Multi-Itinerary Optimization as Cloud Service (Industrial Paper)

Alexandru Cristian
Microsoft
a-alcris@microsoft.com

Luke Marshall
Microsoft Research
luke.marshall@microsoft.com

Mihai Negrea
Microsoft
mihai.negrea@microsoft.com

Flavius Stoichescu
Microsoft
flavius.stoichescu@microsoft.com

Peiwei Cao
Microsoft
peiweic@microsoft.com

Ishai Menache
Microsoft Research
ishai@microsoft.com

ABSTRACT

In this paper, we describe Multi-Itinerary Optimization (MIO) – a novel Bing maps service that automates the process of building itineraries for multiple agents while optimizing their routes to save travel time or distance. MIO can be used by organizations with a fleet of vehicles and drivers, mobile salesforce, or a team of personnel in the field in order to maximize workforce efficiency. MIO accounts for service time windows, duration, and priority, as well as traffic conditions between locations, resulting in challenging algorithmic problems at multiple levels (e.g., calculating travel-time distance matrices at scale, scheduling services for multiple agents).

To support an end-to-end cloud service with turnaround times of a few seconds, our algorithm design targets a sweet spot between accuracy and performance. Towards that end, we build a scalable solution based on the ALNS meta-heuristic. Our experiments show that accounting for traffic significantly improves solution quality: MIO not only avoids violating time-window constraints, but also completes up to 17% more services compared to traffic-agnostic mechanisms. Further, our solution generates itineraries with better accuracy than both a cutting-edge heuristic (LKH3) and an Integer-Programming based algorithm, with twice and orders-of-magnitude faster running times, respectively.

CCS CONCEPTS

• **Information systems** → **Geographic information systems**; *Web services*; • **Theory of computation** → Randomized local search; Integer programming.

KEYWORDS

route optimization, traffic distance matrix, time-dependent travel

ACM Reference Format:
Alexandru Cristian, Luke Marshall, Mihai Negrea, Flavius Stoichescu, Peiwei Cao, and Ishai Menache. 2019. Multi-Itinerary Optimization as Cloud Service (Industrial Paper). In *27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19), November 5–8, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3347146.3359375

1 INTRODUCTION

In many businesses, route planning and service dispatch operations are a time-consuming manual process. This manual process rarely finds efficient solutions, especially solutions that can accommodate traffic, location changes or an increasing number of stops along a route. Additionally, scale is also a challenge: service dispatch planning may involve multiple vehicles that need to be routed between numerous locations over periods of multiple days.

The development of large scale internet mapping services, such as Google and Bing Maps, creates an opportunity for solving route planning problems automatically as a cloud service. Large amounts of data regarding geo-locations, travel history, etc. are being stored in enterprise clouds, and can in principle be exploited for deriving customized itineraries for corporate agents. The goal of such automation is to increase operation efficiency, by determining these itineraries faster (with less man-in-the-loop) and with better quality compared to manually produced schedules. However, multiple challenges stand in the way of making this vision a reality.

First, route planning requires efficiently calculating the travel-time matrices between different locations. While the problem is well understood for free-flow travel times (i.e., assuming no traffic) [8, 20], producing the traffic-aware (e.g., as a function of time-of-day) travel times on-demand and for any point in time requires careful attention to system scalability. Second, the route planning itself has to account for multiple features – time-windows, the priority of each location, amount of time spent in each location (e.g., service duration or dwell time), and the predicted traffic between locations. The single agent version with no traffic, time-windows, dwell-times, etc. corresponds to the Traveling Salesman Problem (TSP) which is already NP-hard. Numerous extensions to TSP have been studied in Operations Research and related disciplines under the Vehicle Routing Problem (VRP) [22, 23, 25, 29]. However, the bulk of the work is not readily extensible to account for traffic between locations, especially at a large scale. To address customer requirements, our service must incorporate traffic, and output an optimized schedule within seconds for instances with hundreds of waypoints.

In this paper, we describe Multi-Itinerary Optimization (MIO), a recently deployed Bing Maps service, available for public use [4]. The design of MIO tackles the above algorithmic challenges, as well as underlying engineering requirements (e.g., efficient use of cloud resources). In particular, our solution consists of a structured pipeline of advanced algorithms. At the bottom layer, we compute travel-time matrices by combining Contraction Hierarchies

(CH) [20] with traffic predictions, resulting in an efficient time-dependent shortest-path algorithm. We then use these matrices for itinerary optimization. Our algorithm for itinerary optimization is built on a popular meta-heuristic, Adaptive Large Neighborhood Search (ALNS) [27], which searches for an optimal schedule by judiciously choosing between multiple search operators (e.g., repair and destroy). Our search operators have been carefully designed to account for traffic and heterogeneous agents. The entire pipeline is implemented as a cloud service, which is easily accessible through a flexible REST architecture.

We perform extensive evaluations on several data sets to examine the quality of our end-to-end solution. In particular, we first highlight the significance of accounting for traffic in route planning. We find that if traffic is overlooked during planning (i.e., assuming free-flow travel times), up to 40% of the planned work (e.g., services, or dwell time at waypoints) violates the time-window system constraints. On the other hand, using conservative travel times (i.e., taking the maximum travel time between locations) results in schedules with up to 17% less satisfied work. Next, we compare MIO to both a state-of-the-art heuristic, Lin-Kernighan-Helsgaun (LKH [22]), as well as a mixed integer programming (MIP) based approach. Our results indicate that MIO obtains higher quality solutions in terms of the number of satisfied services, with less processing time – MIO runs up to 2× faster than LKH and orders of magnitude faster than the MIP approach.

While related commercial offerings exist (e.g., for field service, see Section 5 for an overview), companies typically do not disclose the details of the algorithms, and/or use some algorithmic components as a black box (e.g., the travel-time calculations). To the best of our knowledge, this paper is the first to report the full algorithmic details of an end-to-end cloud service that produces traffic-aware itineraries for multiple agents. The rest of the paper is organized as follows. Section 2 provides some necessary background; Section 3 outlines the details of our algorithms, as well of our cloud deployment. Section 4 describes our experiments and results. We survey related work in Section 5 and conclude in Section 6.

2 BACKGROUND AND MOTIVATION

Internet mapping services. The recent innovations in internet mapping services has created many new business opportunities for large cloud providers, such as Microsoft and Google. For example, in the Business-to-Consumer space, location based services can be used for more personalized user experience and better targeted advertising, among other things. In the enterprise resource planning context, tasks like fleet management and workforce scheduling / dispatching / routing (e.g., field technicians, pickup and delivery trucks, etc.) can benefit from automated GIS services built on accurate and up-to-date geospatial data. However, in the age of the internet, users of mapping services have high expectations. For example, a "large" scheduling task (say, hundreds to thousands of waypoints) is expected to complete within a few minutes, and smaller tasks (tens of waypoints) within seconds. From the cloud provider perspective, these expectations translate to Service Level Objectives (SLOs) on the end-to-end response time. Additionally, providers also wish to generate high-quality solutions (for example, minimize travel time or fuel consumption between a set of locations).

