Internship report – Contraction hierarchies in transportation networks

Olivier Marty, supervised by Laurent Viennot, INRIA, Université Paris 7

14th March – 12th August 2016

Le contexte général

La recherche de plus court chemin dans un graphe est un problème classique d'algorithmique. Dans un environnement où de nombreuses requêtes sont posées sur un même graphe, il devient intéressant d'effectuer un prétraitement, qui peut être coûteux, pour ensuite répondre aux requêtes plus rapidement. Plusieurs méthodes [BDG+15] existent et nombre d'entre-elles utilisent l'idée de raccourcis (ajouter des arêtes dans le graphe), dont *contraction hierarchies* [GSSD08] qui compte parmi les plus efficaces dans le cas des graphes routiers.

Le problème étudié

Nous avons essayé d'appliquer les idées de contraction hierarchies aux graphes de transport en commun (bus, métros...), ce qui pourrait accélérer le temps de réponse lors de la recherche d'un itinéraire. Avec ce type de graphe la donne change car le temps de parcours d'une arête dépend de l'heure de départ et certains sommets ont des degrés bien plus grands que dans un graphe routier.

Ce problème a déjà été étudié mais l'équipe s'était confronté à un échec et avait dû simplifier le graphe d'entrée [Gei09]. Cette simplification a le mérite de faire fonctionner contraction hierarchies mais impose de nombreux ajustements dans l'algorithme, et ne permet d'obtenir parfois qu'un résultat approché. Nous pensions que les ajustements effectués étaient plus compliqués que nécessaires, et voulions simplifier leur méthode.

La contribution proposée

Contraction hierarchies contracte les noeuds du graphe un par un, dans un ordre donné par des heuristiques. Après avoir essayé d'améliorer ces heuristiques et constaté que le prétraitement ne finissait pas en un temps raisonnable, nous avons observé que le graphe est décomposable à l'aide de petits séparateurs équilibrés. Nous avons utilisé cette décomposition pour trouver un ordre de contraction qui, de pair avec une représentation des données ad hoc, permet à l'algorithme de terminer suffisamment rapidement. En outre cet ordre défie la logique des heuristiques habituellement utilisées, c'est-à-dire qu'il commence par contracter les sommets habituellement réservés pour la fin.

Les arguments en faveur de sa validité

D'après nos expériences, contraction hierarchies ne permet pas toujours de calculer les plus court chemins de manière significativement plus rapide, mais les profil de temps (tous les plus courts chemins entre deux stations sur une journée entière) sont, eux, bien plus rapides à obtenir.

De plus, notre solution dépend uniquement de la structure du graphe : il doit avoir récursivement des petits séparateurs équilibrés. Le fait de supprimer certaines optimisations de contraction hierarchies classique rend indépendant la réussite du prétraitement aux poids des arêtes. Notre solution est donc plus robuste, car elle ne dépend que de la structure combinatoire du graphe donné (et de son plongement pour notre façon de trouver les séparateurs).

Le bilan et les perspectives

Notre étude montre qu'il est possible d'utiliser contraction hierarchies avec un graphe de transport en commun. La méthode utilisée est généralisable à tout graphe possédant des

petits séparateurs équilibrés récursivement et pondérés dans un semi-anneau pour lequel aucun cycle n'a de poids négatif (voir Section 3.5 pour une définition abstraite de l'absence de cycle négatifs).

Notre algorithme, au coût d'un prétraitement encore lourd, permet de calculer les itinéraires et les profils de temps en quelques millisecondes.

Finalement nous avons fait des expériences sur le graphe de transport en commun de la RATP connecté au graphes routier, ce qui permet de trouver des itinéraires mélangeant marche à pied et transports sans aucune limite sur la longueur des tronçons à pied. Si les temps de prétraitement et de requêtes sont bien trop longs pour être utilisable en pratique, c'est, à notre connaissance, la première méthode pour accélérer les requêtes sans imposer un temps de marche maximal.

During this Master 2 internship at the University Paris 7 Paris Diderot, I studied for five months with Laurent Viennot the contraction hierarchies technique in the context of transportation networks. Contraction hierarchies is a speed up technique for the Dijkstra's shortest path algorithm which works well with road networks, but fails with transportation graphs. We used one idea to enable the contraction of this kind of graph: the small balanced separator decomposition. Then we adapted the query algorithm to compute earliest arrival time queries as well as time profile queries. Finally we benefit from an interesting structure in the output of contraction hierarchies to design, at the cost of a long preprocessing, a very fast way to compute time profiles in transportation graphs.

1 State of the art

1.1 Contraction hierarchies

Contraction hierarchies [GSSD08, AFGW10] is a technique to speed up shortest path queries in road networks. During a preprocessing phase, shortcuts (new edges) are added to the given graph. Then these shortcuts along with original edges are used in a slightly modified bidirectional Dijkstra's algorithm to compute shortest paths.

The input of the algorithm is a directed weighted graph $G = (V, E)$, where V is its vertex set, and $E \subset V \times V \times \mathbb{R}^+$ its edge set. The set of weights $\mathbb{R}^+$ is the set of non negative reals.

We call the *distance* in G between two vertices u and v the minimum weight $\mathrm{d}(u, v)$ of a path from u to v:

$$\mathrm{d}(u,v) = \min \left\{ \sum_{i=1}^{n} a_i \;\middle|\; n \in \mathbb{N}, (u = w_0, w_1, a_1), (w_1, w_2, a_2), \ldots, (w_{n-1}, w_n = v, a_n) \in E \right\}$$

A *shortest path* is the associated sequence of edges.

1.1.1 Preprocessing

Given $G = (V, E)$, the preprocessing phase outputs a set of new edges E^+ and an ordering of V.

Suppose for the moment that the ordering of V is given, then all the vertices of the graph are contracted from lowest to highest with respect to this order. Each contraction consists in forgetting a vertex, and adding shortcuts between its neighbours in order to maintain shortest paths in the graph. More precisely, when v is contracted, if a path $(u, v, a), (v, w, b)$ is a shortest path, then a shortcut $(u, w, a + b)$ is added to the graph, its weight being the total weight of the path. After being contracted a vertex and its connected edges are forgotten until the end of the preprocessing phase.

The set E^+ produced by the algorithm is the set of shortcuts added during contractions.

