Fast Matrix Queries and Application to Routing Optimization Problems

for 29th ITS WORLD CONGRESS
Jean-Sébastien Gonsette
Expert Software Engineer, AISIN-Europe
Jean-sebastien.gonsette@aisin-europe.com
Noémie Meunier
Senior Software Engineer, AISIN-Europe
Noemie-meunier@aisin-europe.com

ABSTRACT

Complex optimization problems involving the computation of efficient routes to move vehicles between different locations arise in different fields, like in logistics. In contrast to the computation of a single shortest path, they require to minimize globally a set of routes shared by different vehicles to accomplish a certain goal. Solving such problems often starts with the computation of cost matrices related to all the possible pairs of locations. Those matrices can quickly get big and efficient algorithms to deal with them are a necessity. In this paper, we present the implementation of a routing engine that is based on state-of-the-art algorithms. After discussing its performance, we highlight several use cases where we use it to solve real application problems.

INTRODUCTION

Dijkstra's algorithm for finding the shortest paths between nodes in a weighted graph, like the one that represents a road network, dates to 1956 and is known to be optimal. Routing applications based on this algorithm were already introduced in car navigation systems in the early nineteens. Nowadays, routing applications have spread everywhere and are a real commodity. But despite this long history of successes, the quest for efficient routing algorithms remains a hot topic in computer science and transportation studies.

This situation is due to several factors that keep pushing the computational complexity of routing problems. The first explanation comes from the size of real-world graphs that continues to grow, including many more details than in the past. And this increase in size does not come only from the graphs themselves, but also from the data attached to their nodes and edges and that allows building more complex metrics to solve. If the first Personal Navigation Assistant (PNA) could only minimize a simple travelling time to some destination, modern applications can now consider complex information like real-time traffic.

The second point is about the trade-off between speed and optimality that the routing algorithms must cope with. Indeed, even if Dijkstra's algorithm is optimal, it remains far too slow on continental-size graphs or when sub-millisecond decisions must be taken (as it is the case for routing services in the cloud). Therefore, reducing the computation time without relying too much on heuristics or approximations that will deteriorate the output quality remains crucial.

Finally, as we will discuss it later in this paper, there are situations related to complex optimization problems where the computation of a route from A to B is not enough. Such

scenarios typically arise in logistics with the so-called Vehicle Routing Problem (VRP), or when the router is only part of a more general problem. This is for example the case for a planner that must decide where and when to recharge the battery of an electric vehicle, or for multimodal routing. The common point of all those challenges is the need to first compute a full matrix of source-target pairs, which will then be used by the solver at stake. Such route calculation is considerably more involved as even a small set of one hundred locations already requires the processing of ten thousand costs.

CONTRIBUTION

AISIN, through its division Connected & Sharing Solutions, is a global leader in mobility solutions. Since the early age of navigation, AISIN has a successful track of records of innovative technologies. Today, one of our directions is to continue to help the drivers or carriers to reduce their costs, energy consumption and CO_2 emissions. This vision not only requires pushing the capabilities of our routing engine, but also to make it compatible with new applications we encounter beyond the simple navigation system.

This paper focuses on different techniques we have implemented in our router to process efficiently (in terms of speed and optimality) queries involving big matrices of location pairs. We describe an implementation we have developed based on state-of-the-art algorithms found in the literature, and that can handle both single-path and many-to-many request. This one also includes an original technique to conduct searches compatible with time-dependent requests and that shows up better performance for local queries.

In a second part, we go through different real routing applications that AISIN is working on, and that are built upon this router. We explain how it can support optimization use cases related to the Vehicle Routing problem in logistics and EV route planners.

BACKGROUND

For many reasons, the shortest path problem is not a completely solved problem and the last decade has seen the emergence of many techniques to considerably accelerate its computation time or to deal with more complex constraints. Although it is a bit dated, an extensive review of such techniques is available in (1) and would give to the interested reader a good overview of all the available flavours.