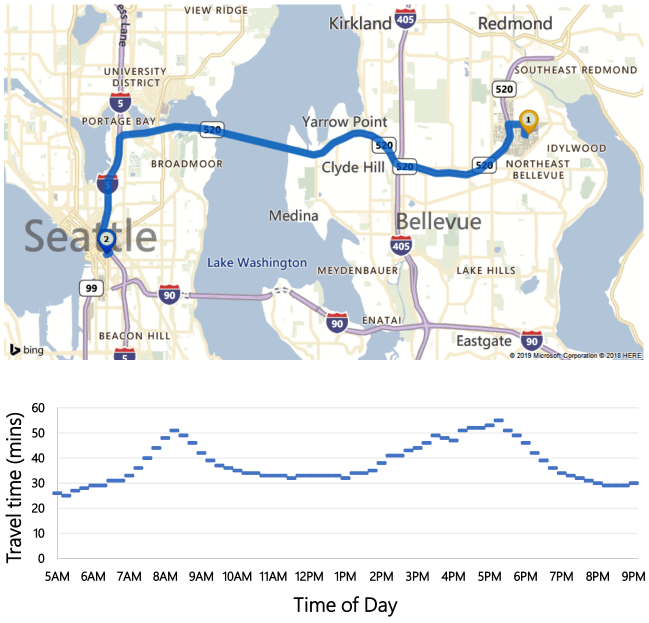

Figure 1: Time-dependent travel times for a single route

Multi-Itinerary Optimization. Bing maps recently deployed an enterprise level planning service called Multi-Itinerary Optimization (MIO) [4]. The term "itinerary" corresponds to a single agent (e.g., truck, or technician), and has carried over from an earlier consumer version for vacation planning. MIO takes as input a set of waypoints, each with a lat/long location, dwell time, time-window, and priority; and a set of agents, each with start/end location (which may differ across agents), and available time-window. Given this input, the goal of MIO is to find a feasible assignment (i.e., respecting time-window constraints) of a subset of waypoints to the given agents, which maximizes some system objective. For example, a popular objective is to maximize the number (or priority) of waypoints visited while minimizing the total travel time.

Design Challenges. The implementation of MIO as a cloud service has multiple levels of complexity. First and foremost, the service must scale robustly to client demand. Our travel time calculations involve taking a set of lat/long locations (anywhere in the world), snapping them to the road network, and quickly calculating pairwise shortest travel distance while accounting for traffic. This has required significant algorithmic and engineering effort to support large volumes of users and waypoints. On top of that, incorporating predictive traffic into route planning poses an additional level of complexity to a problem that is already NP-hard. One may wonder whether it is really necessary to account for traffic. Figure 1 demonstrates time-dependent travel times for a single origin-to-destination route (Microsoft campus to Seattle downtown). Notice that the travel time varies considerably throughout the day due to traffic, particularly during peak times (8-9 AM and 3-6 PM). With such significant variations in travel times, it is essential to explicitly account for traffic. Our experiments in Section 4.3 provide a quantitative analysis of this intuition and highlights potential business ramifications when traffic is ignored.

3 MIO DESIGN

In this section we provide the details of the core algorithms in MIO and its system architecture. Section 3.1 describes how we efficiently calculate travel-time matrices that account for traffic. These travel-time matrices serve as input to our route planning optimization (Section 3.2). In Section 3.3 we highlight some engineering choices that make MIO a scalable cloud service with manageable operation costs.

3.1 Travel-time calculations

Given a set of locations, a *distance matrix* is a two dimensional matrix constructed by calculating the length of the shortest-path between each pair of locations. This shortest-path can be considered to be physical distance, travel time, cost, etc. By convention we use the term *distance matrix* when the shortest-path minimizes free-flow (i.e. without traffic) travel time. A *traffic matrix* adds an extra dimension of time, where the shortest-path minimizes time-dependent travel over a given time horizon.

For efficient calculation, our algorithm for generating a traffic matrix uses an associated distance matrix as a baseline, and extends the time-dependent travel times using pre-computed predictive traffic. There are several implementations for fast distance matrix calculations based on hub labels [8] or contraction hierarchies [20]. Contraction hierarchies is an efficient approach (in data size, pre-processing and query speed) for distance calculations in road networks. It dramatically reduces the query time required to calculate shortest-path distances by performing a pre-processing step. The pre-processing step generates a multi-layered node hierarchy (vertex levels) formed by a 'contraction' step, which removes nodes and adds 'shortcuts' to preserve correctness. Our challenge was to integrate this method of fast computation of travel times, while taking account of traffic fluctuations.

3.1.1 Input and offline processing. Our system combines many sources of traffic related input (e.g. GPS traces) in order to estimate the travel times on every road at any moment in time. To give accurate day-of-week traffic predictions (with a granularity of 15 minutes), we aggregate these travel times for each road, using six months of historical data (giving more weight to recent travel).

After preprocessing our road graph with Contraction Hierarchies (CH), we compute the travel time for every contraction offline, based on the predictive traffic data and other graph properties like turn restrictions or turn costs, see Figure 2. As a result of this offline process we have two outputs:

(1) The CH graph (i.e., shortcut graph and vertex levels)
(2) The predictive traffic data for each edge in the CH graph containing 672 values (7 days x 24 hours x 4 quarter hours)

At query time, the CH graph is loaded in memory (~3GB for Western North America; ~85GB for the whole world) and the predictive traffic data is read from SSD using memory mapped files (~100GB for Western North America; ~5TB for the whole world). The query to calculate the traffic matrix for a given time horizon uses a time-dependent shortest-path algorithm based on bidirectional Dijkstra.

3.1.2 Time-dependent shortest-path. In general, the time-dependent shortest path problem is at least NP-hard (in some cases the output is not polynomially bounded), however under certain assumptions

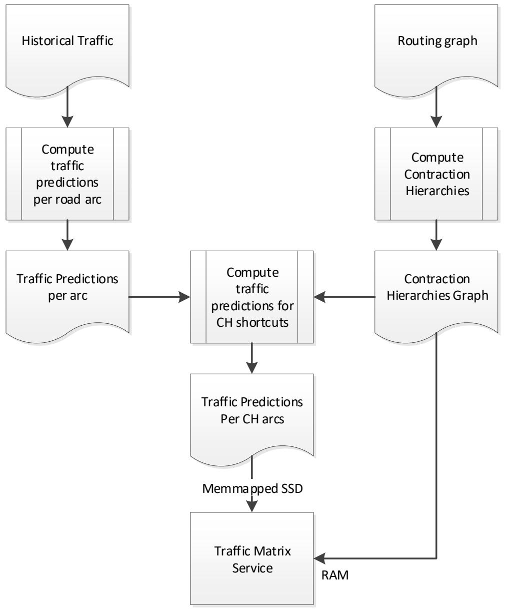
Historical Traffic
Routing graph
Compute traffic predictions per road arc
Compute Contraction Hierarchies
Traffic Predictions per arc
Compute traffic predictions for CH shortcuts
Contraction Hierarchies Graph
Traffic Predictions Per CH arcs
Memmapped SSD
Traffic Matrix Service
RAM

Figure 2: Offline pipeline for traffic matrix service

(FIFO networks), it can be solved efficiently with polynomial-time algorithms [13, 18, 21]. We assume this FIFO property, which essentially implies that later departures have later arrivals.