Figure 1 illustrates the contraction of v. The path $(u, v, 1), (v, w, 2)$ is a shortest path so we add a shortcut $(u, w, 3)$. The path $(u, v, 1), (v, w', 2)$ is not a shortest path: no shortcut is added.

In order to limit the number of shortcuts added, we can perform a shortest path search to see if a shortcut may be unnecessary. However a complete shortest path search is too expensive, hence one solution is to look up among the paths of a maximum fixed number of edges. This method is called the witness search.

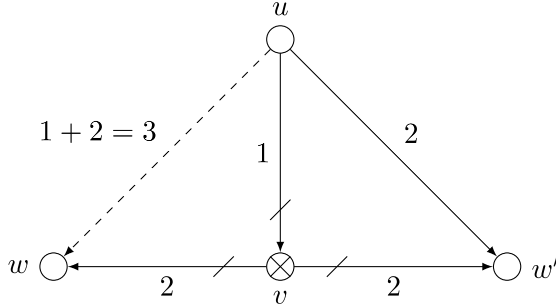

Figure 1: Contraction of v

A lot of other optimisations are possible.

A good order of contraction is the keystone of this algorithm. Indeed the queries will be exact for any order, but the number of shortcuts added and the execution time for preprocessing or answering queries may vary a lot.

1.1.2 Order of contraction

Finding the best order of contraction is APX-hard [Mil12], hence it is computed with some heuristics. Usually, and unlike our work, it is not set at the beginning but at each step the next contracted vertex is chosen. For this purpose, a linear combinations of different values is used, among them [GSSD08, BGSV13]:

- *Edge difference*: the immediate cost on the number of edges: #shortcuts inserted - #edges removed.
- *Cost of queries*, or *depth*: at the beginning $\mathrm{depth}(u) = 0$. Then when a vertex u is contracted, for all its neighbour v we set $\mathrm{depth}(v) = \max(\mathrm{depth}(v), \mathrm{depth}(u) + 1)$. This value is an upper bound for the length of a path looked up during queries, and helps to choose vertices more uniformly.

The vertex minimising the combination is contracted.

1.1.3 Shortest path queries

Given $G = (V, E)$, an order $<$ over V and a set of shortcuts E^+ (the output of the preprocessing) we define $G^\uparrow = (V, E^\uparrow)$ the upward graph where $E^\uparrow = \{(u, v, a) \in E \cup E^+ \mid u < v\}$ the subset of increasing (with respect to $<$) edges and shortcuts. Similarly, we define $G^\downarrow = (V, E^\downarrow)$ where $E^\downarrow = \{(u, v, a) \in E \cup E^+ \mid u > v\}$ are decreasing edges.

Distance We first describe how to find the distance between two vertices in G. Given a source vertex u and a target vertex v, the distance query consists in a bidirectional Dijkstra's algorithm, but forward search is performed in $G^\uparrow$, and backward search in $G^\downarrow$. More formally, forward search gives the distance in $G^\uparrow$ between u and each reachable vertex w. We note $\mathrm{d}^\uparrow(u, w)$ this distance (or $+\infty$ if w is unreachable). Likewise, we define $\mathrm{d}^\downarrow(w, v)$ the distance between w and v in $G^\downarrow$. Then we obtain the distance between u and v in G with

$$\min_{w \in V}(\mathrm{d}^\uparrow(u, w) + \mathrm{d}^\downarrow(w, v))$$

The intuition is that the highest vertex w on a shortest path from u to v in G in reached both in $G^{\uparrow}$ and $G^{\downarrow}$, and because shortcuts maintain shortest path, $d^{\uparrow}(u,w) = d(u,w)$ and $d^{\downarrow}(w,v) = d(w,v)$. A proof of the correctness is given in [GSSD08].

The limitation to $G^{\uparrow}$ and $G^{\downarrow}$ confines the exploration area and the distance computation is restricted to nodes reachable both in $G^{\uparrow}$ and $G^{\downarrow}$. This is why the queries are faster than plain bidirectional Dijkstra's algorithm.

Shortest path Dijkstra's algorithm can return paths along with distances, but here there are paths in $G^{\uparrow}$ and $G^{\downarrow}$ that may contain shortcuts. To retrieve the corresponding shortest paths in G, we must unfold the shortcuts. During the contraction, recall for each path $(u,w,a),(w,v,b)$ that leads to a shortcut $(u,v,a+b)$ that the middle vertex was w. Then given a path consisting of edges and shortcuts, we can retrieve the original path by recursively replacing shortcuts with the corresponding two edges.

1.2 Time dependant contraction hierarchies

Previously road networks were modeled with constant weight edges, but in real situations the traversal time of an edge can vary with traffic conditions. This is why an extension of contraction hierarchies for time dependant edge costs was proposed, using piecewise linear functions to model traversal time [BGSV13]. The cost of an edge is then a function which maps a departure time at the source of the edge to the corresponding arrival time at its target.

In this extension we distinguish two kinds of queries: earliest arrival time queries (EA) that ask for the earliest arrival time for a given source, a given target, and a given departure time, and time profile queries (TP) that ask for the earliest arrival time for a given source, a given target, and all possible departure times.

Some difficulties arise: for contractions there is no concept of shortest path because it depends on the departure time. For EA queries, the forward search is straightforward, during an edge traversal at a given departure time, the cost function is evaluated to get the arrival time. Only the arrival time is unknown for backward search. One solution is to mark vertices that would be explored during backward search, and perform the forward search with increasing edges and marked vertices. A lot of ideas to optimise the preprocessing phase and the queries are described in [BGSV13].

1.3 Station graph model for transportation network

In [Gei09] the same problem as ours has been studied: using contraction hierarchies with transportation networks. However the author finds out that the time dependant contraction hierarchies (see Section 1.2) fails on this kind of graphs, and proposes the station graph model. Indeed it is explained that transportation graphs have a high mean degree, as opposed to road graphs. As one contraction creates a number of shortcuts in the order of the square of the degree, the preprocessing takes too many time.

Their solution is to model each station with only one vertex. However stations may have several platforms, with different transfer times from one to another. The notion of different platforms is kept, but only one internal transfer time is allowed. Because of this model, the contraction of a node becomes trickier. Whether two trains share the same platform or not changes the transfer time, and the edge costs loose the FIFO property: sometimes we can choose between two trains: one arrives before the other, which leaves later but arrives at

the right platform for our change. Furthermore loops may appear in the graph: it may be interesting to take two trains rather than walking for a long change.

