OPEN DOOR LOGISTICS

email: info@opendoorlogistics.com
phone: +44 (0) 208 144 4009

Intelligent software for vehicle routing & territory management

How to build a real-time vehicle route optimiser

Philip Welch, Ph.D.

Open Door Logistics - www.opendoorlogistics.com

June 2017

Table of Contents

Abstract

We describe the development of a commercial real-time logistics management system for on-demand services - last mile deliveries, taxis and mobile workforce scheduling. We highlight the research and technical challenges that arose during the development, so an interested reader can (a) avoid making costly errors if they chose to develop their own system and (b) weigh the pros and cons of developing an in-house system or sourcing an external one.

Introduction

Over the last couple of years Open Door Logistics has been developing ODL Live, a low-cost (flat fee) cloud-based planning engine for real-time route optimisation and delivery management. With real-time vehicle scheduling, some or all future jobs for the current shift are unknown. Appointment scheduling – the process of offering service time slots to customers and then confirming their selection, is also a real-time planning problem as the booking of one time slot affects the availablity of others. Real-time route scheduling problems, usually called 'dynamic vehicle routing problems' in the research literature, appear in a number of industries, for example:

- **On-demand deliveries -** takeaway food, drinks, laundry collection etc.
- **Taxi services**, both for general public and specialised services for the elderly, less-abled or patients.
- **Field force optimisation**, repair technicians, surveyors, engineers or any other workers who drive significant distances and either (a) accept new jobs on the same day or (b) need route-efficient efficient appointment booking.
- **Time slot generation for home delivery networks**, i.e. offering time-slots to customers browsing an e-commerce site, based on planned routes.

We started developing ODL Live by integrating two open source libraries for travel time estimation and static vehicle routing - Graphhopper and Jsprit respectively (now merged into just Graphhopper). A 'static' vehicle routing problem is the traditional academic problem where jobs and drivers are 100% known prior to the delivery period starting, and a single plan can therefore be generated at the start of the period. We had previously integrated these libraries into ODL Studio. ODL Studio is our open source desktop-based application for static vehicle routing problems (i.e. not real-time), designed for planning daily routes at the start of the day. Re-using these libraries for a real-time engine was therefore a logical choice for us.

Graphhopper and Jsprit provided a solid foundation but also lacked key features which proved essential for real-time route planning, which we discuss in-detail in the following sections. Whilst extending these libraries and building a dynamic logistics management system we had to overcome many different technical and research challenges. We group these technical challenges - and their associated solutions - into five loose categories:

1. 100% incremental processing ('need for speed')

2. Mapping a real-time to a static problem ('the day the earth stood still')

3. Simulating live problems ('the time machine')

4. Accurate travel time estimation using free data ('an unexpected journey')

5. Adapting real-time route planning to cloud-based architecture ('cloud atlas')

The ODL Live feature list discussed here relates to the current production release as per June 2017. Many other features are in active development - consult the road-map (at the end of this document), for features that are coming in the near-future.

100% incremental processing ('need for speed')

A real-time planning engine for last mile deliveries takes a constant feed of data - GPS from mobile devices, new jobs coming in from order processing systems, delivery arrival or completion events etc. The schedule - ordered list of planned deliveries for each vehicle - must constantly adapt to stay efficient.

Typically there are two phases to solving a vehicle routing problem:

1. Generate a matrix of travel times and distances between all relevant locations.

2. Use a solver method to generate a plan.

A naive implementation of a real-time scheduler would recalculate the matrix and plan from scratch whenever new data is available, but for most practical applications this would be far too slow. In a real-time dispatching scenario - for example hot foods delivery - waiting a minute or two for the planner to catch-up with a new job before it could be dispatched is not acceptable. Similarly on-the-fly generation of delivery timeslots for a customer browsing your e-commerce site could never be powered by a 'recalculate all' approach.

When we consider the dependence on problem size, a 'recalculate all' approach quickly becomes intractable as the number of jobs increases. Live data updates arrive more frequently but, as the problem is larger, good solutions take longer to generate when you start the optimisation from scratch. For an insertion-based heuristic solver such as Jsprit, the processing time required to perform a single search iteration scales with at least the square power of the number of jobs, and in many circumstances would scale with the cubed power. These are state-of-the-art solvers, with much better scaling than mathematical programming approaches, but performance still degrades rapidly with increased problem size. Pretty soon a solver used in 'recalculate all' mode wouldn't be able to generate new high-quality solutions quick enough.

