijkstra's algorithm for multimodal networks. Using goal direction and heuristics, we further accelerate the queries and show that the obtained paths are, most of the time, only a few seconds longer than the shortest paths. We also validated the performance of MUSE against SDALT, a landmark-based goal-directed multimodal algorithm. MUSE is an order of magnitude faster than SDALT and proved more robust to varying modal constraints. Furthermore, SDALT could not be scaled to handle large graph instances such as France's multimodal transportation network. A problem that MUSE circumvents using graph partitioning.

The partitioning is critical for the performance of both the preprocessing and the query. We first tested our implementation of the PUNCH algorithm (Delling et al. 2011b) but switched later on to METIS, the general graph partitioning algorithm, which was more reliable and provided better results. Therefore, a formal analysis of graph partitioning algorithms tailored to transportation networks is essential to improve our current results. Furthermore, to improve query times, multilevel partitioning for multimodal graphs is an interesting next step. It allows skipping over most of the network using the overlay edges on the partition's low level while reducing the search space in the source and target cells by exploring the high levels of the partition. During preprocessing, we also need to compute multimodal profile cliques. We observed that the *merge* and *link* operations are the bottleneck of lcD_{pRegLC} due to a rapidly increasing number of breakpoints in the travel time functions. An interesting work perspective would be to rely on heuristics (Strasser 2017) to simplify these operations and evaluate the tradeoff between computational speed and impact on correctness. Finally, we considered here only a route computation that minimizes the travel time. Handling multiple criteria will require computing the cliques in each cell by maintaining Pareto sets. Then, the different Pareto sets will be explored during the query stage. While we conjecture that keeping a small number of solutions would be sufficient, we have to investigate how MUSE can support multiple criteria with an acceptable computation time and memory.

References

Antsfeld L, Walsh T, 2012 *Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm. ITS World Congress.*

Barrett C, Jacob R, Marathe M, 2000 *Formal-language-constrained path problems. SIAM Journal on Computing* 30(3):809–837, URL http://dx.doi.org/10.1137/S0097539798337716.

Bast H, 2009 *Car or public transport—two worlds. Efficient Algorithms*, 355–367 (Springer), URL http://dx.doi.org/10.1007/978-3-642-03456-5_24.

Bast H, Delling D, Goldberg A, Müller-Hannemann M, Pajor T, Sanders P, Wagner D, Werneck RF, 2016 *Route planning in transportation networks. Algorithm engineering*, 19–80 (Springer), URL http://dx.doi.org/10.1007/978-3-319-49487-6_2.

Bast H, Funke S, Matijevic D, 2006 *Ultrafast shortest-path queries via transit nodes. Ninth DIMACS Implementation Challenge*, 175–192, URL http://dx.doi.org/10.1090/dimacs/074/07.

Bast H, Hertel M, Storandt S, 2016 *Scalable transfer patterns. Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, 15–29, URL http://dx.doi.org/10.1137/1.9781611974317.2.

Bauer R, Delling D, Wagner D, 2011 *Experimental study of speed up techniques for timetable information systems. Networks* 57(1):38–52, URL http://dx.doi.org/10.1002/net.20382.

Baum M, Buchhold V, Sauer J, Wagner D, Zündorf T, 2019 *UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. Annual European Symposium on Algorithms (ESA)*, volume 144, 14:1–14:16, URL http://dx.doi.org/10.4230/LIPIcs.ESA.2019.14.

Baum M, Dibbelt J, Pajor T, Wagner D, 2016 *Dynamic time-dependent route planning in road networks with user preferences. Experimental Algorithms*, 33–49, URL http://dx.doi.org/10.1007/978-3-319-38851-9_3.

Bentley JL, Ottmann TA, 1979 *Algorithms for reporting and counting geometric intersections. IEEE Transactions on computers* C-28(9):643–647, URL http://dx.doi.org/10.1109/TC.1979.1675432.

Bhatia N, et al., 2010 *Survey of nearest neighbor techniques. International Journal of Computer Science and Information Security* 8(2).

Brodal GS, Jacob R, 2004 *Time-dependent networks as models to achieve fast exact time-table queries. Electronic Notes in Theoretical Computer Science* 92:3–15, URL http://dx.doi.org/10.1016/j.entcs.2003.12.019.

Brüggemann-Klein A, 1993 *Regular expressions into finite automata. Theoretical Computer Science* 120(2):197–213, URL http://dx.doi.org/10.1016/0304-3975(93)90287-4.

Chen Y, Guizani M, Zhang Y, Wang L, Crespi N, Lee GM, Wu T, 2019 *When traffic flow prediction and wireless big data analytics meet. IEEE Network* 33(3):161–167, URL http://dx.doi.org/10.1109/MNET.2018.1800134.

Cionini A, D'Angelo G, D'Emidio M, Frigioni D, Giannakopoulou K, Paraskevopoulos A, Zaroliagis C, 2017 *Engineering graph-based models for dynamic timetable information systems. Journal of Discrete Algorithms* 46-47:40 – 58, URL http://dx.doi.org/https://doi.org/10.1016/j.jda.2017.09.001.