This work is exclusively built upon the so-called (Customizable) Contraction Hierarchies (CCH) algorithm. We chose it because, even if it is not the fastest one, it is nevertheless very efficient without compromising too much the flexibility of its use. This technique appeared first in the literature in 2008 in (2) and since, has been consistently improved year after year. It would be difficult to screen all the literature related to the CH but we can still mention some of them. In 2010, (3) proposed a first attempt to deal with *time-dependent* requests. In 2015, (6) introduced the concept of *customization* in the CH, resulting in the possibility to change the weights of the contracted network without having to redo the whole (and slow) contraction process. In 2019, (7) explained how to improve the contraction quality with *nested dissections*. In 2021, (8) presented a way to significantly reduce the size of the contracted graph when dealing with time-dependent constraints for traffic predictions.

Many-to-Many cost queries are less predominant in the literature and are often only slightly approached in papers addressing single-path routing queries in the first place. We can still mention (9) which focuses on this problem in the context of the Customizable Route Planning algorithm.

FAST MATRIX QUERIES

OVERVIEW OF CONTRACTION HIERARCHIES

Acceleration techniques based on the Contraction Hierarchies (CH) is a big subject that is out of the scope of this paper. However, a minimum understanding of its underlying concepts is mandatory to present the solution we have developed.

Even if solving the shortest path problem with contracted graphs has been presented in (2) in 2008, the mathematical theory it relies on dates from the late fifties, with different advances related to the fields of semidefinite optimization and sparse matrix decomposition. To put it simply, it was shown that any difficult problem that has a graph structure can be solved efficiently provided that this graph is *chordal*; see (4) and (5).

A *chordal* graph is one in which all cycle of four or more vertices has a chord. This is obviously not the case of a road network, but it is always possible to transform a graph in such a way this property holds. This operation (called the contraction) does not affect the number of nodes but adds a significant number of supplementary fictional edges along the way. Those new edges are called *shortcuts* and they can join potentially distant elements of the graph.

In a chordal graph, nodes are numbered and hierarchically *organized in levels*, with the most important ones at the top. They correspond to nodes that have higher odds to be part of any solution path. The different levels exhibit a *tree structure* (called the *elimination tree*) where the number of nodes is reducing from the leaves at the bottom to the root. In the case of a perfectly balanced tree structure, the decreasing rate is exponential.

The chordal graph's elimination tree structure resulting from the contraction guarantees an *important* property: Any shortest path *st* between a pair of source node *s* and target node *t* can always be split into an upward path P_u and a downward path P_d in the contracted graph. The sequence of node orders in the path P_u must occur by increasing rank, while those in P_d must unfold by decreasing rank.

This property can be understood by looking at the shortest path between nodes 3 to 19 on the illustrative toy graph of Figure 1. Thanks to the available shortcuts, it can be split into the upward path *3-4-10-21* and the downward path *21-20-19*. This sequence of increasing, decreasing node numbers would not have been possible without the shortcut arcs *3–4* and *20-19*.

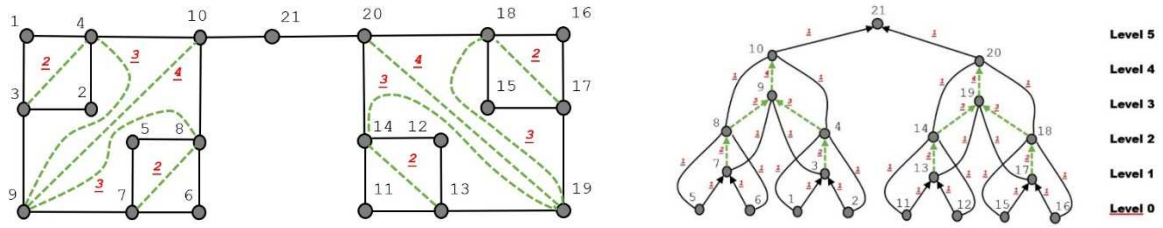

***Figure 1 Left**: Toy example to illustrate the advantages of CH graphs. Shortcuts arcs resulting from the contraction are in dashed green, with their associated weights in red. Legacy arcs are assumed to have a cost of 1 for simplicity in this example.*

***Right**: Same toy graph arranged by levels and showing the elimination tree structure. Branches of this tree are the arcs with an arrow. There is one such arrow arc by node, corresponding to the edge reaching the lowest node in the upper neighbourhood. Those arrow arcs form a skeleton that gives a mathematical structure to the original graph. Without the green shortcuts, this structure would be lost.*

This decomposition is the key to the efficiency of the CH shortest path algorithm. Indeed, performing a Dijkstra search in the *upward graph* of the source node *s* enables to only look at nodes of *increasing order*, which is equivalent to climb in the elimination tree. But as the number of nodes is decreasing exponentially fast in this direction, it significantly reduces the number of nodes to explore. The same efficient approach can be taken at the target side if the weights are not time dependent. Starting from the target node *t*, the search algorithm can follow the *downward* graph until a common node is found between the two sides.