We think that this model is good enough for train network. Indeed each change has to be long enough to get into the next train, even with a small delay, and we don't often know the platform in advance. A minimal transfer time of 15 minutes, for example, is not absurd. However for a metropolis transport network, like one with bus and metro, platforms are known in advance and transfer time may vary from one to a dozen minutes in the same station.

Moreover the method we propose is more general as stations with constant time inner transfers can be modeled with small star graphs in our model.

1.4 Small balanced separators of road networks

A small balanced separator S of a graph $G = (V, E)$ is a subset of its vertices V, such that if we remove them the graph is split into several (at least two) connected components. To be small we ask the subset to contain few vertices – classically $|S| \leq |V|^{\varepsilon}$ for $\varepsilon < 1$. To be balanced it is required that the remaining connected components are relatively small – say at most $2|V|/3$ vertices.

Small balanced separators are useful hand in hand with divide and conquer algorithms: splitting the graph creates smaller instances, whose solutions may be patched up with the separator.

A simple heuristic that works well for road networks is the following [SS15]: sort the vertices by longitude (or another direction of the plan), then compute the minimum cut between the first and the last third of the vertices (with a maximum flow routine). This cut is a balanced separator of the graph, and is generally small.

1.5 Other methods

There exists other methods to compute journeys in graphs, in particular in transportation graphs. We reference two of them: RAPTOR [DPW15] and Transfer Patterns [BCE+10].

In a few words, RAPTOR is not Dijkstra-based but takes benefits from the structure of the networks in lines: at each step all reached lines are processed to jump to all their stations. It is possible to compute time profiles by launching one query for each departure time by decreasing order, in order to keep previously computed journeys which are still valid. Moreover RAPTOR does not perform any preprocessing, which makes it fully dynamic: graph updates are directly handled.

Transfer Pattern also uses the structure in lines, but it preprocesses the graph to finds where the changes are most often done. To make the preprocessing feasible the query results may be approximate: it finds journey with at most three different trains, which is often enough in practice.

Both methods can take into account several criteria, like the number of transfers, or the travel fees.

1.6 Contributions

We have, for the first time up to our knowledge, applied directly contraction hierarchies to some real life transportation networks, thanks to the small separators of the underlying graph found with [SS15]. If we were disappointed by the speed up obtained for earliest arrival time queries, time profile queries take advantage of a good performance. Our observation that

increasing and decreasing graphs (called $G^{\uparrow}$ and $G^{\downarrow}$) are directed acyclic graphs improves a little bit the time profile queries, and allows to precompute efficiently labels for all vertices. With these labels we can compute profile queries in few milliseconds only, and thus answer earliest arrival time queries in the same order of time. Finally, our first experiments with mixed transportation and road networks are not entirely satisfactory but our method seems to be the first speed up technique to compute routes in transportation networks without any limitation on the maximum walking time.

2 Definitions and problem statement

In this section, we state a general framework for using contraction hierarchies with time dependant edge costs.

2.1 Definition

Let $G = (V, E)$ be a directed labelled graph. Each edge (u, v, f) is labelled with an *arrival time function* (ATF) f which maps a departure time from u to the corresponding arrival time $f(t)$ at v.

Definition 1 (Arrival Time Functions)**.** *Let $\mathcal{D}$ be a totally ordered set, seen as dates. A set of functions*

$$\mathcal{T} \subset \mathcal{D}^{\mathcal{D}}$$

is a set of arrival time functions over $\mathcal{D}$ if

i) these functions are increasing: $\forall f \in \mathcal{T}, \forall x, y \in \mathcal{D}, x \geq y \implies f(x) \geq f(y)$

ii) these functions are above the first bisector: $\forall f \in \mathcal{T}, \forall t \in \mathcal{D}, f(t) \geq t$

iii) this set is closed under composition: $\forall f, g \in \mathcal{T}, f \circ g \in \mathcal{T}$

iv) and under point by point minimum: $\forall f, g \in \mathcal{T}, t \mapsto \min(f(t), g(t)) \in \mathcal{T}$

Remark 1. *In [BGSV13, Gei11, Gei09, Wir15] $\circ$ is called* link. *In [Wir15]* min *is called* merge. *In [BGSV13], time travel function (TTF) are used instead of ATF. They are similar, but a TTF gives the duration of a traversal, instead of the arrival time. For a given ATF f, if $\mathcal{D} = \mathbb{R}$, its corresponding TTF is $t \mapsto f(t) - t$.*

Condition ii ensures that one arrives after leaving. Closure under composition iii allows to concatenate edges: if one traverses the edge (u, v, f) then (v, w, g), then the ATF of the path is $g \circ f$. Closure under minimum iv allows to consider the shortest paths among two paths for each departure time: if one can choose between the path $u \to v \to w$, whose ATF is f, and the path $u \to v' \to w$, whose ATF is g, then $\min(f, g)$ gives the earliest arrival dates going through v or v'. Condition i gives a *FIFO* property, i.e. it is never faster to wait before traversing an edge. Moreover this property is needed to give a meaning to compositions: in order to get the earliest possible arrival time, we don't want to compute the arrival times for all possible waiting times at the inner vertex but only for no waiting.

2.2 Problem statement

A *path* of G is a finite sequence of edges $((s_0, t_0, f_0), \ldots, (s_n, t_n, f_n))$ such that two consecutive edges are connected: $\forall 0 \leq i < n, t_i = s_{i+1}$. Given a path P of G, we denote by $\bigcirc P$ the composition of the labels along the path, i.e. $\bigcirc((s_0, t_0, f_0), \ldots, (s_n, t_n, f_n)) = f_n \circ \cdots \circ f_0$. This function is the ATF of the path.

Definition 2 (Earliest arrival query)**.** *Given $G = (V, E)$, given a source u and a target v in V, given a departure time t in $\mathcal{D}$, we want to compute the earliest possible arrival time. That is $a \in \mathcal{D}$ such that*

$$\exists P \text{ path from } u \text{ to } v, \bigcirc P(t) = a$$

$$\forall P \text{ path from } u \text{ to } v, \bigcirc P(t) \geq a$$

This problem can be solved using Dijkstra's algorithm [BGSV13, Algorithm 1]: when an edge is traversed, the departure time is known, therefore its ATF is evaluated to get the corresponding arrival time. It works because of property i, and because $\mathcal{D}$ is totally ordered. This extension of Dijkstra's algorithm is the reason why ATFs are a natural extension to constant weights.