There are many variants of shortest path algorithms with time-dependent travel times, typically differing on if/how waiting at intermediate nodes is allowed. For example, minimizing the arrival time at the destination prohibits waiting at any node; minimizing travel duration allows waiting at the start (but not at intermediate nodes); and, minimizing travel time allows waiting everywhere (i.e. only the time spent travelling is counted). Other variants consider limits or differing costs on waiting. Regardless of the variant, the solution to these algorithms is a distance function, parameterized by a starting 'dispatch' time. Our algorithm has been specifically designed to produce a (three-dimensional) traffic matrix, that is, a piecewise-constant time-dependent distance function (discretized into 15 minute intervals) for each pair of locations. We assume that no waiting is allowed, and thus, for each dispatch time, we minimize the arrival time at the destination. We base our time-dependent shortest-path algorithm on a bidirectional Dijkstra algorithm, in order to quickly calculate all time-dependent distances from a single origin to many destinations simultaneously.

3.1.3 Algorithm details. CH is applied over the road network and the resulting shortcut graph and vertex levels are used. Predicted travel times are stored in 15-minute intervals over 1 week (for a total of 672 values). The CH graph is loaded into RAM, and traffic predictions are accessed via memory mapped files on SSD.

A traffic matrix request of N waypoints over a time horizon with T intervals, performs N individual one-to-N time-dependent shortest-path queries (one for each origin, for a total of $N \times N \times T$ travel times). The CH graph is used to expedite the bidirectional (forward / backward) search, since it is sufficient to only explore nodes that have a higher vertex level.

In the forward search, starting from the origin, vertices (with higher vertex levels) are explored based on a level based priority queue, by following outgoing hops (edges). For each vertex encountered, the time-dependent travel time is calculated from the origin, and the shortest travel times for each vertex (and requested time intervals) are cached. To improve query speed, the forward search is bounded by the number of hops from the origin (2× maximum number of shortcut edges in a contraction) and distance travelled (10× free-flow travel time from origin to farthest destination).

For each destination, a backward Dijkstra like search is performed, again by only visiting nodes (via incoming edges) with higher vertex levels in the CH graph, and using free-flow travel time to determine the shortest path. Once a vertex is reached by the backward search that has been seen during forward search, the backward route from the found vertex to the destination, and the travel times (with predicted traffic) are calculated. Each resulting travel time at each interval is then compared with previously found travel times (initially set to a very high value) – if any are smaller than what has already been found, then the results array is updated. After a fixed maximum number of rounds have been performed without finding a smaller travel time for any interval, the backward search is stopped. After all destinations have been processed, the algorithm returns the $N \times T$ shortest travel times for the given origin node and time horizon.

We note that using free-flow travel times to guide the backward search may result in an approximation to the time-dependent shortest-path, however this is unlikely to occur in practice due to the continued search for better solutions after the first intersection between the forward and backward graphs was found.

3.1.4 Algorithmic performance. We evaluated the traffic matrix performance on a queryset of 1200 instances consisting of waypoints from Germany. These queries vary by the distance between waypoints and the matrix request size, with 100 queries per variant. Each query was run with a time horizon of both 96 and 672 intervals (a single day and a full week, respectively), and the output is shown in Table 1. The average response times were partitioned by distance, number of intervals, and the size of the request matrix – where *cold* indicates the response time when traffic data was read from SSD, and *warm* indicates the response time when the traffic data was already cached in memory.

Size	Intervals	Distance (km)	Cold (s)	Warm (s)
10x10	96	<15	0.1224	0.0656
10x10	96	<80	0.3620	0.0810
10x10	96	<250	1.0522	0.0887
10x10	96	<15	0.1224	0.0656
10x10	672	<15	0.3384	0.3085
1x10	672	<15	0.0786	0.0363
10x10	672	<15	0.3384	0.3085
1x100	672	<15	0.2163	0.1737
100x100	672	<15	17.8519	17.8524

Table 1: Traffic matrix cold/warm response times

Observe that distance has little impact on the *warm* response time, however it significantly impacts the *cold*. This is due to less vertices being shared between the routes, causing more SSD trips to fetch the traffic data; as well as requiring a longer forward exploration phase. On the other hand, the *warm* response time seems to be most impacted by the number of intervals – requesting 672 intervals is clearly more expensive than 96 intervals for both *cold*/*warm*, but the *warm* response time for the full week is almost equivalent to the *cold*. Finally, changes in matrix size impacts performance as expected – all experiments were run single threaded, hence the linear relationship between the single source matrix (e.g., 1x10) and the multiple source matrix (e.g., 10x10). This relationship is most noticeable for the *warm* response time (e.g., 0.03 and 0.3 seconds respectively).

3.2 Itinerary Optimization

Equipped with travel-time matrices, our MIO service solves the (NP-hard) technician routing and scheduling problem (TRSP). We describe below our algorithm design, which pays close attention to performance and scale.

3.2.1 Simplified MIP Formulation. The complete formal definition of our problem (i.e., with traffic, generalized objectives, multiple days, etc.) is difficult to express in a simple compact MIP representation. For the sake of exposition, we present a simplified model that highlights the important aspects of our optimization problem. Note that the complete model is best expressed as a MIP using advanced decomposition techniques (i.e., branch-and-price [9]).

The challenge of the TRSP is to schedule a set of agents to visit a set of waypoints, where each waypoint is visited at most once. When visiting a waypoint the agent must dwell at its location (e.g., perform a service) within the time-window associated with the waypoint. In particular, the service must start within the time-window. Agents may arrive earlier, but they must wait until the start of the time-window before they can begin their service. Our objective is to visit as many waypoints as possible, while minimizing the total travel time of the agents.