Clearly the only viable technical solution is an *incremental* optimiser, which reuses previous calculations wherever possible. ODL Live implements this in a number of ways:

Preloaded data. A model is always ready pre-loaded in server memory (e.g. the road network data, travel time matrix, solution state are always ready) and the optimiser is always running, constantly refining its routing plans. Memory consumption increases but disk I/O is no longer a factor.

Incremental matrix calculations. The matrix is updated incrementally. The base road network travel calculations are performed using the Graphhopper library, which uses the *Contraction Hierarchies* algorithm (Geisberger et al. 2008) to speedup traditional Dijkstra driving direction calculations. Contraction hierachies within Graphhopper are however geared towards finding single A to B route pairs, not filling a matrix of travel times to and from many locations. Knopp et al. 2007 describe a method to efficiently calculate matrices for a similar algorithm to Contraction Hierarchies, where the shortest path trees from/to each location are cached and reused. We use a suitably adapted implementation of this method and cache the trees in a compact form, then use them in an auxiliary algorithm to incrementally fill the matrix when new A to B travel times and distances are required.

Reusing previous routing plan. The solver itself is also incremental. The Jsprit library solves vehicle routing problems but (at the time of writing), cannot be run incrementally, starting its search from a previous solution when the input data has changed. We use a modified and extended version of Jsprit which can restart its search method from a previous solution, but adjust accordingly when the solution is no longer feasible - for example when constraints such as vehicle capacity, skills or delivery time windows make parts of the solution infeasible.

Technically Jsprit uses a search method called ruin and recreate which is guided by another search method (or 'metaheuristic'), which for practical purposes is equivalent to simulated annealing. In our incremental optimisation framework, we run short bursts of optimisation on a 'fixed' vehicle routing problem, then repoll for changes to the input data, update the matrix and run the next burst starting with the previous solution. Simulating annealing (Khachaturyan et al. 1979) is unsuitable for running short repeated bursts of optimisation on a changing problem as it uses an annealing schedule - a slow decrease in the probability of accepting worse solutions. A short burst of optimisation might only last a couple of iterations whereas an annealing schedule would need to act over hundreds of iterations, so an annealing schedule cannot be used within a burst. Moreover it is inappropriate to use the same annealing schedule across multiple optimisation bursts as the underlying vehicle routing problem has changed, meaning the solution fitness landscape has also changed and the principles behind simulated annealing no longer apply. Instead we use a parallel multi-start algorithm of our own devising as similar algorithms have been shown to be effective for dynamic optimisation problems (e.g. Li 2011). The rationale is that maintaining parallel solutions within the search increases the likelihood of having a solution which is still suitable - with minimal updates - when the underlying problem changes. Multi-start also helps to prevent the solver becoming stuck in local minima when the underlying data changes significantly.

Our incremental framework allows us to efficiently run the optimiser in short bursts of a few hundred milliseconds up to a couple of seconds, for problems with up to 1000 live jobs. It therefore allows a near-instantaneous response to new data. Larger problems may need longer burst durations, or to be split into suitable independent sub-problems (e.g. by defining delivery territories).

Mapping a real-time to a static problem ('the day the earth stood still')

Pillac et al. 2013 discuss two different approaches to solving dynamic / real-time vehicle routing problems:

1. *Periodic reoptimisation*, where the real-time routing problem is translated into a static problem and solved periodically (presumably from scratch each time), and:

2. *Continuous reoptimisation* - where the current best solution(s) are updated as the input data changes.

Although technically our approach falls under the continuous optimisation category - as we reuse and update the solution from the preceding optimisation burst - each individual burst optimises the routing problem as a static routing problem. We therefore assume the problem is fixed for the duration of the burst and have to map our 'real-time' problem to the fixed static problem.

The basic conversion from a real-time to a static problem - converting GPS traces, vehicle state, stop arrival and completion events, pending jobs etc. into vehicle starting points, fixed costs, jobs - is a little complex but not insurmountable. We did however find that the resulting static problem contained unusual constraints and costs and other features not commonly found in traditional static vehicle routing problems. Solving these required the implementation of various custom constraints within Jsprit and some modifications to Jsprit itself. We present a selection of these features in the following list.