IMPLEMENTATION

We have implemented our CCH router, along with our many-to-many query algorithm, in C++ for performance reasons. Nodes of the graph are ordered according to the *IntertialFlowCutter* algorithm described in (7). It consists in iteratively splitting the routing graph into two balanced subgraphs whose node separator has a minimal size. Each subgraph is then further halved until only obtaining trivial graphs whose nodes can be ordered directly. All the nodes can then be numbered, and the graph contracted accordingly.

The contracted graph is stored in an adjacency matrix that can easily be indexed to find the neighbourhood of any node. Such graph fits easily in memory, even for a big map like the one of full Europe containing ~80M nodes. In this very case, around 8 GB of RAM is required.

The graph customization is done as explained in (6) and consists in weighing all the shortcut arcs introduced during the contraction in accordance with the legacy arcs. This operation is done in two steps, with a *basic customization* followed by a *perfect customization*.

A basic single-path query consists in running two instances of the Dijkstra's algorithm. The first one expands from the *source* node and only follows arcs in the *upward* graph, while the second runs from the *target* node in the *downward* graph. We alternate between the two sides in order that the radius stay in sync. The two sets of nodes settled at both sides eventually meet at some junction nodes, forming shortest path candidates. Search is stopped once the radius at one side is larger than the best solution found so far.

The downside of such implementation is that it assumes searching from the target side is possible. This is only true for non-time-dependent queries. In the other case, the arrival time is not known, and the arc weights are undetermined. This point is discussed later in this paper.

MANY-TO-MANY MATRIX QUERIES

Compared to vanilla Dijkstra's shortest path algorithm, CH-based shortest path searches can expect a significant reduction of computation time. But even if the improvement is considerable, it is still possible to go further in the context of *many-to-many* shortest path requests. Such request is typically expressed as two vectors of source nodes S and target nodes T, and it must return a $|S| \times |T|$ cost matrix C where c_{ij} expresses the cost of the shortest path between s_i and t_j. A naïve and costly implementation is easily achieved by just iterating on the product space to compute all those costs with single-path queries.

The optimization technique consists in avoiding exploring several times the same search space by storing the information collected while doing the backward search from the different target nodes t_j. Therefore, the query can iterate on each source node s_i while leveraging on the previous Dijkstra backward search from the $|T|$ target nodes. This technique works well for a large family of routing techniques but is especially suitable with the CH, thanks to the narrowness of its search space (9). This allows processing huge matrices without consuming too much memory.

FORWARD SEARCH IN CONTRACTION HIERARCHIES

If the weights of the graph are time dependent (as in most of the real-life applications), it is no longer possible to conduct a backward search from the target node. A solution to this problematic is described in (8). It consists in avoiding the usage of the Dijkstra priority queue at the target side by exploring the whole elimination tree from the source and target to the root. This builds a corridor in which a forward-only Dijkstra search is executed in a second step.

The downside of this method is that the elimination tree technique requires to climb the whole graph up to the root. Therefore, the computation time of the request becomes sensitive to the height of the elimination tree, and thus to the size of the map. We present here a slightly different approach we have implemented in our router and that avoids performing a complete traversal. It allows reducing the search time for queries where the source and target are not too far from each other, as it may be the case for scenarios encountered in urban logistics.

The first phase of this forward search algorithm consists once again in conducting two expansions, from the source and target sides. The search from the source side remains the same: a standard Dijkstra's algorithm expanding in the *upward* graph thanks to a priority queue. All the weights can be determined exactly as the arrival time is known at each node.