Sets

W	Waypoints
K	Agents
N	Combined Agent start/end locations and waypoints

Parameters

$\delta_n^k \in \mathbb{R}_+$	dwell time at location $n \in N$ for agent $k \in K$
$\tau_{(n,n')} \in \mathbb{R}_+$	free-flow travel time between $n, n' \in N$
$s_n, e_n \in \mathbb{R}_+$	start/end time window for $n \in N$
$o^k, d^k \in N$	origin/destination for agent $k \in K$

Decision variables

$x_{n,n'}^k \in \{0,1\}$	1 if n is visited before $n' \in N$ by agent $k \in K$
$y_w \in \{0,1\}$	1 if waypoint $w \in W$ is ever visited
$t_n^k \in \mathbb{R}_+$	arrival time at $n \in N$ by agent $k \in K$

Model

$$\max \left\{ \sum_{w \in W} y_w, - \sum_{n \in N} \sum_{n' \in N} \tau_{(n,n')} \sum_{k \in K} x^k_{n,n'} \right\} \tag{1}$$

s.t.

$$y_w = \sum_{k \in K} \sum_{n' \in N} x^k_{n',w} \quad \forall\, w \in W \tag{2}$$

$$\sum_{n' \in N} x^k_{n,n'} - \sum_{n' \in N} x^k_{n',n} = \begin{cases} 1 & n = o^k \\ -1 & n = d^k \\ 0 & \text{o/w} \end{cases} \quad \forall k \in K,\, n \in N \tag{3}$$

$$s_n \le t^k_n \le e_n \quad \forall k \in K,\, n \in N \tag{4}$$

$$t^k_{n'} + M\left(1 - x^k_{n,n'}\right) \ge t^k_n + \delta^k_n + \tau_{(n,n')} \quad \forall k \in K,\, n, n' \in N \tag{5}$$

$$\sum_{n \in N} \sum_{n' \in N} \left(\delta^k_n + \tau_{(n,n')}\right) x^k_{n,n'} \le e_k - s_k \quad \forall k \in K \tag{6}$$

$$x^k_{n,n'} \in \{0,1\} \quad \forall k \in K,\, n, n' \in N$$
$$y_w \in \{0,1\} \quad \forall\, w \in W$$
$$t^k_n \ge 0 \quad \forall k \in K,\, n \in N$$

The multiple objective (1) hierarchically maximizes the total number of waypoints visited while minimizing the total travel time. This can be easily extended to maximize total priority of visited waypoints, by providing a suitable weight for the y_w. Constraints (2) capture if a waypoint is visited, and enforce that it is visited only once. The flow constraints (3) ensure that each agent leaves their starting location, visiting zero or more waypoints, and reaches their end location. The next set of constraints deals with time: (4) ensures that an agent arrives within the requested time window of the waypoint – note that they can arrive early, but must wait until s_n. (5) correctly accounts for the dwell and travel time between waypoints, and (6) enforces that the agent can visit all scheduled waypoints within their available time window. The remaining constraints define the variable domains.

3.2.2 MIO algorithmic approach. Adaptive Large Neighborhood Search (ALNS) is a popular meta-heuristic used for many vehicle routing and optimization problems in general. ALNS uses a simulated annealing framework, with a local search at each iteration that adapts its behavior based on previous iterations. This local search is controlled by randomly choosing a pair of destroy/repair operations (from a predefined set). Then, starting with a feasible solution, the destroy "local search" modifies the solution, such that the solution may become infeasible. The repair operation then alters the solution with a guarantee of feasibility. If the new solution is better than the previous, then the probability of choosing these destroy/repair operations will increase. This is the adaptive learning component of ALNS. Our implementation follows a similar approach as outlined by [27], and in addition, we support parallel processing and multiple objectives (via an automatic weighting scheme). See Algorithm 1 for a high-level overview of the approach. Our algorithm terminates when either the current solution has not changed within the last 1500 iterations (i.e., all new solutions were rejected), or a fixed timeout, given by # agents × # waypoints × 600 milliseconds, has been reached.

The most important component for efficient computation is the design of the destroy and repair operators. These guide the local search and attempt to exploit the structure of the combinatorial optimization problem. A careful balance must be considered – a trade-off between fast iterations and intelligent decisions. Below we list our set of operators, with a brief description of their intent.

Algorithm 1 High-level ALNS algorithmic framework

```
1:  procedure ALNS_SOLVE($x, T, \alpha, \pi$)
2:      $X_{best} \leftarrow x$
3:
4:      while not terminated do
5:          REPAIR, DESTROY $\leftarrow$ CHOOSE_PAIR($\pi$)
6:          $x' \leftarrow$ REPAIR(DESTROY($x$))
7:
8:          if OBJ($x'$) > OBJ($X_{best}$) then
9:              result $\leftarrow$ NewIncumbent
10:             $X_{best} \leftarrow x'$
11:             $x \leftarrow x'$
12:         else if OBJ($x'$) > OBJ($x$) then
13:             result $\leftarrow$ DominatesCurrent
14:             $x \leftarrow x'$
15:         else if SIMULATED_ANNEALING($x, x', T$) then
16:             result $\leftarrow$ Accepted
17:             $x \leftarrow x'$
18:         else
19:             result $\leftarrow$ Rejected
20:         end if
21:
22:         $\pi \leftarrow$ UPDATE_WEIGHTS($\pi$, result, REPAIR, DESTROY)
23:         $T \leftarrow \alpha T$
24:     end while
25:
26:     return $X_{best}$
27: end procedure
```

Destroy Methods RANDOM: removes a random number of waypoints from the current schedule; BEST: removes the most 'expensive' (e.g., in terms of travel distance) waypoints; SWAP: performs a 2-opt swap heuristic on the current schedule, ignoring time-window constraints.

Repair Methods NN: inserts waypoints using the nearest neighbor heuristic; BEST: attempts to insert waypoints in the best position (i.e., checks the benefit of each insertion); EARLY: prefers to insert random waypoints near the beginning of an agents schedule; LATE: prefers to insert random waypoints near the end of an agents schedule; END: appends waypoints to the end of the agents schedule (if possible).

3.3 Cloud deployment and engineering

The design of MIO as a cloud service must take into consideration resource costs. We need to store large amounts of data in memory (for hub labels and the contraction hierarchies graph along with supporting structures ~200GB) and on SSD (traffic data ~5TB), so the cost of the underlying virtual machines (VMs) might be high. A simple implementation would consist of a single VM role that hosts the entire service, where the number of the VMs can be scaled based on request volume. However, we would like to exploit the special structure of our service for a more resource efficient architecture. From an operational perspective, we have three different components that support the entirety of the algorithms described earlier in this section: distance matrix, traffic matrix and route optimization. As we elaborate below, these components have fundamentally different resource requirements.

(1) *Distance Matrix* which needs a lot of memory to host the hub labels;
(2) *Traffic Matrix* which needs little memory to host the contraction hierarchies graph. On the other hand, the component is CPU intensive and requires a lot of SSD storage.
(3) *Route Optimization* which does not need to hold any persistent data in memory; the computation itself is CPU intensive.

Accordingly, the MIO system includes three different VM roles, one for each of the above components (See Figure 3 for a high-level architecture overview). The roles communicate via binary HTTP protocol. Each component can be separately scaled down/up based on current load conditions. Specifically, the distance matrix role is hosted on memory optimized VMs [3]; because our design achieves high throughput, we rarely need to scale down/up this component. For the traffic matrix, storage needs to be provisioned for the maximum load conditions. Nonetheless, we use compute optimized VMs [3] for the computation; these VMs can be scaled down/up quickly to reduce operating costs. Similarly, the route optimization role is hosted on compute optimized VMs that are scaled down/up as necessary. Obviously, the alternative implementation with a single VM role would not be able to achieve this level of flexibility and efficiency - any scale up is likely to result in resource wastage in at least one resource dimension.