- **Soft end time windows**. In a real-life real-time vehicle routing problem soft end time windows - penalising but still allowing a late arrival - are unavoidable. Minor delays will often put deliveries slightly off schedule and simply not serving a delivery that will be one second late is not an option.
- **Handling manual overrides to the plan**. Users can override the automated scheduler's chosen dispatches. Particularly in the case of pickup-delivery problems (e.g. for couriers), this can result in an infeasible static problem - where the user's manual override breaks quantity constraints. We had to re-write the quantity constraint model within Jsprit to a more refined version which handles this situation gracefully.
- **Spreading work out across the fleet**, to maximise the chances of servicing future jobs. If certain drivers are filled to capacity, then any new incoming jobs which require those specific drivers cannot be served. Balancing work between vehicles is therefore surprisingly important for a real-time problem.
- **Prioritising on-board jobs, locking deliveries but not collections**.

Whilst not strictly part of the 'static' vehicle routing problem, several other 'algorithmic' issues croped up related to the optimisation model, which we had to solve. Examples include:

- **Deciding rules for job dispatching**. Should a job be dispatched to a driver as soon as it becomes known or should it be buffered for a while? Clearly the driver needs a small buffer of dispatched job(s) which are 'locked down' (e.g. their next one or two stops are fixed), as communication delays may occur when transmitting new dispatches. A small buffer is good as it gives the optimiser more flexibility, but how small a buffer is feasible?
- **Determining when new sub-optimal solutions are actionable**. Given the constant feed of incoming data, when many jobs are added at once, or perhaps a vehicle gets significantly delayed, the current plan will no longer be efficient. For larger problems - particularly if you're running many problems in-parallel on the same server, the optimiser might take a while to 'catch-up' and generate a new high-quality solution. Should intermediary quality solutions be presented to the user or not? Should we delay a pending dispatch, particularly one that's

already late, in-case the optimiser decides five seconds later that the dispatch is no longer the best one to make?

Simulating live problems ('the time machine')

One of the key issues we encountered in developing a real-time delivery management engine was how to test it in a realistic way, ensuring modelling of historic data gives an accurate evaluation of the system's future performance? The key point about a real-time optimiser is that some or all of the jobs are unknown at the start of the driver shifts and new jobs become known when drivers are already serving existing jobs. It is impossible to generate a single plan at the start of the day for all jobs. Offline modelling using a single fixed plan for all jobs would grossly overestimate the routing efficiency, as it assumes the optimiser can see the future.

Instead we developed a discrete event simulation tool to allow us to accurately model historic data, without assuming future knowledge. The simulator wraps around the planning engine and simulates the evolution of the planning over the entire planning period (e.g. day, week etc). At the start of the planning period, only a few jobs are known and the simulator plans just those jobs. Separately the simulator models the vehicles as 'agents' which depart for stops, arrive and complete them, whilst generating GPS traces. As more jobs become known, the simulator re-plans taking into account the positions of the simulated vehicles and what deliveries are already complete. Within the simulation, the delivery plan evolves continually over the course of the day as the situation changes.

The simulator lets us accurately model historic scenarios for real-time planning problems across a number of different industries (e.g. taxis, service engineers, on-demand deliveries), realistically gauging the performance of ODL Live. We consider the ability to perform correct, accurate offline modelling to be an essential part of any dynamic optimiser system.

Accurate travel time estimation using free data ('an unexpected journey')

Pricing for other commercial vendors of dynamic vehicle routing systems is normally either per driver (e.g. USD $10 or $20 per driver per month) or per job (e.g. USD $0.20 per job). One of our key aims is to instead retain a low-cost flat subscription fee independent of the number of jobs or vehicles optimised. We ensure this is financially viable by (a) by separating out the cost of the Amazon Webservices (AWS) virtual servers to run ODL Live and letting the client size the server to their own needs and (b) using free crowd-sourced OpenStreetMap (OSM) road network data. We work the OSM data - and other free data sources - as much as possible to get accurate travel times at minimum cost. Commercial mapping data providers - e.g. HERE - would typically charge per vehicle, making a flat subscription fee for ODL Live impossible. Although OSM data may be less accurate than commercial data, we believe this can be mitigated, to the point it is unlikely to be an issue, by taking the time to appropriately calibrate the data.