Now we are interested in time profiles: no departure time is given and we want to find the earliest feasible arrival dates to join v from u in G for all departure dates, as an ATF.

Definition 3 (Time profile query)**.** *Given $G = (V, E)$, given a source u and a target v in V, we want to compute the time profile going from u to v. That is the ATF f such that*

$$\forall t \in \mathcal{D}, \exists P \text{ path from } u \text{ to } v, \bigcirc P(t) = f(t)$$

$$\forall t \in \mathcal{D}, \forall P \text{ path from } u \text{ to } v, \bigcirc P(t) \geq f(t)$$

Remark 2. *An earliest arrival query is the evaluation of the time profile for the given departure date.*

This problem can be solved using the Bellman-Ford algorithm. In [BGSV13, Algorithm 2] a version is presented more like Dijkstra's algorithm, but where vertices can be examined several times.

2.3 Working framework

Two algorithms are already known to solve these two problems. However in the following we allow ourselves to make some calculations on the graph before any queries: the preprocessing phase. Preprocessing may take some time, though the queries can be answered several orders of magnitude faster. There is a trade-off between preprocessing time, memory usage, and query time.

3 Contraction hierarchies with ATFs

We present now an adaptation of contraction hierarchies for graph labelled with ATFs. The contraction of a vertex and the queries are really similar to previous works.

3.1 Preprocessing

3.1.1 Order of contraction

We tried classic heuristics for guessing a good enough order of contraction, but the preprocessing would not terminate in a reasonable time. However, we discovered with the heuristic of [SS15] (see Section 1.4) that our transportation graphs, seen as undirected for this purpose, have small balanced separators like road networks. For example we found a separator of 150 vertices in the RATP graph (see Section 5.2.4) which has more than 20.000 vertices.

Given a separator S, we contract, using recursively this technique, vertices in each connected component of the disconnected graph, and finally we contract each vertex in S. When the graph is small enough (we use a limit of around ten vertices) we contract all its vertices without using separators any more. During the contraction of a connected component, some shortcuts may be added in the separator, but never in other connected components. At most $|S|^2$ shortcuts are thus added in S.

This hierarchy of separators ensures that not too many shortcuts are added. If we note $C(n)$ the maximum number of shortcuts added in a graph with n vertices, and assume that the maximum size $S(n)$ of a separator satisfies $S(n) = O(\sqrt{n})$ (as in planar graphs), then

$$\begin{aligned} C(n) &= C(\alpha n) + C(\beta n) + S(n)^2 \qquad \text{with } \alpha, \beta \leq \frac{2}{3} \text{ and } (\alpha + \beta)n = n - S(n) \\ &\leq C(\alpha n) + C((1-\alpha)n) + O(n) \\ &= O(n \log n) \end{aligned}$$

There is at most a quasilinear, in the number of vertices, number of shortcuts. However note that each shortcut may be updated several times, hence this is not a bound on execution time.

We did not explore this idea, but this decomposition is favourable to parallel computing: the contraction of the two connected components are nearly independent. Only updates to the inner edges of the separator have to be taken carefully.

Finally, the biggest stations which have several vertices highly connected and look like cliques, won't be, presumably, in the first separators. Hence its vertices are contracted among the first, whereas they have the highest degrees. This is in conflict with the idea of starting the contraction with small degree vertices, which appear with classic heuristics (see Section 1.1.2).

3.1.2 Contraction

The contraction of a vertex is a straightforward adaptation of the classic contraction, but the sum of two weights become their composition, and their minimum become the point by point minimum. See figure 2 for an example.

Furthermore, we do not perform witness search: they are expensive and the order of contraction constructed with separators ensures that not too many shortcuts are created. We measured that trying to avoid some shortcuts does not improve significantly the size of the resulting graph, and increases the execution time. Another benefit is that the contraction will work in the same manner whatever the functions are.

3.2 Earliest arrival queries

We solve these queries like in [BGSV13]: we mark the vertices from which the target vertex is reachable in $G^{\downarrow}$, and we perform a Dijkstra's search using increasing edges (those in $G^{\uparrow}$)

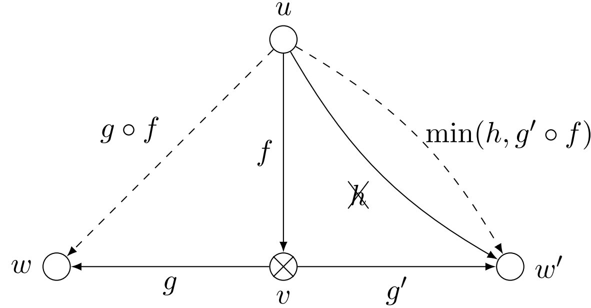

Figure 2: Contraction of v: a shortcut $(u, w, g \circ g)$ is created, and the ATF of the edge (u, w', h) is updated to $\min(h, g' \circ f)$.

and edges going to a marked vertex. For each edge, the ATF is evaluated to know the arrival time and its target vertex is inserted in the priority queue.

3.3 Time profile queries

For time profile queries, we want to answer an ATF that maps each departure time to the corresponding earliest arrival time. In [Gei11, section 4.1.3] and [BGSV13, Algorithm 2] an algorithm is proposed, which is an improved version of Bellman-Ford algorithm. It maintains an ATF from the source to each vertex, initialised to $t \mapsto +\infty$ for all vertices but the source for which it is initialised to $t \mapsto t$. Then it works like Bellman-Ford but only edges whose source has been updated are processed again, starting with the source. To do this the algorithm maintains a priority queue of vertices to be examinated, where the key is the duration of the shortest travel to this vertex (i.e. for an ATF f it is $\min_{t \in \mathcal{D}}(f(t) - t)$). In other words, the closest vertex throughout all possible departure times is processed: its neighbour are (re)inserted in the priority queue after their ATFs have been updated.

As for earliest arrival queries in the time independent contraction hierarchies, for each vertex v in V, we compute with this algorithm an ATF $f_v^{\uparrow}$ from the source in $G^{\uparrow}$ (forward ATFs), and an ATF $f_v^{\downarrow}$ to the target in $G^{\downarrow}$ (backward ATFs). Then the answer is the time profile

$$\min_{v \in V}(f_v^{\downarrow} \circ f_v^{\uparrow})$$

The correctness of the result follows from the correctness of the earliest arrival algorithm. Indeed for each departure time t, let v be the maximum vertex in the shortest path. Then $f_v^{\downarrow} \circ f_v^{\uparrow}(t)$ is exactly its arrival time, which appear in the final minimum. Moreover no better time can be achieved.

