DORC: Distributed Online Route Computation – Higher Throughput, more Privacy

Niklas Schnelle, Stefan Funke
Universität Stuttgart, Germany

Sabine Storandt
Universität Freiburg, Germany

***Abstract*—We propose a technique to distribute the workload of online route planners as offered for example by Bing/Google/Yahoo Maps, etc. among the clients requesting the routes. Our scheme not only increases the throughput of a server answering the requests of clients but also yields a simple way of providing some degree of privacy for the user. A prototype implementation of our system is available as an Android app in Google Play and on Github.**

***Keywords*-route planning, distributed computing**

I. Introduction

The use of online route planners like Bing/Google/Yahoo Maps, etc. has become ubiquitous in our daily lives. When booking a hotel or planning a sightseeing trip to a city, most websites provide direct links to services like Google maps. And indeed, most of the time we actually check for good reachability of the venue before deciding in favor or against a particular hotel or restaurant. On a road trip, the use of paper-based map material has almost become extinct. Most of the time we just type in the destination address in our smartphone to guide us there. When we are unsure about the traffic conditions, we use Google maps or the likes to suggest the best route under the current circumstances.
In this work we consider two of the challenges that arise in the context of online route planners, namely distribution of the computational load and privacy concerns of the users.

Load Distribution: It is clear that considerable compute power is necessary to cope with the load created by the route planning requests issued at any given moment in time e.g. to Google maps. This is a problematic issue for smaller companies or open source initiatives aiming to enter the online route planning arena and even for big companies more servers mean more energy consumption, which for companies like Google is one of the biggest cost factors; an online route planner might have the coolest features and the best user interface, if the servers cannot cope with the requests, it is doomed to fail on the market. And not all companies or initiatives have the resources to set up large server farms. The first goal of this paper is to show how recent techniques to speed-up shortest path queries can be instrumented to *efficiently distribute* the computational load for calculating routes among the clients which issue the actual requests. This allows even projects with limited resources to have an impact on the market.

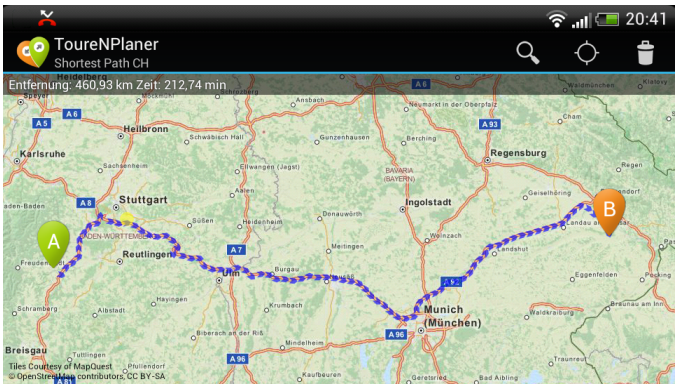

Figure 1. Screenshot of our Android App.

Privacy: Most of the route queries issued from mobile devices like smartphones or handheld tablets are of the kind that the shortest/fastest/best route from the *current location* to some destination is requested. Obviously, by issuing such a request to a maps service, the respective service provider can collect information regarding the whereabouts of the issuer of the request as well as his intended destination. Some people might object to this kind of data collection and the natural solution for that is the use of *offline route planners* which have all the necessary road network data on the device and can perform the route planning without contacting any external service. *Online route planning* has some indisputable advantages, though: it can make use of up-to-date road network information, in particular temporary disturbances like traffic jams or construction work can be taken into account; also, more involved route planning tasks might require more memory/compute power than offered by typical mobile devices. The second goal of this paper is to exhibit means which allow for online route planning but still keep some degree of privacy. More concretely, we try not to let the provider of the route planning service know exactly which trip we are planning. It turns out that the same idea which helps with distributing the load will also allow for the provision of some privacy towards the service provider.