The search from the target side is different and consists in traversing the elimination tree without a priority queue. Starting from the target node *t*, all the parent nodes are visited in sequence with respect to their node order. All the encountered edges *uv* form a *corridor* that is used later in the second phase. Edges weights are not known exactly but we assume we can get lower and upper bounds, namely $\underline{b}_{uv}$ and $\bar{b}_{uv}$. Those bounds correspond to the lowest and highest costs along the edge profiles. They are used to relax similar bounds for each encountered node *v*, that is checking if $\bar{t}_u + \bar{b}_{uv} < \bar{t}_v$ or $\underline{t}_u + \underline{b}_{uv} < \underline{t}_v$ and improving the bounds of *v* if possible.

We still maintain a kind-of priority queue during the elimination tree traversal that records the lowest bound of each settled node that is not visited yet. The first item in the queue is constantly the one with the smallest lower bound among the remaining nodes, so that we always know the optimistic cost of a potential solution.

We once again alternate the process between the two sides so that the search radius stays in sync. The two sets of nodes settled at both sides eventually meet at several junction nodes, forming *potential* shortest path candidates. Indeed, the paths formed at those junctions are only understood in the extent of their bounds at the target side. Therefore, the cost c_j of each path going through a junction node *j* is bounded by $\bar{c}_j = s_j + \bar{t}_j$ and $\underline{c}_j = s_j + \underline{t}_j$.

We maintain a best worst cost *bwc* along the junction candidate collection process of this first phase: $bwc = \min_j \bar{c}_j$. It allows us to early filter out junction candidates whose best cost is worse than this value, that is if $\underline{c}_j > bwc$. In addition, it permits to stop searching early without having to exhaust the whole elimination tree at the target side. The search is stopped once the radius at the source side and the optimistic cost at the target side are both above bwc.

This first phase ends up with a collection of suitable junction nodes *j*, all compatible with *bwc*. This set initiates the starting point of the second phase, consisting in performing a Dijkstra search down the *corridor* conducting to the target node. All the nodes encountered along the way are filtered out if they are not part of the corridor, or if their total cost is above *bwc*.

Finally, this *forward-search* technique can be combined with the *many-to-many* optimizations described above and we discuss the results we can get from it in the next paragraph.

EXPERIMENTS

Table 1 provides an overview of the many-to-many query performance, *single thread*, for different query sizes $n=|S|=|T|$. Experiments were conducted on an AMD Ryzen 9 5900X 12-Core Processor@3.70 GHz. *All the costs are static*, which means the running time comparison does not consider the additional processing that would be needed to unpack actual time-dependent metric values. As such, those experiments only compare the impact of the exploration algorithm itself.

The underlying map is the one of *Texas*, extracted from Open Street Map. This one is made of 4.5M vertices and 5.7M edges, rising to 17.2M edges after the contraction process. All the graph data required to run the router consume around 700 Mb of RAM. The contraction phase took 4 min, parallelized on 11 cores, while the customization phase took 15 sec single thread.

In contrast to what we often see in the literature, we do not choose the query nodes uniformly at random across the whole map. Instead, we opt for discriminating different query radii that better reflects real local use cases.

A query of radius r (in terms of nodes) is built by choosing a centre node randomly, then by generating n random path starting at this position. A node among the r first nodes of each path is then selected evenly at random. This procedure ensures all the source and target nodes inside a query remain local.

Each query in the table is run x times, such that $x.|S| > 10000$. This means that a query with $|S|=|T|=100$ was run 100 times and its corresponding measurements averaged. This allows for a good balance between sufficiently large test sizes and acceptable running times.

		Static (Dual-Dijkstra)	**Static (Forward-Search)**	
Query radius	**|S|=|T|**	Time (s)	Time (s) - FT	Time (s) - ES
#100	100	0.042	0.453	0.205
(~17min)	200	0.100	1.487	0.710
	500	0.361	8.433	3.838
	1000	1.493	42.56	18.02
#200	100	0.071	0.444	0.288
(~39 min)	200	0.197	1.681	1.082
	500	0.887	10.76	7.410
	1000	3.077	42.08	32.63
#500	100	0.139	0.497	0.438
(~1h45min)	200	0.308	1.589	1.404
	500	1.233	10.593	9.497
	1000	4.944	40.01	38.42
#1000	100	0.166	0.509	0.494
(~4h)	200	0.962	1.775	1.702
	500	1.548	11.79	11.64
	1000	7.684	51.24	54.72