3.4 Time profile queries using dynamic programming

The following is, up to our knowledge, a new idea.

3.4.1 Dynamic programming

We note that the graphs $G^{\uparrow}$ and $G^{\downarrow}$ are directed acyclic graph (DAG). Indeed each edge in $G^{\uparrow}$ (resp. $G^{\downarrow}$) goes to a strictly larger (resp. smaller) vertices, with regard to the contraction order. In DAGs, we can compute distances using dynamic programming. At the end each

vertex knows the distance to all its successors. The initialisation takes care of the leafs, and once all successors of a vertex are done, we can compute the distance between this vertex and all its successors, until the roots are processed.

Here, we can use this method to construct the ATF between any pair of nodes in $G^{\uparrow}$ or in $G^{\downarrow}$. We also already know a topological order of these graphs: the one given by the order of contraction. Thus starting with the last contracted vertex, we construct the ATF from each and every vertex to all its successors.

This gives us an improvement of our time profile query: we can construct forward and backward ATFs more efficiently, the rest of the algorithm remaining the same.

3.4.2 Labelling scheme

This also gives us a natural labelling scheme. We define the label of a vertex as the time profiles to all its successors in the DAG $G^{\uparrow}$ as well as in $G^{\downarrow}$. All the labels can be computed using the dynamic programming process explained before. Then to answer a time profile query, we use directly the forward ATFs in the label of the source and the backward ATFs in the label of the target. This way, only the compositions between forward and backward ATFs and the final minimum are needed.

3.5 Generalisation

This adaptation of contraction hierarchies has been studied using graphs labelled with ATFs, which are a semiring whose multiplication is the composition and addition is the minimum. It is generalisable to any semiring such that no cycles are negative, i.e. [BT10]

$$\forall \text{cycle } c, w(c) + 1 = 1$$

where $w(c)$ is the weight of the cycle c and 1 is the identity element for the multiplication. Briefly when the non-negativity of cycle holds, we can compute shortcuts with the operations of the semiring, split a distance query onto increasing and decreasing graphs, and then patch the results together which gives the correct result because no cycles are allowed in shortest paths. Contraction hierarchies will work as soon as the graph has recursively small balanced separators.

4 Walk or bus networks

In [BGSV13] continuous piecewise linear functions are used to model arrival time functions. We propose to work only with a subset of piecewise linear functions which is especially adapted to public transportation graphs because it fits exactly the structure of ATFs in these graphs. Moreover these functions may not be continuous.

4.1 Definition

For the sake of simplification, we fix the set of dates $\mathcal{D} = \mathbb{R} \cup \{+\infty\}$, but the construction can be easily generalised to other dates, and in particular to finite sets. We consider two types of ATFs over these dates:

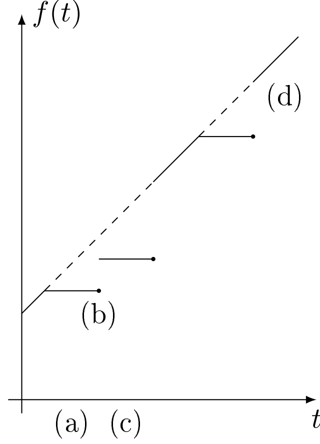

Figure 3: Example of an arrival time function f. t is the departure date, $f(t)$ the arrival. A dot (d, a) corresponds to a bus leaving at d and arriving at a (b). Right before we should wait the bus and, consequently, arrive at the same date (a). Right after we miss the bus and have to wait for the next one (c). Finally sometimes it may be faster to walk (d).

- The walk profiles: let $\mathcal{W} = \mathbb{R}^+ \cup \{+\infty\}$ be the set of durations. Given $w \in \mathcal{W}$, we define the walk profile
$$W_w : t \mapsto t + w$$
These profiles represent constant traversal time.

- The bus profiles: let $\mathcal{C} = \{(d, a), d \in \mathcal{D}, a \geq d\}$ be the set of *connections*: a departure date and a larger arrival date. Given $C \subset \mathcal{C}$, we define the bus profile
$$L_C : t \mapsto \min\{a \mid \exists(d, a) \in C, d \geq t\}, \text{ where } \min(\emptyset) = +\infty$$
which is a piecewise constant and increasing function on $\mathcal{D}$. These profiles can represent buses, subways, ferries, ..., connections.

We now define $\mathcal{T}$ the closure of $\{W_w, w \in \mathcal{W}\} \cup \{L_C, C \subset \mathcal{C}\}$ for composition and minimum, which are the so-called *walk or bus* profiles. An example is given figure 3.

Proposition 1. *$(\mathcal{T}, \min, \circ)$ is a semiring. Its zero is $t \mapsto +\infty$ (identity element for* min*), and its one is $t \mapsto t$ (identity element for $\circ$).*

Proposition 2. *Every profile in $\mathcal{T}$ can be written as the minimum of a walking profile and a bus profile*
$$\mathcal{T} = \{t \mapsto \min(W_w(t), L_C(t)), w \in \mathcal{W}, C \subset \mathcal{C}\}$$

Proof. Because $W_w = \min(W_w, L_\emptyset)$ and $L_C = \min(W_{+\infty}, L_C)$, it suffices to show that this set is closed under composition and minimum.
Given
$f(t) = \min(W_{w_f}(t), L_{C_f}(t))$
$g(t) = \min(W_{w_g}(t), L_{C_g}(t))$

we have

$$\begin{aligned}\min(f(t), g(t)) &= \min(W_{w_f}(t), W_{w_g}(t), L_{C_f}(t), L_{C_g}(t)) \\ &= \min(W_{\min(w_f, w_g)}(t), L_{C_f \cup C_g}(t))\end{aligned}$$

and

$$\begin{aligned}g \circ f(t) &= \min(f(t) + w_g, L_{C_g}(f(t)) \\ &= \min(t + w_f + w_g, L_{C_f}(t) + w_g, L_{C_g}(\min(t + w_f, L_{C_f}(t))) \\ &= \min(\underbrace{W_{w_f + w_g}}_{(a)}, \underbrace{L_{C_f}(t) + w_g}_{(b)}, \underbrace{L_{C_g}(t + w_f)}_{(c)}, \underbrace{L_{C_g}(L_{C_f}(t)))}_{(d)} \qquad \text{because } L_{C_g} \text{ is increasing} \\ &= \min(W_{w_f + w_g}, L_C)\end{aligned}$$

for some C because the minimum of three piecewise constant and increasing functions is a piecewise constant and increasing function. □