The Graphhopper library provides estimates of travel times using OSM road network data. However, the current out-of-the-box version (as per June 2017) combined with the Jsprit library does not take traffic into account. We have therefore built traffic modelling functionality on top of these libraries. The integration of traffic effects into a live route planning and job dispatching system is hard from a development point of view, but it can be made simpler by separating the work out into three separate stages. With each stage that is integrated into the system, the scheduling results become more precise. As of June 2017, the production version of ODL Live models these first two stages and the third is in active development.

Stage I - predictable time-independent traffic for city centres. Roads in urban areas, particularly the centres of large cities, will be on-average a lot slower than roads in suburban or rural areas, even if they have the same legal speed limits.

Our SpeedRegions project, built on-top of Graphhopper, lets road speeds be fine-tuned to high geographic accuracy (e.g. cells of 100 metres x 100 metres). To enable quick rebuilding of the routing graph with speed regions, we developed a specialised quadtree-like lookup algorithm to quickly identify the speed region a particular stretch of road falls within. We also developed a customised version of our ODL Studio desktop app for analysts and planners to fine-tune road speeds by postcode or zip and compare travel times to historic journeys.

Stage II - predictable time-dependent traffic (a.k.a. rush hours). Urban areas will have rush hours - periods of the day where heavy commuter traffic causes delays. Rush hour modelling was integrated with minimal changes to Graphhopper but large changes to Jsprit - we had to re-write the insertion heuristics to properly evaluate adding jobs to existing routes with rush hours and soft end time windows. Our rush hour implementation is based around distinct periods (e.g. 8:00 to 9:30) having their own speed profiles. Journeys that span two or more 'speed periods' have their travel time correctly calculated using a closed-form time-dependent mathematical equation (e.g. departing between 9:04 and 10:21 the travel time in seconds will be $20 + 31\ t$). Closed form equations let us quickly calculate travel time for a given departure time, minimising the additional CPU burden. They also make the development of future planned speed-ups to the insertion processing more tractable. The equations are carefully formulated to ensure they don't break any assumptions within the Jsprit search algorithm (e.g. the triangle inequality / no overtaking yourself by leaving later). The equations are derived on-the-fly as needed and cached.

Speed periods can be customised as desired based on a driver team's knowledge. Intermediary periods (e.g. 'early rush hour') can also be defined, blending speeds from multiple time profiles within a built Graphhopper road network graph. Daylight savings time changes are also fully accounted for using speed periods.

Stage III - unexpected delays, particularly congestion caused by accidents or temporary road closures, requiring real-time traffic data feeds. Integrating real-time traffic updates is currently work in-progress and we will report more when it is complete. We plan to use real-time data feeds on specific traffic incidents that are available for free or at low cost (e.g. see http://www.trafficengland.com/services-info). These feeds and the OSM road network graph will be incorporated into a model which can extrapolate a reasonable approximation of the impact of traffic incidents further across the road network - for example on parallel alternate routes, similar to that investigated by Wirtz et al. 2004.

Adapting real-time route planning to cloud-based architecture ('cloud atlas')

The de-facto standard for modern IT systems is a cloud-based architecture comprised of a collection of loosely coupled services (or microservices), each of which can be independently updated or replaced. As a real-time optimiser is always running, potentially 24-7, it must be resilient to hardware failure, making cloud-based server virtualisation with automated failover the ideal (or possibly the only) hosting solution. ODL Live is deployed as a microservice on AWS and is compatible with other cloud hosting providers.

Developing a RESTful webservice API for ODL Live, with model objects passed as JSON, was relatively straightforward. However getting ODL Live - and any other real-time scheduler - to 'play nicely' in the cloud came with a number of challenges. We outline these in the next paragraphs.

Data syncing strategy. A real-time scheduler takes a constant data feed from a client's back-end systems - e.g. their order processing systems and their server-side system corresponding to their mobile driver app. With multiple data updates per second, how do we keep the state of the ODL Live model (jobs, drivers locations etc.) properly in-sync with a client's systems, whilst minimising the incoming data bandwidth? Again the answer was to use incremental processing. We engineered our RESTful API to support both (a) updating individual jobs or vehicle records for small, frequent updates such as GPS location and (b) refreshing all data at once but less frequently, e.g. once a minute, to ensure ODL Live's state correctly mirrors the clients backend systems. To avoid the client having to constantly repoll for updated routes, webhooks call back to the client's system and tell it when planned routes have changed.