To enable the above described flexibility, each of the three roles is composed of:

(1) A dynamic scale-set of VMs that scales based on a custom performance counter; the counter is tuned to the needs of the specific role.
(2) An Azure load balancer which spreads the requests to the VM scale-set.
(3) An Azure traffic manager which handles the distribution of requests to the closest datacenter, as well as failover management in case of outage.

We conclude this section by briefly describing the interface of MIO. MIO can be used by applications with different requirements. For example, there are applications that need to display the output schedule quickly. On the other hand, there are applications that require solving large route optimization problems. For such applications, latency is less of a concern, however their owners would like to run the service in the background while displaying perhaps a progress bar for the optimization. To address these different needs, we have designed the service with both synchronous and asynchronous interfaces (e.g., for the former and latter application classes, respectively). The synchronous interface receives an optimization request and responds with the optimized itinerary. The asynchronous interface returns a callback id, and schedules the job to be executed in the background. An Azure queue is used to orchestrate the execution of the background job on the service side, and the response is in turn saved in Azure storage. When the client uses the callback id to poll for the completion of the job, MIO checks if the optimization result exists in Azure storage, and if so, sends the output back to the application.

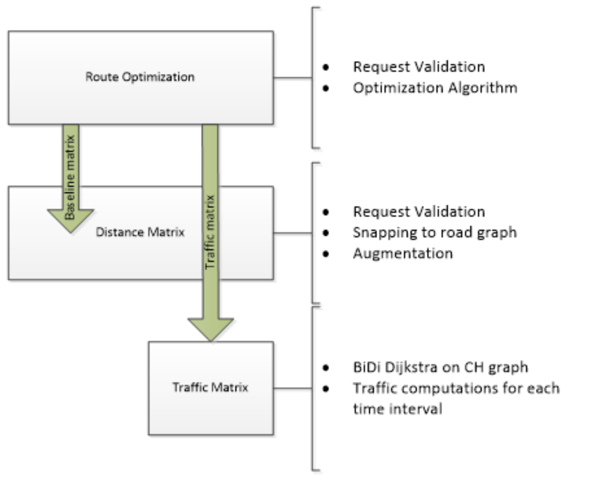

Figure 3: System Architecture

4 EXPERIMENTAL RESULTS

This section details our experimental results, starting with our setup in Section 4.1. Our first set of experiments (Section 4.2) compares MIO to baseline algorithms and instances without traffic; its main purpose is to examine the accuracy of MIO on instances with known optimal solutions. We then turn to examine instances with traffic. In Section 4.3 we highlight the cost of ignoring traffic predictions in the optimization, and finally, Section 4.4 evaluates rigorous comparisons to other baselines which demonstrate the superiority of MIO in both solution quality and running time.

4.1 Experiment setup

Throughout this section, we compare MIO to two other optimization approaches:

(1) linear programming (LP) based solutions, and
(2) heuristics based on Lin-Kernighan.

Specifically, we have two LP based approaches: with, and without traffic support. The MIP model without traffic (Section 3.2.1) can be evaluated directly, whereas our approach supporting traffic is much more complex. We attempt to solve a LP using column generation techniques [9], where each column incorporates time-dependent travel. As this is prohibitively time consuming, we have a time limit of 10 minutes, after which we use the (partial) LP solution in a MIP heuristic to obtain a feasible solution.

The Lin-Kernighan heuristic has been specifically designed to quickly find high-quality solutions to symmetric TSPs. For our experiments, we use LKH3, a state-of-the-art extension to Lin-Kernighan. It iteratively improves a feasible solution by swapping pairs of sub-tours to make a new tour, using an adaptive k-opt approach, that is, at each iteration it chooses an appropriate k number of edges to try and swap. An approximation to MIO can be expressed as a symmetric TSP by using several well-known transformations [22].

All experiments ran on a four-core Xenon 3.50Ghz 64GB RAM. Although some of the heuristics can exploit multiple cores, we restrict all algorithms to a single thread for a fair comparison.

4.2 Experiments without traffic

Our experiments begin with a single agent and a simple setup of waypoints with time-windows; the travel time between any two locations is symmetric (i.e., the travel time from A to B is the same as from B to A) and fixed. Our goal is to compare MIO's solution quality to the optimum, and its running time to LKH3. Towards that end, we use the following publicly available datasets for TSP with time-windows:

- Dumas [15] : 135 instances (20-200 waypoints)
- da Silva & Urrutia [12] : 125 instances (200-400 waypoints)

We compare MIO to both LKH3 and our MIP without traffic (Section 3.2.1), which is solved using Gurobi 5.7.2. Table 2 shows the comparison of running times and quality on the Dumas instances. LKH3, which is optimized for symmetric TSP, found the optimal solution in all instances. MIP, due to its 30 minute time limit, did not find optimal schedules for 2 instances. MIO was always able to schedule all waypoints, however not necessarily optimally (with respect to travel time). Accordingly, we show the average increase in travel time compared to optimum under MIO %-COST.

#WP	LKH3(s)	MIP(s)	MIO(s)	MIO %-COST
n20	0.02	0.28	0.84	0.0 ± 0.0 %
n40	0.02	179.20	3.48	0.8 ± 1.2 %
n60	0.07	330.80	8.66	1.8 ± 1.8 %
n80	0.08	571.69	15.05	2.0 ± 1.6 %
n100	0.11	479.97	18.92	1.9 ± 1.8 %
n150	0.40	712.48	30.42	3.4 ± 1.8 %
n200	2.06	975.96	40.43	3.5 ± 2.3 %

Table 2: Comparison of MIO, LKH and MIP on the Dumas instances. MIO %-COST is the average relative distance of MIO from the optimum, and its standard deviation.

In Table 3 we break down the running-time results based on the size of the time-window (ranging from 20 to 100 minutes). The results show that the running time for LHK3 and MIO are not significantly affected by the time-window size. However, it is important to observe that MIP is severely affected – this is a well-known drawback of the direct MIP formulation.

TW	LKH3(s)	MIP(s)	MIO(s)
w20	0.03	0.41	5.16
w40	0.06	7.47	8.41
w60	0.08	25.87	8.27
w80	0.10	620.55	10.43
w100	0.07	1,104.19	11.02

Table 3: Average running times for the Dumas dataset as a function of the time-window size.

We next evaluate experiments on the da Silva and Urrutia dataset. A summary of the results is given in Table 4. Again, LKH3 found all optimal solutions. Note that we excluded results for MIP as the bulk of the instances hit the time-limit without finding the optimal solution, and the results did not meaningfully contribute to our discussion.