Remark 3. *Two ways to traverse each edges give four ways to traverse two edges in a row: walking and walking (cost (a)), taking a bus then walking (cost (b)), walking then taking a bus (cost (c)), or taking two buses (cost (d)).*

Definition 4 (Size of an ATF)**.** *We define the size of an ATF $f \in \mathcal{T}$ as the minimum number of connections needed to express f:*

$$\text{size}(f) = \min\{|C| \mid C \subset \mathcal{C}, \exists w \in \mathcal{W}, f = \min(W_w, L_C)\}$$

Proposition 3. *The size of the composition or the minimum of two ATFs is at most the sum of their sizes*

$$\forall f, g \in \mathcal{T}, \text{size}(\min(f, g)) \leq \text{size}(f) + \text{size}(g)$$

$$\forall f, g \in \mathcal{T}, \text{size}(f \circ g)) \leq \text{size}(f) + \text{size}(g)$$

Proof. The result is immediate for the minimum. For the composition, given an ATF f and a set of connections C which expresses f (i.e. there exists a walking time w such that $f = \min(W_w, L_C)$), since for one departure time only one arrival time is relevant, we can upper bound the size of f by looking at the number of different departure times there are in C. Resuming the proof of proposition 2, C is the union of $\{(d, a + w_g) \mid (d, a) \in C_f\}$ (from term (b)), $\{(d - w_f, a) \mid (d, a) \in C_g\}$ (from term (c)), and $\{(d, a') \mid \exists(d, a) \in C_f, a' = \min\{a' \mid \exists(d', a') \in C_g, a \leq d'\}\}$ (from term (d)). The departure times are $\{d \mid (d, a) \in C_f\}$ and $\{d - w_f \mid (d, a) \in C_g\}$, hence the bound. □

4.2 Implementation and algorithms

4.2.1 Data representation

Walking profiles are represented either with an integer, the number of seconds needed to traverse the edge, or $+\infty$ if no walking is possible.

Bus profiles are represented with an array of connections. Each connection is a pair of integers: the departure and the arrival date, in seconds since a fixed date. Moreover, this array is always sorted by increasing departure time, breaking ties by decreasing arrival time.

4.2.2 Dominated connections

The same ATF can be represented with more or less connections, all of them being not necessary meaningful. We explain how to prune the dominated connections to get a minimal set whose cardinal is the size of the ATF.

For example a bus that leaves at 8 a.m. and arrives at 11 a.m. is overtaken by a bus that leaves at 9 and arrives at 10. Indeed the second one leaves after and arrives before the first one, thus no shortest path that use the longer bus are not possible with the faster one. We say that the useful connection *dominates* the other one.

Proposition 4. *A dominated connection is meaningless in the bus profile. That is given a set of connections C, if $c = (d, a) \in C$ is dominated by $c' = (d', a') \in C$ i.e. $d \leq d'$ and $a \geq a'$, then*

$$L_{C \setminus \{c\}} = L_C$$

Proof. Recall that given a set of connections C,

$$L_C(t) = \min\{a \mid \exists (d, a) \in C, d \geq t\}$$

If $t > d$, then c is not considered in the minimum and $L_C(t) = L_{C \setminus \{c\}}(t)$.
If $t \leq d$, then $t \leq d'$ holds too. Both a and a' are considered in the minimum, and since $a' \leq a$, we have $L_C(t) = L_{C \setminus \{c\}}(t)$.

□

Likewise, a bus connection can be longer than a walk:

Proposition 5. *A connection longer than the walking time is meaningless in the walk or bus profile. That is given a set of connections C, a walking time w, and a connection $c = (d, a) \in C$, if $a - d \geq w$ then*

$$\min(W_w, L_{C \setminus \{c\}}) = \min(W_w, L_C)$$

Proof. If $t > d$ then c is not considered. Otherwise $t \leq d$ and $a \geq d + w \geq t + w = W_w(t)$. Thus a will never be meaningful in the minimum.

□

Because dominated connections are meaningless, we keep only dominating ones in an ATF during all the computations, while taking care to keep only one representative for duplicates. Connections dominated by a walk are plainly filtered out, and other dominated connections are removed with algorithm 1.

Proposition 6. *Algorithm 1 returns all non dominated connections of a sorted array of connections, and the result is sorted in increasing order for both departure time and arrival time. Its runtime is $O(n)$: linear in the size of the input.*

The algorithm processes each connection in increasing order. It maintains an array R of non-dominated (so far) connections. Before inserting another connection at the end of R, it checks if this connection dominates the last entry of R. If it is true, the last connection of R is removed and the current connection is compared again to the last entry of R. This way, all strictly dominated connections are removed, only one for duplicates is kept, and all other connections are kept in R.

Data: A sorted array of connections $C = [(d_1, a_1), \ldots, (d_n, a_n)]$
Result: A sorted array of non-dominated connections R

```
1  R ← empty array
2  i ← 1                                   // connection currently treated
3  while i ≤ n do
4      if R is empty then
5          R.append(d_i, a_i)
6          i ← i + 1
7      else
8          (d, a) ← R.last()
9          if a_i > a then
10             R.append(d_i, a_i)
11             i ← i + 1
12         else                            // R.last() is dominated
13             R.pop_last()
14         end
15     end
16 end
17 return R
```

Algorithm 1: Filter dominated connections

Proof. Correctness: We note $c_i = (d_i, a_i)$ for $1 \leq i \leq n$. Suppose that c_i is dominated by c_j, with j minimum. Then $d_j \geq d_i$ holds so we can suppose without loss of generality (in the case $c_i = c_j$) that $j > i$. We claim that for all $i \leq k < j$, c_k is dominated by c_j. Indeed $d_i \leq d_k \leq d_j$ holds and if $a_k \leq a_j$ then c_k dominate c_i which contradicts the minimality of j. Thus $a_k > a_j$ and c_k is dominated by c_j. This claim shows that a dominated connection is removed as soon as the first connection that dominates it is processed.