CPU resource balancing across multiple vehicle routing problems. It is common for organisations running on-demand deliveries, people transportation (i.e. taxis) or similar across multiple cities to have different driver teams and separate management structures for each city. The performance of an optimiser degrades with at least the square of the number of live jobs in the vehicle routing model. This makes a 'divide-and-conquer' approach - where different cities are treated as independent problems - ideal for applications where different cities are served independently. Unfortunately, this brings about a further complication, how to intelligently allocate CPU resources when running multiple vehicle routing models in-parallel on the same server?

To solve this we created an 'intelligent CPU agent' which assigns CPU to routing models based on their relative need, so larger models or rapidly changing models receive more CPU time than small or seldom-changing ones. The determination of 'relative need' turned out to be difficult as it required an estimate of solution quality compared to the (unknown) optimal solution; we approached this using statistical techniques. Using the 'intelligent CPU agent' a relatively modest virtual server with a fixed rental cost will comfortably run many routing models in parallel (greatly exceeding the number of CPU cores on the server).

Minimising database I/O. ODL Live stores both its current solution and input model data in a managed, automatically replicated database. Having a managed database backend ensures that the solver can come back up again correctly - and without loss of best solution - should the virtual server failover or otherwise reboot. This itself brings about another problem - if the solution changes every 100 milliseconds the amount of writes to the database could create a major bottleneck (and incur significant AWS data transfer charges). For this reason we developed a solution update throttling technique to put the brakes on excessive solution writes to the database but still prioritise those writes - e.g. acceptance of new job - which are time-crucial. We also implemented data caching strategies to minimise data throughput between the optimiser and managed database.

Road-map of future developments

We have a number of different developments planned for the live planning engine in the near-future, based on the needs of our key clients. In addition to the integration of live traffic data as discussed in the preceding section, we plan to develop tools for the automated calibration of speeds in the built road network graph. Several of our clients have historic data containing A to B travel times for their fleets, numbering tens or hundreds of thousands of journeys. We plan to mine this data to automatically fine-tune the road network speeds across the day (e.g. early morning, rush hour, etc.) and generate predicted journey times with an accuracy as good as, or exceeding commercial map data. This will require some degree of inference, as the number of historic journeys although great, will not give complete coverage of the road network. By assuming drivers usually take the quickest routes, it should be possible to make inferences on road speeds for parts of the network nearby to, but not covered by the historic journeys. The exclusion of a nearby road link from a historic route may place an upper bounds on its speed.

The integration of demand prediction into ODL Live has been a long-term goal since development first started several years back. The engine is now reaching a level of maturity - in terms of available functionality - where demand prediction becomes a logical next development step. A sophisticated integration of spatio-temporal demand predictions into the routing engine should have two key benefits (a) planning routes with known jobs to better serve future unknown jobs and (b) opportunistic repositioning - placing drivers awaiting new jobs in locations where new jobs are likely to occur.

References

Geisberger R., Sanders P., Schultes D., Delling D. (2008) "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: McGeoch C.C. (eds) Experimental Algorithms. WEA 2008. Lecture Notes in Computer Science, vol 5038. Springer, Berlin, Heidelberg

Khachaturyan A., Semenovskaya S., Vainshtein B. (1979). "Statistical-Thermodynamic Approach to Determination of Structure Amplitude Phases". Sov.Phys. Crystallography. 24 (5): 519–524.

Li W. (2011) "A Parallel Multi-start Search Algorithm for Dynamic Traveling Salesman Problem". In: Pardalos P.M., Rebennack S. (eds) Experimental Algorithms. SEA 2011. Lecture Notes in Computer Science, vol 6630. Springer, Berlin, Heidelberg

Pillac V., Gendreau M., Guéret C., Medaglia A. (2013). "A review of dynamic vehicle routing problems", European Journal of Operational Research, Volume 225, Issue 1, 2013.

Knopp S., Sanders, P., Schultes D., Schulz F., and Wagner D. (2007). "Computing Many-to-Many Shortest Paths Using Highway Hierarchies" Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX). 2007, 36-45

Wirtz J., Schofer J., Schulz D., (2004). "Managing Large Scale Transportation Disruptions: Using Simulation to Test Traffic Incident Management Strategies: Illustrating the Benefits of PrePlanning", Northwestern University: Evanston