***Table 1**: Performance of many-to-many queries. Query time and memory usage are presented for both Dual-Dijkstra and Forward-Search algorithms (**FT**: Full Traversal, **ES**: Early Stop) in the case of static costs. All queries calculate a full cost matrix from a source and target set with size $|S|=|T|$, for different query radii (in terms of nodes). We used a time metric in this experiment, and the equivalent driving time of each query radius is also given for information.*

As expected, the forward-search queries are always slower than their dual-Dijkstra counterparts. This is not a surprise since searching only up in the elimination tree is what makes contraction hierarchies particularly fast. However, it is interesting to look at the Full Traversal and Early Stop variations, since they put in light how the size of a map can impact the query times.

We can see that, in the case of the FT variation, the query radius size only slightly affects the computation time. The reason is that it must traverse the full elimination tree in any case, while the height of the tree directly reflects the map, not the query. As a result, running a local query with a small radius costs almost the same as a countrywide query.

This behaviour is different with the Early Stop variation where we can see that, for a given $|S|=|T|$, the computation time increases smoothly with the query radius. Small radius (i.e. 100 nodes), like we can encounter in urban logistics, run twice faster. This advantage then decreases with bigger radius and vanishes completely when we hit the size of the map.

APPLICATIONS

Efficient calculation of cost matrices is key in many applications. For example, they arise in the logistics field for the scheduling of a fleet of delivery vehicles, with the so-called Vehicle Routing Problem (VRP). They can also be used in EV route planner applications, for the trip planning of Electric Vehicles (EV). In this section we have a look at these two applications we are working on, and that rely on our many-to-many request algorithm.

APPLICATION IN LOGISTICS

The aim of the VRP applied to logistics is to determine optimal routes from a depot to a set of delivery locations (customers) with various constraints, such as route length, delivery time windows, precedence relations (pick-up and delivery), etc. Each location is visited exactly once by a vehicle with a potentially limited capacity. VRP is a NP-hard combinatorial optimization problem. Many exact methods and (meta)heuristics have been studied to compute good solutions to this problem and its numerous variants for several years (10)(11).

VRP is not only hard to solve, but preparing its input is also a complex computational task. Indeed, the first step asks to evaluate the driving cost (e.g. distance, time, energy consumption) between all pairs of locations. Many datasets exist to benchmark different methods solving the VRP and its variants (12). However, most of these datasets assume that the customer locations are separated by a straight line. In this case, the distance as the crow flies is easily calculated whatever the number of requests. This is unfortunately not very representative of real-life applications and even worse, it ignores traffic and more complex constraints related to electric vehicles. Therefore, the usage of an efficient many-to-many algorithm to compute shortest paths (according to some metric) becomes an important matter.

As stated previously, the size of a matrix query grows in n^2 with n the number of locations to consider. Besides, for real-life cases we are working on, this rate can even be worse; either because of the need to determine several times the same cost matrix with different parameters, or because we need to integrate extra locations to solve the problem.

The basic VRP supposes that travel times are constant during the day. However, when considering customer time frame constraints (called the time windows constraint), traffic congestion consideration can be crucial to perform the delivery when the customer is available. Time Dependent Vehicle Routing Problem (TDVRP), a VRP including traffic information, has been introduced in 1992 in (13) and (14) to make more realistic

optimizations. In this case, time varies in function of the period of the day and the cost between the same location pairs can evolve drastically. It is then required to compute different distance matrices for different time intervals in the day. For traffic information varying every 15 min, 15 × 8 distance matrices should be computed for a driving working period of 8 hours.

Let us now present a concrete application of the VRP we are working on and using our many-to-many request algorithm. Industrial parks are often lacking any public transport infrastructure. Hence, some companies address this issue by renting vehicles to transport their employees between the working facilities and the bus stations they need to use to commute to home. This is a good option for employees but, because of the dispersion of the different bus stations, the implemented solution may lead to longer driving distances or greater uses of vehicles than necessary.

This problem is highlighted in the left map of Figure 2 where each company is depicted by a coloured dot, along with the routes followed by their transport vehicles. We can see a lot of redundancy in the south of the map, because of different companies dropping their commuters in the same cluster of locations. Also, in the problem at stake, most of those vehicles drive under capacity.