Runtime: At each iteration, either a connection is appended to R, or the last connection in R is discarded. Every connection can be appended only once, and can be discarded only once. Thus the runtime is $O(n)$.

□

In the following we call *cut* a routine that takes an ATF (a walking time and an array of connections), and returns an array of connections of minimum size for the same ATF.

4.2.3 ATF evaluation

The ordering and the filtering of dominated connections enable the evaluation of an ATF f for a given t in time $O(\log(\text{size}(f)))$, by binary search.

4.2.4 Minimum

The composition of two bus profiles uses the merge routine of the merge sort.

Proposition 7. *Algorithm 2 returns an ATF of minimum size that corresponds to the minimum of the two given ATFs. Its runtime is $O(n + n')$ where n and n' are the sizes of the given ATFs.*

Data: Two ATFs $w, C = [(d_1, a_1), \ldots, (d_n, a_n)]$ and $w', C' = [(d'_1, a'_1), \ldots, (d'_{n'}, a'_{n'})]$
Result: An ATF w_o, C_o
1 $w_o \leftarrow \min(w, w')$
2 $C_o \leftarrow \text{cut}(\text{merge}(L,\ L')),\ w_0)$
3 **return** w_o, C_o

Algorithm 2: Minimum of two ATFs

4.2.5 Composition

As we seen in the proof of proposition 3, the bus profile of a composition comes from three sets of connections. The last one (taking two buses in a row) is computable efficiently because the inputs are sorted. Algorithm 3 computes the composition of two ATFs in linear time.

Proposition 8. *Algorithm 3 returns an ATF of minimum size that corresponds to the composition of the two given ATFs. Its runtime is $O(n + n')$ where n and n' are the sizes of the given ATFs.*

Here are some explanations for the *connection and connection* part: we have seen that all possible changes are $\{(d, a') \mid \exists(d, a) \in C, a' = \min\{a' \mid \exists(d', a') \in C', a \leq d'\}\}$. Hence for all connection in C, as soon as the change is possible (line 10), a new connection is appended to the result. However for $(d_i, a_i), (d_{i+1}, a_{i+1}) \in C$ and $(d_j, a_j) \in C'$ with $a_{i+1} \leq d_j$, the connection (d_i, a_j) will also be dominated by (d_{i+1}, a_j) and is discarded (line 9). At the end, line 15 allows to iterate over C' until a feasible change is found.

5 Experiments

5.1 Environment

The tests have been performed on a CentOs 5.1, linux 2.6.18, using one core of an Intel Xeon E7420 64 bits, clocked at 2.13Ghz, with 132 GB of main memory and 8 MB of L2 cache. The code has been compiled with ocamlopt 4.02.3, and uses opam packages ocamlgraph 1.8.6 and csv 1.5.

5.2 Instances

Four instances were tested.

5.2.1 Instances with transportation network

Three instances come from a metropolis transit feed in the format GTFS [GTF]. The feeds were downloaded the 10th August 2016, and truncated to this one day.

- RATP
  `http://data.ratp.fr/explore/dataset/offre-transport-de-la-ratp-format-gtfs/`
- STIF
  `http://opendata.stif.info/explore/dataset/offre-horaires-tc-gtfs-idf/`
- VBB
  `https://daten.berlin.de/datensaetze/vbb-fahrplandaten-ende-juni-bis-dezember-2016`

```
Data: Two ATFs w, C = [(d_1, a_1), ..., (d_n, a_n)] and w', C' = [(d'_1, a'_1), ..., (d'_{n'}, a'_{n'})]
Result: An ATF w_o, C_o
1  CC ← empty array
2  CW ← empty array
3  WC ← empty array
   /* Walk and walk                                                        */
4  w_o ← w + w'
   /* Connection and connection                                            */
5  i ← 1                                          // current connection in L
6  j ← 1                                          // current connection in L'
7  while i ≤ n and j ≤ n' do
8      if i + 1 ≤ n and a_{i+1} ≤ d'_j then
9          i ← i + 1                                         // skip (d_i, a_i)
10     else if a_i ≤ d'_j then
11         CC.append(d_i, a'_j)                              // new connection
12         i ← i + 1
13         j ← j + 1
14     else
15         j ← j + 1                                         // skip (d'_j, a'_j)
16     end
17 end
   /* Connection and walk                                                  */
18 for i ← 1 to n do
19     CW.append(d_i, a_i + w')
20 end
   /* Walk and connection                                                  */
21 for j ← 1 to n' do
22     WC.append(d'_j − w, a'_j)
23 end
24 C_o ← cut(merge(CC, merge(CW, WC)), w_0)   // merge the three arrays and cut
25 return w_o, C_o
```

Algorithm 3: Composition of two ATFs

In this format each station is split in several stops which are different physical bus or train platforms, and are connected together with walking transfers. Transfers also occur between stops of two near stations. Unlike in the station graph model (see Section 1.3), we do not take into account this hierarchy: each vertex embodies one stop, and we directly model transfers with walk profiles, while transport connections are naturally modeled with bus profiles.

5.2.2 Instance with roads and transportation networks

The fourth instance RATP+OSM is based on the feed from RATP, but roads are added. The data comes from Geofabrik (`https://www.geofabrik.de/`), whose data comes from OpenStreetMap. In this graph road segments are edges while junctions are vertices. It is preprocessed and truncated to longitudes in the range $[2, 2.8]$ and latitudes in $[48.68, 49.06]$ (area around Paris). This data is only geographical, so we derive the walking traversal time of an edge from its length, assuming a walking speed of 6 kilometers per hour. Then all bus stops are connected to the nearest vertex in the road network. These connections are only an approximation of all possible changes, but they supply an easy way to link both graphs.

This double graph (transportation and roads) allows to find routes which mix walking and transportation in any order and for any walking distance. To our knowledge, this is the first speed up technique for computing such routes without any limitation on walking distance.

Because this graph is mainly composed of roads, it should be advantageous to apply contraction hierarchies optimisation for road graphs, which we did not tried.

5.2.3 Notes

We have written our program in Ocaml, which has good enough performance but uses more memory than low level languages. Thus there is scope for improvement with the memory footprints reported here. The same is relevant for graph sizes: our format is far to be optimal.