Delling D, Dibbelt J, Pajor T, Zündorf T, 2017 *Faster Transit Routing by Hyper Partitioning.* D'Angelo G, Dollevoet T, eds., *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59, 8:1–8:14, URL http://dx.doi.org/10.4230/OASIcs.ATMOS.2017.8.

Delling D, Goldberg AV, Pajor T, Werneck RF, 2011a *Customizable route planning. International Symposium on Experimental Algorithms (SEA)*, 376–387, URL http://dx.doi.org/10.1007/978-3-642-20662-7_32.

Delling D, Goldberg AV, Razenshteyn I, Werneck RF, 2011b *Graph partitioning with natural cuts. International Parallel & Distributed Processing Symposium (IPDPS)*, 1135–1146 (IEEE), URL http://dx.doi.org/10.1109/IPDPS.2011.108.

Delling D, Pajor T, Wagner D, 2009 *Accelerating multi-modal route planning by access-nodes. European Symposium on Algorithms*, 587–598, URL http://dx.doi.org/10.1007/978-3-642-04128-0_53.

Delling D, Pajor T, Werneck RF, 2014 *Round-based public transit routing. Transportation Science* 49(3):591–604, URL http://dx.doi.org/10.1287/trsc.2014.0534.

Delling D, Sanders P, Schultes D, Wagner D, 2006 *Highway hierarchies star. Ninth DIMACS Implementation Challenge*, 141–174, URL http://dx.doi.org/10.1090/dimacs/074/06.

Demetrescu C, Goldberg AV, Johnson DS, 2009 *The Shortest Path Problem: Ninth Dimacs Implementation Challenge* (American Mathematical Society), URL http://dx.doi.org/10.1090/dimacs/074.

Dibbelt J, Pajor T, Strasser B, Wagner D, 2013 *Intriguingly simple and fast transit routing. International Symposium on Experimental Algorithms (SEA)*, 43–54, URL http://dx.doi.org/10.1007/978-3-642-38527-8_6.

Dibbelt J, Pajor T, Wagner D, 2015 *User-constrained multimodal route planning. Journal of Experimental Algorithmics (JEA)* 19, URL http://dx.doi.org/10.1145/2699886.

Dijkstra EW, 1959 *A note on two problems in connexion with graphs. Numerische mathematik* 1(1):269–271, URL http://dx.doi.org/10.1007/BF01386390.

Djidjev HN, 1982 *On the problem of partitioning planar graphs. SIAM Journal on Algebraic Discrete Methods* 3(2):229–240, URL http://dx.doi.org/10.1137/0603022.

DOT U, 2015 *Beyond traffic 2045: Trends and choices*. Technical Report https://www.transportation.gov/sites/dot.gov/files/docs/BeyondTraffic_tagged_508_final.pdf, US Department of Transportation.

Eppstein D, Goodrich MT, 2008 *Studying (non-planar) road networks through an algorithmic lens. ACM SIGSPATIAL international conference on Advances in geographic information systems* (ACM), URL http://dx.doi.org/10.1145/1463434.1463455.

Falek A, 2019 *osm-to-graph*. https://github.com/aminefalek/osm-to-graph.

Falek A, 2021a *Muse: Multimodal separators for efficient route planning in transportation networks*. Zenodo dataset, URL http://dx.doi.org/10.5281/zenodo.5276749.

Falek A, 2021b *Muse: Multimodal separators for efficient route planning in transportation networks*. https://github.com/aminefalek/muse.

Garey MR, Johnson DS, 2002 *Computers and intractability*, volume 29 (W. H. Freeman & Co.).

Geisberger R, Sanders P, Schultes D, Delling D, 2008 *Contraction hierarchies: Faster and simpler hierarchical routing in road networks. International Workshop on Experimental and Efficient Algorithms (WEA)*, 319–333, URL http://dx.doi.org/10.1007/978-3-540-68552-4_24.

Geofabrik O, 2019 *Openstreetmap data extract*. URL https://download.geofabrik.de/europe/france/ile-de-france.html.

Giannakopoulou K, Paraskevopoulos A, Zaroliagis C, 2019 *Multimodal dynamic journey-planning. Algorithms* 12(10):213, URL http://dx.doi.org/10.3390/a12100213.

Goldberg AV, Harrelson C, 2005 *Computing the shortest path: A* search meets graph theory. ACM-SIAM symposium on Discrete algorithms (SODA)*, 156–165.

Goldberg AV, Kaplan H, Werneck RF, 2006 *Reach for a*: Shortest path algorithms with preprocessing. Ninth DIMACS Implementation Challenge*, 93–140, URL http://dx.doi.org/10.1090/dimacs/074/05.

Hart PE, Nilsson NJ, Raphael B, 1968 *A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107, URL http://dx.doi.org/10.1109/TSSC.1968.300136.