Our Contribution

In this paper we devise a scheme which allows for offloading the computational load of online route planning from the server to the clients. This is achieved by transferring to the client a small *synopsis* of the graph data which

captures all the information necessary to perform the route planning locally at the client. On the server side, the effort to compose and transfer these synopses is considerably lower than performing the actual route planning itself, hence increasing the throughput of the server considerably. As the graph synopses are very small, a client which wants a single route planning task to be solved can easily request several small synopses, keeping the server in the dark which source-destination pair the client is actually computing a route for. Our scheme is derived from a very recent speed-up technique for shortest path queries called *contraction hierarchies*, [1].

II. Preliminaries

A. The Classical Shortest Path Problem

Computing the shortest path in a directed, weighted graph is one of *the* classical graph optimization problems. For non-negative edge weights, Dijkstra's algorithm [2] has been the non plus ultra in terms of asymptotic running times since decades. In a graph with n nodes and m edges, Dijkstra's algorithm takes $O(m+n\log n)$ time to compute the shortest path from some node s to some other node t. In practice, a decent implementation on a modern Desktop PC takes on the order of a few seconds on a road network like Germany ($n = 16 \cdot 10^6$, $m = 30 \cdot 10^6$) which clearly prohibits its application in an online route planner scenario.

B. Speed-up Schemes

Classical speed-up techniques for Dijkstra's algorithm like A* or bidirectional search achieve some acceleration by pruning the search space e.g. via goal direction. The effect on the query time for road networks is limited, though. Typical query times only improve by a small constant factor. In recent years, more advanced speed-up schemes have been developed which rely on a *preprocessing phase* which precomputes some data allowing for faster query times later on. Edge reach [3], or transit nodes [4] are schemes that reduce query times to as little as *microseconds* - an improvement by a factor of around 1 million *without compromising optimality of the result*.

1) Contraction Hierarchies (CH): This speed-up technique was introduced in [1]. The basic idea of CH is to assign *levels* to the nodes and augment the graph with additional *shortcut edges*, such that every shortest path $s \rightsquigarrow t$ has a representation in the augmented graph where the levels of the nodes along the path first increase monotonically until reaching a node of maximum level and then decreases monotonically towards the target t. This special structure of shortest paths allows for the pruning of most edges at query time and still obtain the optimal path. Better query times are obtained if the level of a node is correlated to its importance in the road network, see [1] for more details.

Our concrete implementation of CH works in phases: Starting with phase $i = 0$ we pick a maximal independent set I_i of the nodes all of which get assigned level i and are removed/contracted from the graph. For the final outcome to preserve all shortest path distances, *shortcuts* have to be inserted. More precisely, when contracting a node $v \in I_i$, for every path uvw a shortcut edge $e = (u, w)$ has to be added iff uvw is the only shortest path from u to w (this can be checked via a Dijkstra from u to w). The cost of e equals the added costs of the edges (u, v) and (v, w). Phase i ends when all nodes in I_i have been contracted and phase $i+1$ starts. The preprocessing step is finished as soon as all nodes have been contracted.

The outcome of the preprocessing step is a new graph G' consisting of all nodes (now with levels!) and edges of the original graph and all shortcuts created during the process. An edge $e = (v, w)$ (original or shortcut) is called upwards if the level of v is smaller than that of w and downwards otherwise. The construction scheme guarantees that every shortest path has a representation in G' which is sequence of upward edges followed by a sequence of downward edges. Therefore s-t-queries can now be answered via a variant of the bidirectional Dijkstra algorithm where the forward search (starting at s) considers only outgoing upward edges, while the backward search (starting at t) is restricted to incoming downward edges. This results in query times which are about a factor of 1000 better than ordinary Dijkstra.

III. DORC - Distributed Online Route Computation

How to distribute the computation load from the server amongst the clients? For a query from some source node s to a target t – instead of running a bidirectional Dijkstra on the server – we perform the following steps on the server:

1) use breadth- or depth-first search starting from s to identify all nodes reachable via upward edges from s (we call this the upward graph G_s^{up}).
2) identify all nodes that can reach t via downward edges (G_t^{down}).
3) send G_s^{up} and G_t^{down} to the client.