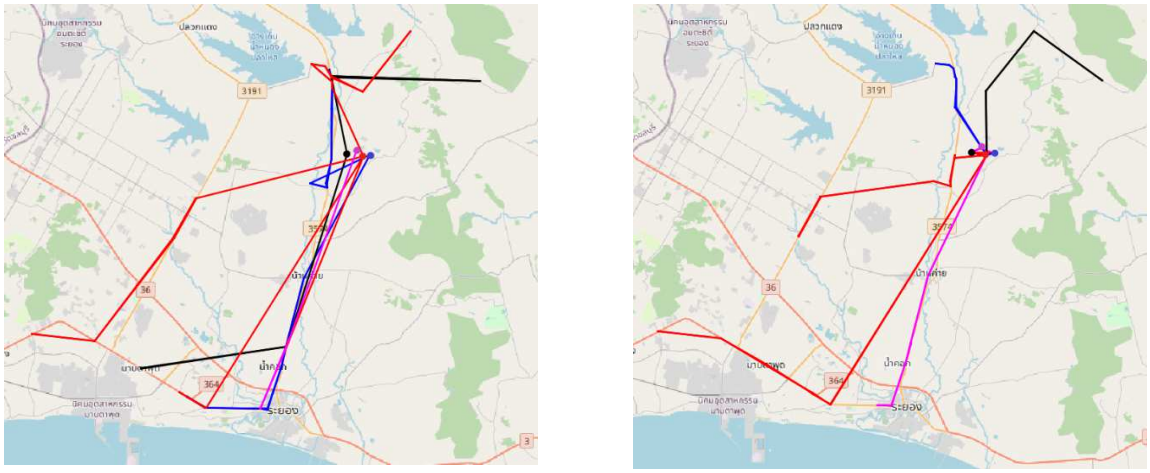

***Figure 2 Left**: Original routes followed by the 9 different vehicles bringing the employees of different companies (4 coloured dots) to the different bus stations in the area.*

***Right**:New solutions computed by our solver in the context of a transport-sharing system. It only requires 5 vehicles to achieve the same result.*

This situation could be avoided by using a vehicle sharing system in which a company does not transport only its own commuters but also pick-up employees from other companies. This is a typical VRP problem with pick-up and delivery constraints that can be solved by using the results delivered by a many-to-many request algorithm. This later enables to compute massively all the shortest paths between all the location pairs of interest, and this information is used in a second step by an operational research engine. The vehicle usage can then be considerably reduced, as depicted on the right part of the picture. In this example, the new solution needs only five vehicles instead of nine in the original approach, hence reducing considerably the costs for each industry as well as CO_2 emissions by decreasing the total travelled distance.

Solving this problem quickly and efficiently is, of course, important if a day-by-day solution is required to handle different dynamic constraints. The solution can indeed be impacted by traffic conditions, but also by the schedule of the different companies shifts.

APPLICATION IN EV ROUTE PLANER

A performant many-to-many request algorithm is not only useful to prepare inputs for the VRP, but also to tackle the problem of determining a single long trip for an electric vehicle. Indeed, such path can potentially diverge from the simple shortest path joining the starting and ending locations. The vehicle's battery capacity is limited and potential stops at charging stations are unavoidable.

AISIN is developing an EV route planner that can compute optimal routes for an EV driver given different parameters like the remaining battery level, the weather and the temperature or the traffic. The optimality can be expressed as the function of the total trip time (including recharging time), the energy consumption, the cost or any convenient metric mixing those variables. The goal of the planner is then to recommend at which charging station to stop and what charging level to reach to continue the trip.

Solving this problem requires to work with a graph of stations, where nodes stand for charging locations and edges represent feasible trips in between. Different information can be attached to those edges so that the joining cost can be adapted with the variation of some dynamic factors like the weather and the temperature. However, the driving range of an EV can reach hundreds of kms. This means that this graph must be precomputed efficiently as each charging station is connected with the many other ones within this range.

Besides this precomputed graph, each planning request requires to determine the cost to reach every station in range from the starting and target locations. This is only with this combined information that the best trip can be selected to reach the destination while minimizing the metric of interest. Considering the high number of charging stations, efficient many-to-many requests is thus a key building block of such solution implementation.