Hilger M, Köhler E, Möhring RH, Schilling H, 2009 *Fast point-to-point shortest path computations with arc-flags. Ninth DIMACS Implementation Challenge*, volume 74, 41–72, URL http://dx.doi.org/10.1090/dimacs/074/03.

Huber S, Rust C, 2016 *Calculate travel time and distance with openstreetmap data using the open source routing machine (osrm). The Stata Journal* 16(2):416–423, URL http://dx.doi.org/10.1177/1536867X1601600209.

Ile de France Mobilités, 2020 *Données offre de transport ile-de-france mobilités au format gtfs*. https://data.iledefrance-mobilites.fr/explore/dataset/offre-horaires-tc-gtfs-idf/information/.

Karypis G, Kumar V, 1998 *A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing* 20(1):359–392, URL http://dx.doi.org/10.1137/S1064827595287997.

Kaufman DE, Smith RL, 1993 *Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. Journal of Intelligent Transportation Systems* 1(1), URL http://dx.doi.org/10.1080/10248079308903779.

Kernighan BW, Lin S, 1970 *An efficient heuristic procedure for partitioning graphs. Bell system technical journal* 49(2):291–307, URL http://dx.doi.org/10.1002/j.1538-7305.1970.tb01770.x.

Kirchler D, Liberti L, Pajor T, Wolfler Calvo R, 2011 *Unialt for regular language contrained shortest paths on a multi-modal transportation network. Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, URL http://dx.doi.org/10.4230/OASIcs.ATMOS.2011.64.

Maue J, Sanders P, Matijevic D, 2010 *Goal-directed shortest-path queries using precomputed cluster distances. Journal of Experimental Algorithmics (JEA)* 14:3–2, URL http://dx.doi.org/10.1007/11764298_29.

Mooney P, Minghini M, et al., 2017 *Mapping and the Citizen Sensor*, chapter A review of OpenStreetMap data, 37–59 (Ubiquity Press), URL http://dx.doi.org/10.5334/bbf.c.

Müller-Hannemann M, Schulz F, Wagner D, Zaroliagis C, 2007 *Timetable information: Models and algorithms. Algorithmic Methods for Railway Optimization*, 67–90, URL http://dx.doi.org/10.1007/978-3-540-74247-0_3.

Nannicini G, Delling D, Liberti L, Schultes D, 2008 *Bidirectional a* search for time-dependent fast paths. International Workshop on Experimental and Efficient Algorithms*, 334–346, URL http://dx.doi.org/10.1007/978-3-540-68552-4_25.

Navitia Open Data, 2020 *France public transport data*. https://navitia.opendatasoft.com/explore/?refine.geographicarea=France.

Orda A, Rom R, 1990 *Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. Journal of the ACM (JACM)* 37(3):607–625, URL http://dx.doi.org/10.1145/79147.214078.

Pyrga E, Schulz F, Wagner D, Zaroliagis C, 2008 *Efficient models for timetable information in public transportation systems. Journal of Experimental Algorithmics (JEA)* 12, URL http://dx.doi.org/10.1145/1227161.1227166.

Pyrga E, Schulz F, Wagner D, Zaroliagis CD, 2004 *Experimental comparison of shortest path approaches for timetable information. ALENEX/ANALC*, 88–99 (Citeseer).

Sanders P, Schulz C, 2012 *Distributed evolutionary graph partitioning. Workshop on Algorithm Engineering and Experiments (ALENEX)*, 16–29 (SIAM), URL http://dx.doi.org/10.5555/2790265.2790267.

Sauer J, Wagner D, Zündorf T, 2020 *Faster Multi-Modal Route Planning With Bike Sharing Using ULTRA. 18th International Symposium on Experimental Algorithms (SEA)*, volume 160, 16:1–16:14, URL http://dx.doi.org/10.4230/LIPIcs.SEA.2020.16.

Schilling H, 2012 *Tomtom navigation–how mathematics help getting through traffic faster. International Symposium on Mathematical Programming (ISMP)*.

Sedgewick R, Vitter JS, 1986 *Shortest paths in euclidean graphs. Algorithmica* 1:31–48, URL http://dx.doi.org/10.1007/BF01840435.

Strasser B, 2017 *Dynamic time-dependent routing in road networks through sampling. Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, URL http://dx.doi.org/10.4230/OASIcs.ATMOS.2017.3.

Thomson RC, Richardson DE, 1995 *A graph theory approach to road network generalisation. International cartographic conference*, 1871–1880.

Ulloa L, Lehoux-Lebacque V, Roulland F, 2018 *Trip planning within a multimodal urban mobility. IET Intelligent Transport Systems* 12(2):87–92(5), URL http://dx.doi.org/10.1049/iet-its.2016.0265.

Yang F, Vereshchaka A, Dong W, 2018 *Predicting and optimizing city-scale road traffic dynamics using trajectories of individual vehicles. IEEE International Conference on Big Data (Big Data)*, 173–180, URL http://dx.doi.org/10.1109/BigData.2018.8622356.