The first part of the table 1 describes instances, the second part the preprocessing phase, the third the contracted graphs (original graph with shortcuts added), the fourth describes labels computation with dynamic programming (see Section 3.4), and the last one describes performance on at least 100 random queries:

- *EA Dijkstra* is the classic Dijkstra's algorithm for earliest arrival time, on the input graph;
- *EA Dijkstra CH* is the bidirectional Dijkstra's algorithm on the contracted graph (see Section 3.2);
- *TP Bellman-Ford* is the classic Bellman-Ford for time profiles on the input graph;
- *TP Bellman-Ford CH* is the bidirectional Bellman-Ford on the contracted graph (see Section 3.3);
- *TP DP CH* is the same but use dynamic programming instead of Bellman-Ford (see Section 3.4);
- *TP labels* is the same but with precomputed labels (see Section 3.4).

5.2.4 Results

Table 1 presents the results of our experiments.

	RATP	STIF	VBB	RATP+OSM
Instance graph				
#vertices	22483	35040	11945	555291
#edges	185904	205032	41673	1405682
mean degree in/out	8/8	5/5	3/3	2/2
max degree in/out	128/128	85/84	20/21	128/128
mean/max ATF size	7/574	10/615	29/1372	1/574
size (Mo)	17	24	12	42
Preprocessing				
separator (s)	134	370	24	5143
contraction (s)	734	1897	17	113392
total (s)	868	2267	41	118535
memory usage (Go)	8.4	?	0.36	?
Contracted graph				
#edges	1062487	814364	97680	4610377
mean degree in/out	47/47	23/23	8/8	34/34
max degree in/out	1029/1018	1001/929	219/219	1708/1792
mean/max ATF size	84/847	57/1008	40/1372	110/885
size (Mo)	791	423	37	6306
Labels				
duration (s)		9709	152	
memory usage (Go)		114	4.1	
size (Go)		11	0.534	
Queries min/mean/max (ms)				
EA Dijkstra	0/140/853	0/304/25118	0/45/355	0/1011/7146
EA Dijkstra CH	28/302/663	34/179/315	2/14/41	19/1164/2472
TP Bellman-Ford	0/55028/281829	600/16725/262257	0/799/2896	0/101435/395141
TP Bellman-Ford CH	287/5405/24007	221/1259/7009	5/45/327	1275/11922/42847
TP DP CH	281/2690/10835	321/1044/4380	4/44/247	759/5338/19793
TP labels		0/3/13	0/0/4	

Table 1: Results of our experiments

Instances ATFs have small size in RATP: this is because a lot of changes are given, between two platforms of the same station of two near stations, and these changes have size 0 as they have only a walking profile. The mean size of ATFs that have at least one connection is 61.

Contracted graph The contracted graph has between 2 and 5 times more edges than the original graph, which is in some cases a little more than observed with road networks, for which it doubles.

EA queries No significant speed up has been measured on RATP and RATP+OSM. We think that this is due to the high degrees in the contracted graph: even if exploration areas are small in vertices, a lot of edges are traversed. On STIF and VBB, a speed up of 2 is measured on average, but may be a lot more with some queries. This is disappointing comparing to the

speed up measured with road graphs, which is of four orders of magnitude [BDG+15].

TP queries A speed up of 10 is measured on all instances. It is even a little bit faster with the trick of dynamic programming (especially on bad queries). Finally, TP queries with precomputed labels are extremely faster, within a few milliseconds, at the cost of a precomputing of a few hours. We recall that this result may be used to answer EA queries as well (see remark 2).

RATP+OSM The running times for this instance are too high to be useful: the preprocessing phase takes more than a day while queries last from 3 seconds for EA queries to a minute for TP queries.

5.2.5 Comparison with other algorithms

We did not run other algorithms on the same machine and the same instances, but we recall here some performances.

Station graph In [Gei09] some experiments are made with the graph VBB, but with data from the winter 2000/01. To compare raw results, our preprocessing is faster (41 seconds against 216), but our queries are slower: 15 vs 0.5ms for EA queries, and 44 vs 24ms for TP queries (TP without precomputed labels). However we take the data accurately, when station graph identifies all inner transfers to the same duration.

RAPTOR We did not use the same instances than the authors of [DPW15]. On their London instance, they perform EA queries in 7.3ms, and TP queries in 87ms. No preprocessing is necessary with this method.

Transfer Patterns As for RAPTOR we did not test our algorithm on the instances presented in [BCE+10]. Even if their instances are larger, their preprocessing takes several hundred of CPU hours, but the queries take less than 10ms to complete. We recall that the results can be approximate if the shortest path includes 3 changes or more.

Conclusion

Our first results were promising, applying contraction hierarchies to transportation graphs and getting a good speed up for time profile queries. Contraction hierarchies even worked on a graph with a transportation network as long as roads of its whole area. We would like to improve the preprocessing time on these kind of instances, which may be possible with usual optimisations and distributed computing. About our labelling scheme, labels are too expensive to be computed for the graph with road networks, thus we are investigating a trade-off between the number of precomputed labels and the query time with queries that mix partial Dijkstra's algorithms and labels data.

Finally I would like to thank Laurent for working with me, for his good mood and his good advice.

References

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 782–793. SIAM, 2010.

[BCE+10] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In Mark de Berg and Ulrich Meyer, editors, *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[BDG+15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.

[BGSV13] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18, 2013.

[BT10] John S. Baras and George Theodorakopoulos. *Path Problems in Networks*. Synthesis Lectures on Communication Networks. Morgan & Claypool Publishers, 2010.

[DPW15] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2015.

[Gei09] Robert Geisberger. Contraction of timetable networks with realistic transfers. *CoRR*, abs/0908.1528, 2009.

[Gei11] Robert Geisberger. Advanced route planning in transportation networks. *Diss. Karlsruher Instituts fr Technologie*, 2011.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

[GTF] GTFS format description. `https://developers.google.com/transit/`. Accessed the 14th August 2016.

[Mil12] Nikola Milosavljevic. On optimal preprocessing for contraction hierarchies. In Stephan Winter and Matthias Müller-Hannemann, editors, *5th ACM SIGSPATIAL International Workshop on Computational Transportation Science 2011, CTS'12, November 6, 2012, Redondo Beach, CA, USA*, pages 33–38. ACM, 2012.

[SS15] Aaron Schild and Christian Sommer. On balanced separators in road networks. In Evripidis Bampis, editor, *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2015.

[Wir15] Alexander Wirth. *Algorithms for Contraction Hierarchies on Public Transit Networks*. PhD thesis, Informatics Institute, 2015.