An example of planning computed by our planner is depicted in Figure 3. The optimal route without recharging takes around four hours and is unreachable straight given the limited autonomy of the EV at stake. The planned route takes five hours in total, including half an hour of recharging time.

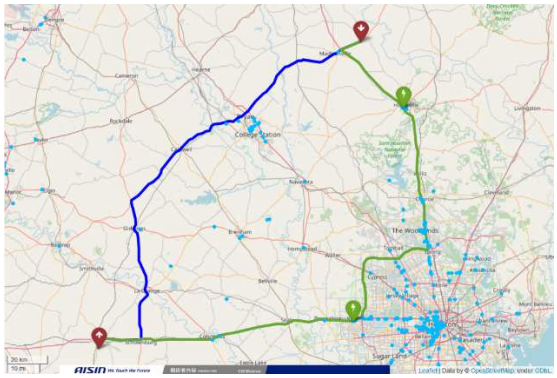

Figure 3: *Example of route planning where the selected solution (in green) can only be found by considering charging stations at potentially far distances from the initial path (in blue). This initial path is the fastest one if recharging is not needed. The different charging stations are scored individually by considering the extra driving time they require, but also many other factors like the estimated waiting time and charging time.*

CONCLUSION

The problem of routing has evolved considerably during the last decade in order to support more advanced scenario. As such, single path queries are not the norm for problems where complex operating points must be found. In this context, AISIN enhanced its routing algorithm to strive for better performance and flexibility.

Leveraging on the contraction hierarchies' capabilities, we were able to successfully revise our routing algorithm to support many-to-many requests. In addition, this algorithm has been

thought to be favourable to the case of a local time dependent queries, as it can be found in urban logistics. This new development opens the door to more present-day use cases like the one encountered in logistics or in route planning.

REFERENCES

(1) Madkour, A., Aref, W. G., Rehman, F. U., Rahman, M. A., & Basalamah, S. (2017). A survey of shortest-path algorithms. *arXiv preprint arXiv:1705.02044*.

(2) Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms: 7th International Workshop, 2008 Proceedings 7* (pp. 319–333). Springer Berlin Heidelberg.

(3) Batz, G. V., Geisberger, R., Neubauer, S., & Sanders, P. (2010). Time-dependent contraction hierarchies and approximation. In *Experimental Algorithms: 9th International Symposium, SEA 2010, Proceedings 9* (pp. 166–177). Springer Berlin Heidelberg.

(4) Blair, J. R., & Peyton, B. (1993). An introduction to chordal graphs and clique trees. In *Graph theory and sparse matrix computation* (pp. 1–29). Springer New York.

(5) Vandenberghe, L., & Andersen, M. S. (2015). Chordal graphs and semidefinite optimization. *Foundations and Trends® in Optimization*, *1*(4), 241–433.

(6) Dibbelt, J., Strasser, B., & Wagner, D. (2016). Customizable contraction hierarchies. *Journal of Experimental Algorithmics (JEA)*, *21*, 1–49.

(7) Gottesbüren, L., Hamann, M., Uhl, T. N., & Wagner, D. (2019). Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, *12*(9), 196.

(8) Strasser, B., Wagner, D., & Zeitz, T. (2021). Space-efficient, fast and exact routing in time-dependent road networks. *Algorithms*, *14*(3), 90.

(9) Agterberg, P. (2017). *Many-to-many customizable route planning with time-dependent driving restrictions* (Master's thesis).

(10) Elshaer, Raafat, and Hadeer Awad. "A taxonomic review of metaheuristic algorithms for solving the vehicle routing problem and its variants." *Computers & Industrial Engineering* 140 (2020): 106242.

(11) Baldacci, Roberto, Aristide Mingozzi, and Roberto Roberti. 'Recent exact algorithms for solving the vehicle routing problem under capacity and time window constraints.' *European Journal of Operational Research* 218.1 (2012): 1–6.

(12) **https://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article=7040&context=sis_research**

(13) Malandraki, C., & Daskin, M. S. (1992). Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms. *Transportation science*, *26*(3), 185–200.

(14) Hill, A. V., & Benton, W. C. (1992). Modelling intra-city time-dependent travel speeds for vehicle scheduling problems. *Journal of the Operational Research Society*, *43*(4), 343–351.