#WP	LKH3(s)	MIO(s)	MIO %-COST
n200	31.18	134.98	6.8 ± 5.7%
n250	61.41	185.65	7.7 ± 6.4%
n300	116.02	200.10	8.1 ± 6.4%
n350	183.90	200.18	8.9 ± 7.2%
n400	268.09	200.23	8.4 ± 6.6%

Table 4: Results for the da Silva and Urrutia dataset. We use the same notations as in Table 2.

Overall, the results in Tables 2–4 indicate that the direct MIP formulation becomes impractical (in terms of performance) as the problem size grows – both LKH3 and MIO are orders of magnitude faster, with optimal (or near optimal) results. MIO's accuracy falls behind only by up to 9% on average compared to LKH3 (which produced the optimal solution). MIO is typically slower than the state-of-the-art LKH3, but catches up in large problem instances; overall, MIO's running times are satisfactory for our requirements. As we show later in Section 4.4, MIO beats LKH3 in more complex settings with traffic, which is what we really care about.

4.3 The significance of traffic

The Dumas and da Silva & Urrutia datasets are useful for testing TSP algorithms under the assumption of symmetric and static travel times. However, in order to examine the performance of different algorithms under traffic conditions, we have created our own dataset. We describe below its main characteristics.

The physical locations of waypoints are selected randomly from a set of cities in Washington, USA. Traffic data is obtained from our own service (see Section 3.1), and can be used for testing other algorithms as well. Waypoints have an average time-window of T_w and average dwell-time of T_d (both in minutes), where T_w and T_d are configurable parameters; in the bulk of the experiments, we use $T_w = 560$ and $T_d = 5$. We created a total of 70 instances with the above characteristics, with the number of waypoints ranging from 20 to 140. We henceforth refer to this dataset as *Traffic-1*.

Before comparing the performance of the different algorithms in traffic conditions, we first wish to examine the significance of accounting for traffic. To that end, we run MIO with traffic predictions, and compare the results to settings where we replace the traffic predictions with static travel times. In particular, we perform separate comparisons with two variants of traffic-agnostic settings: (i) Free-Flow (FF) – using the traffic-free travel times (which lower bound the real travel-times); (ii) Maximum travel time (MaxT) – using the maximum travel-times during the day (which upper bound the real travel times). For simplicity, we perform the experiments for one agent with a shift length of 14 hours.

Running MIO with FF conditions might actually schedule a waypoint that violates its time-window "in practice", that is, the agent would arrive late when considering the predicted traffic. When such a violation occurs, we consider that underlying waypoint "dropped" (i.e., unfulfilled).

Table 5 summarizes the results for *Traffic-1*. Since most of the connections between cities in Washington state pass through rural areas or highways, the variability in travel times is not large (up to 6% between average and maximum travel time). Still, the percent of drops is non-negligible (up to 8.7 percent depending on the number of waypoints).

#WP	FF(#)	Avg #drops	Avg drop %	Max drop %
21	20.8	0.2	1.0%	5.0%
41	38.8	1.4	3.7%	7.9%
61	48.3	2.9	6.0%	8.3%
81	57.1	2.8	4.9%	8.5%
101	61.9	2.8	4.6%	6.5%
121	67.0	2.7	4.0%	6.0%
141	67.7	4.8	7.0%	8.7%

Table 5: Drop statistics for *Traffic-1* under the FF setting, where travel-times have relatively low variability. FF(#) represents the average number of waypoints that are planned to be fulfilled by the agent (some would have to be dropped due to time-window violations).

To further understand the effect of using optimistic travel times, we repeated the above experiments (FF setting) on a dataset (80 instances) with waypoints corresponding to an urban area around Seattle. Naturally, the travel time variability here is higher (up to 30% between average and max). We also decreased the average time window to five hours ($T_w = 300$). We refer to this dataset as *Traffic-2*. The results are summarized in Table 6. We observe up to 38% of dropped waypoints, which would be unacceptable in many logistics contexts.

#WP	FF(#)	Avg #drops	Avg drop %	Max drop %
21	17.6	2.0	11.7%	28.6%
41	28.1	5.3	18.9%	25.0%
61	33.4	7.9	24.2%	38.5%
81	39.5	8.3	21.2%	31.4%

Table 6: Drop statistics for *Traffic-2* with the FF setting; where travel-times have high variability. FF(#) represents the average number of waypoints that are supposed to be reached by the agent.

To conclude this subsection, we examine the consequences of using the maximum travel times instead of traffic predictions (MaxT setting). Here, the scheduled waypoints will never violate their time-window because we use conservative estimates for travel time. However, the issue now is that a schedule that accounts for the true travel times can visit more waypoints; in service contexts, that corresponds to performing more work. The results on the *Traffic-2* dataset are shown in Table 7. We observe that using conservative travel times instead of predicted traffic results in up to 17% less visited waypoints.

#WP	MIO(#)	MaxT(#)	Avg Gain	Max Gain
21	19.0	18.6	1.9%	10.5%
41	31.0	28.7	8.1%	11.5%
61	37.2	34.4	8.4%	13.3%
81	43.3	39.3	10.5%	16.7%

Table 7: Fulfillment statistics for the MaxT setting; in these experiments travel-times have high variability. MIO(#) and MaxT(#) represents the average number of waypoints that are fulfilled when running with traffic prediction and in the MaxT setting, respectively.

In summary, we have showed here that being agnostic to traffic may result in substantial business consequences – either not satisfying scheduled work (FF) or doing less work than possible (MaxT).

4.4 Experiments with traffic

Our last set of experiments compares the different algorithms under traffic conditions. Before presenting our results, we briefly describe some adjustments we had to make for the MIP and LKH3 approaches, to accommodate traffic. For the MIP, the direct formulation (described in Section 3.2.1) does not scale with traffic. Thus, we have used instead the linear relaxation approach described in Section 4.1; and use a time-limit of ten minutes. LKH3 does not support time-dependent travel time. Consequently, to avoid time-window violations, we had to use the maximum travel times between any two locations. Furthermore, since LKH3 does not support dwell times, we incorporated the dwell time at each destination into the travel time away from the waypoint.

In our experiments, we use the *Traffic-1* dataset (Section 4.3). The results for a single agent are summarized in Table 8. The MIP running-time has been omitted, since for every instance, it reached its time-limit of 600 seconds. We observe that MIO is the best algorithm in terms of both running time and quality of the solution (number of waypoints).

#WP	LKH3(#)	MIO(#)	MIP(#)	LKH3(s)	MIO(s)
21	21.0	21.0	21.0	1.87	2.28
41	39.9	39.9	39.4	30.27	7.93
61	41.4	49.9	47.5	43.62	21.31
81	53.6	58.2	53.9	68.62	43.21
101	57.5	61.9	55.8	86.97	59.40
121	42.8	67.0	58.4	93.30	71.40
141	43.8	69.7	59.7	108.78	83.40

Table 8: Comparisons of quality and performance on instances with traffic (single agent). We use the notation Alg(s) for the average running time of Alg (in sec), and Alg(#) for the average number of visited waypoints.

We conclude our experiments by evaluating multiple agents. As we have already established the superiority of MIO with traffic, we will focus on its performance as a function of the number of agents and waypoints. Our setup for multi-agent experiments is as follows. The agents are "symmetric", that is, they start and end their shift at the same single location (e.g., a depot or hub), and have the same shift length. We use the *Traffic-1* dataset for waypoints, which are shared between the agents. That is, to satisfy a waypoint, only one of the agents must visit and remain for the corresponding dwell time.

As a first proxy to the quality of solution, we expect that more waypoints would be fulfilled as the number of agents increases. This is indeed verified in Table 9. Note that increasing the number of agents does not necessarily lead to fulfilling all the waypoints. This is because there are some waypoints which are not reachable by the shift time constraint. In other words, depending on the instance size, we reach a saturation point where additional agents are not useful with respect to waypoint satisfaction (e.g., eight agents for 141 waypoints).

AGT	#21	#41	#61	#81	#101	#121	#141
1	0.0	1.1	19.6	27.4	43.5	78.2	97.2
2	0.0	0.0	1.1	7.8	18.9	31.7	49.2
4	0.0	0.0	0.2	0.1	0.7	5.8	14.0
6	0.0	0.0	0.2	0.1	0.1	0.9	1.8
8	0.0	0.0	0.2	0.1	0.1	0.7	0.6
10	0.0	0.0	0.2	0.1	0.1	0.7	0.6
12	0.0	0.0	0.2	0.1	0.1	0.7	0.6

Table 9: Multi-agent experiments on the *Traffic-1* dataset. For each number of agents (AGT) and waypoint combination, we show the average number of unfulfilled waypoints.

As part of our quality study, it is interesting to observe what happens with other metrics of interest when we reach a saturation point in terms of waypoint fulfillment. In Table 10, we zoom in on the larger instances and report the total travel-time, which is an important measure for system cost. Observe that the average decreases with the number of agents, demonstrating that MIO can find more efficient routes even when no additional waypoints can be visited.

AGT	#121(min)	#141(min)
8	1723.1	2035.0
10	1696.5	1983.2
12	1692.5	1925.3

Table 10: Total travel-time averages (in minutes) as a function of the number of agents (AGT) and instance size.

We conclude this section by examining the running times for the above experiments. One may expect the running time would be impacted as the number of agents increase. However, the results in Table 11 indicate that the dominant factor is the problem size (number of waypoints). In fact, the worst running times are obtained for the largest instances (141 items) with a *single* agent. This can be explained by recalling that this combination is worst with respect to fulfilled waypoints (See Table 9) which result in challenging conditions for the optimization. The fact that running time is not affected by a larger number of agents is yet another positive indication for the applicability and scalbility of MIO.

AGT	#21	#41	#61	#81	#101	#121	#141
1	1.8	7.9	21.3	43.2	59.4	71.4	183.4
2	1.8	7.8	17.2	32.5	49.8	69.4	83.4
4	1.8	5.0	10.3	19.2	32.1	50.4	73.2
6	2.1	4.8	8.4	14.3	23.5	36.8	55.0
8	2.3	5.1	8.2	13.2	19.7	29.1	42.1
10	2.7	5.6	8.8	13.1	18.9	27.0	36.2
12	3.2	6.4	9.8	13.6	18.9	25.2	33.3

Table 11: Running time averages (in seconds) for multiple agents for different instance sizes.

5 RELATED WORK

Scope. Most literature relating to NP-hard vehicle routing problems (VRP) makes the assumption that the travel time between locations is given as input [24, 28, 30]. Often these problems have a fixed set of locations (i.e., the distance matrix can be cached), but in general, the effort to optimize the VRP is so significantly difficult that the shortest-path travel time calculations are ignored. We in contrast pay close attention not only to the route optimization, but also to the efficient calculation of travel times. Our system provides a complete end-to-end service that supports waypoints anywhere across the globe, and aims to give a high-quality solution within a very short time frame. Inside our service we have implemented the necessary distance calculations between locations. Importantly, we incorporate traffic, which is by itself algorithmically non-trivial.

Related commercial products. As an enterprise cloud service, MIO can be compared to other commercial offerings, such as Microsoft Dynamics 365 for Field Service (Resource Scheduling Optimization) [2], Routific Routing Engine [6], Google Maps Directions [5], and TomTom Routing [7]. Such commercial offerings typically do not make public the details of the algorithms, and/or use some algorithmic components as black box (e.g., the travel-time calculations). To the best of our knowledge, this paper is the first to report the full algorithmic details of an end-to-end cloud service that supports traffic-aware itineraries for multiple agents.

Time-dependent shortest-path. For an overview of query acceleration techniques for time-dependent shortest path algorithms we refer to [14]. An approach similar to our traffic matrix algorithm, [19] also uses contraction hierarchies with a modified Dijkstra to calculate time-dependent shortest travel for a single origin to multiple destinations.

The technician routing and scheduling problem (TRSP). As mentioned in Section 3.2, MIO solves the TRSP, which is often considered an extension to the vehicle routing problem – the key difference being the dwell time at each location, which can be different for each agent (i.e., based on skill/experience). Initially TRSP[16] was based on a real problem in a telecommunications company, assigning tasks to teams of technicians and matching skill requirements / levels. This paper was popularized as part of a 2007 challenge set by the French Operation Research Society [1]. Although this inspired many papers, none considered the routing aspect to the problem, including [11] which used an ALNS based approach.

ALNS was introduced in [27] where it was described as a general purpose approach for solving vehicle routing problems (i.e., with time windows, capacity, multi-depot, etc). It has since been used in many papers, including several directly solving the TRSP [23, 26].

In the literature, the TRSP has been extended even further by considering uncertainty in service and travel times [17, 25, 29], and experience-based learning [10]. In practice, [31] describes how they successfully combined GIS with TRSP (distance matrix created using ESRI products) and solving a tabu based heuristic.

6 CONCLUSION

In this paper, we describe Multi-Itinerary Optimization (MIO) – a Bing Maps service that has been recently released worldwide. MIO gets as input a list of agents with start/end times and locations, and desired waypoints with time-windows and dwell time. In turn, MIO outputs a schedule for each agent. MIO builds on a hierarchy of algorithms, including shortest-path mechanisms for travel-time calculation and a carefully designed heuristic for route optimization. Importantly, our algorithms account for *traffic* predictions, which are significant especially for logistics scenarios in urban areas. Our experiments show that MIO achieves a sweet point between accuracy and performance – it produces near-optimal schedules with running time significantly better than both a state-of-the-art heuristic (LKH) and a rigorous Integer-Programming based approach.

We see MIO as an attractive tool that can be used to automate relevant logistics operations of numerous businesses and organizations. We are planning to extend the scope of MIO; future directions include incorporating capacity constraints for vehicles, supporting pickup and delivery scenarios, and more generally, settings where there is a dependency between tasks.

REFERENCES

[1] 2007. What Is the ROADEF Challenge? http://www.roadef.org/challenge/2007/en/.

[2] 2018. Resource Scheduling Optimization (RSO). https://docs.microsoft.com/en-us/dynamics365/customer-engagement/field-service/rso-overview.

[3] 2019. Azure pricing. Retrieved June 7, 2019 from https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/

[4] 2019. Bing Maps Multi-Itinerary Optimization. https://docs.microsoft.com/en-us/bingmaps/rest-services/routes/optimized-itinerary.

[5] 2019. Calculate directions between locations using the Google Maps Directions API. https://developers.google.com/maps/documentation/directions/intro.

[6] 2019. Routific Engine API. https://docs.routific.com/.

[7] 2019. TomTom Routing API. https://developer.tomtom.com/routing-api/routing-api-documentation-routing/calculate-route.

[8] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Experimental Algorithms (Lecture Notes in Computer Science)*, Panos M. Pardalos and Steffen Rebennack (Eds.). Springer Berlin Heidelberg, 230–241.

[9] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. 1998. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research* 46, 3 (June 1998), 316–329. https://doi.org/10.1287/opre.46.3.316

[10] Xi Chen, Barrett W. Thomas, and Mike Hewitt. 2016. The Technician Routing Problem with Experience-Based Service Times. *Omega* 61 (June 2016), 49–61. https://doi.org/10.1016/j.omega.2015.07.006

[11] Jean-François Cordeau, Gilbert Laporte, Federico Pasin, and Stefan Ropke. 2010. Scheduling Technicians and Tasks in a Telecommunications Company. *Journal of Scheduling* 13, 4 (Aug. 2010), 393–409. https://doi.org/10.1007/s10951-010-0188-7

[12] Rodrigo Ferreira da Silva and Sebastián Urrutia. 2010. A General VNS Heuristic for the Traveling Salesman Problem with Time Windows. *Discrete Optimization* 7, 4 (Nov. 2010), 203–211. https://doi.org/10.1016/j.disopt.2010.04.002

[13] Brian C Dean. 2004. Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. *Rapport technique* (2004), 13.

[14] Daniel Delling and Dorothea Wagner. 2009. Time-Dependent Route Planning. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–230. https://doi.org/10.1007/978-3-642-05465-5_8

[15] Yvan Dumas, Jacques Desrosiers, Eric Gelinas, and Marius M. Solomon. 1995. An Optimal Algorithm for the Traveling Salesman Problem with Time Windows. *Operations Research* 43, 2 (April 1995), 367–371. https://doi.org/10.1287/opre.43.2.367

[16] Pierre-Francois Dutot, Alexandre Laugier, and Anne-Marie Bustos. 2016. Technicians and Interventions Scheduling for Telecommunications. (2016), 7.

[17] F. Errico, G. Desaulniers, M. Gendreau, W. Rei, and L.-M. Rousseau. 2016. The Vehicle Routing Problem with Hard Time Windows and Stochastic Service Times. *EURO Journal on Transportation and Logistics* (Nov. 2016), 1–29. https://doi.org/10.1007/s13676-016-0101-4

[18] Luca Foschini, John Hershberger, and Subhash Suri. 2014. On the Complexity of Time-Dependent Shortest Paths. *Algorithmica* 68, 4 (April 2014), 1075–1097. https://doi.org/10.1007/s00453-012-9714-7

[19] Robert Geisberger. 2010. Engineering Time-Dependent One-To-All Computation. *arXiv:1010.0809 [cs]* (Oct. 2010). arXiv:cs/1010.0809

[20] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms*, Catherine C. McGeoch (Ed.). Vol. 5038. Springer Berlin Heidelberg, Berlin, Heidelberg, 319–333. https://doi.org/10.1007/978-3-540-68552-4_24

[21] Edward He, Natashia Boland, George Nemhauser, and Martin Savelsbergh. 2019. Computational Complexity of Time-Dependent Shortest Path Problems. *Optimization Online* (Feb. 2019), 12.

[22] Keld Helsgaun. 2017. *An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems.* https://doi.org/10.13140/RG.2.2.25569.40807

[23] Attila A. Kovacs, Sophie N. Parragh, Karl F. Doerner, and Richard F. Hartl. 2012. Adaptive Large Neighborhood Search for Service Technician Routing and Scheduling Problems. *Journal of Scheduling* 15, 5 (Oct. 2012), 579–600. https://doi.org/10.1007/s10951-011-0246-9

[24] Chryssi Malandraki and Mark S. Daskin. 1992. Time Dependent Vehicle Routing Problems: Formulations, Properties and Heuristic Algorithms. *Transportation Science* 26, 3 (Aug. 1992), 185–200. https://doi.org/10.1287/trsc.26.3.185

[25] Douglas Moura Miranda and Samuel Vieira Conceição. 2016. The Vehicle Routing Problem with Hard Time Windows and Stochastic Travel and Service Time. *Expert Systems with Applications* 64 (Dec. 2016), 104–116. https://doi.org/10.1016/j.eswa.2016.07.022

[26] V. Pillac, C. Guéret, and A. L. Medaglia. 2013. A Parallel Matheuristic for the Technician Routing and Scheduling Problem. *Optimization Letters* 7, 7 (Oct. 2013), 1525–1535. https://doi.org/10.1007/s11590-012-0567-4

[27] David Pisinger and Stefan Ropke. 2007. A General Heuristic for Vehicle Routing Problems. *Computers & Operations Research* 34, 8 (Aug. 2007), 2403–2435. https://doi.org/10.1016/j.cor.2005.09.012

[28] Marius M. Solomon and Jacques Desrosiers. 1988. Survey Paper—Time Window Constrained Routing and Scheduling Problems. *Transportation Science* 22, 1 (Feb. 1988), 1–13. https://doi.org/10.1287/trsc.22.1.1

[29] Duygu Taş, Nico Dellaert, Tom van Woensel, and Ton de Kok. 2013. Vehicle Routing Problem with Stochastic Travel Times Including Soft Time Windows and Service Costs. *Computers & Operations Research* 40, 1 (Jan. 2013), 214–224. https://doi.org/10.1016/j.cor.2012.06.008

[30] Hamza Ben Ticha, Nabil Absi, Dominique Feillet, and Alain Quilliot. 2018. Vehicle Routing Problems with Road-Network Information: State of the Art. *Networks* 72, 3 (2018), 393–406. https://doi.org/10.1002/net.21808

[31] Don Weigel and Buyang Cao. 1999. Applying GIS and OR Techniques to Solve Sears Technician-Dispatching and Home Delivery Problems. *Interfaces* 29 (1999), 112–130. https://doi.org/10.1287/inte.29.1.112