On the other end, the client receives the G_s^{up} and G_t^{down} and computes the shortest path distance in $G_s^{up} \cup G_t^{down}$. The practicability of this approach depends critically on three aspects: a) how much cheaper is the identification of G_s^{up} and G_t^{down} on the server compared to an actual shortest path computation? b) how large is the data package with G_s^{up}/G_t^{down} to send over the network? c) is the computational effort required at the client end still acceptable?

Unfortunately, it turns out that identifying and transmitting the complete up- and down-graphs G_s^{up}/G_t^{down} only provides a modest improvement in terms of the achievable server throughput. Looking closer at the up-graphs for different nodes, though, one realizes that the high-level parts of those graphs exhibit considerable overlap. So one reasonable strategy is to transfer the whole subgraph induced by all nodes of level at least l (we call this subgraph the *CORE*) to the client *once*. Later queries then save both

the exploration as well as the transmission of these high-level portions of the CH-augmented graph. Table I lists the sizes of the CORE graphs above a certain level. For example, when choosing the CORE graph to be above level 80, the respective induced subgraph has 476 nodes and $18,698$ edges, and a space consumption of 0.49 MB (uncompressed). This CORE graph has to be transmitted once to the client. Then every subsequent query requires the collection and transmission of the up- and down-graphs below level 80 only. In the section IV we will see how the choice of the CORE graph size affects the achievable throughput on the server side.

Level $\geq l$	40	80	100	120
#Nodes /#Edges	1092/39019	476/18698	226/10443	103/4542
Space (MB)	2.2	0.492	0.277	0.121

Table I
NUMBER OF NODES AND EDGES OF THE CORE GRAPH (NODES AND EDGES ABOVE LEVEL l) FOR OUR SAMPLE ROAD NETWORK OF GERMANY (16271859 NODES, 62062727 EDGES).

On the client side, G_s^{up}/G_t^{down} (or their respective lower parts below level l) arrive as a sequence of triples (v, w, c) representing an edge from v to w at cost c each, which has to be turned into a proper graph representation before the actual route computation can take place.

A. Prototype Implementation on Google Play

To show that the DORC scheme actually pays off, we have incorporated DORC into our already existing ToureNPlaner Android client that was developed as student project at the Universität Stuttgart. We were able to implement our scheme with only modest changes to the apps core architecture and kept all of it's previous functionality intact. The enhanced version of our ToureNPlaner app is available on Google Play. The most important additional software component on the client is a light weight graph data structure. Unlike typical high performance graph representations as used in our server (e.g. offset array based), this one has to cope with node/edge id's that are distributed over a large range, which makes it impossible to use them as index into an array. This problem is amplified by the fact that we need to keep the exact id values, at least for the nodes, to be able to link them back to the corresponding items on the server.
Therefore we used a hash based data structure so we can address nodes by the same id as on the server, this also enables us to transfer subsets of the server's graph as simple lists of the aforementioned triples. Building on this principle we developed a graph representations that can be initialized with the CORE. When running a request the client then receives the additional graph data from the server and augments it's own graph with it. After a request the client can easily discard parts or all of the augmentation.

IV. EXPERIMENTAL RESULTS

A. Road Network Data

Our experiments are based on the road network of Germany extracted from publicly available OSM data. The respective graph has $16,271,859$ and $62,062727$ edges (when augmented with CH shortcuts), as edge costs we used travel times based on the respective road categories.

B. Throughput Analysis

1) Testing Environment: All measurements have been performed on typical server hardware with relatively low single core performance. We used a dual socket motherboard with 16 GB RAM and two AMD Opteron 6128 with a total of 16 cores clocked at 2.0 Ghz. On the software side we used Arch Linux with Kernel version ≥ 3.6 and 64 bit Java 7 (OpenJDK 64-Bit Server VM build 23.2-b09).

2) Methodology: Unless otherwise noted all measurements have been performed with a special testing client over *loopback* networking on the same system. It uses the same HTTP based interface as our Android prototype to issue concurrent requests for randomly selected nodes of the graph and acquires timing and throughput information.

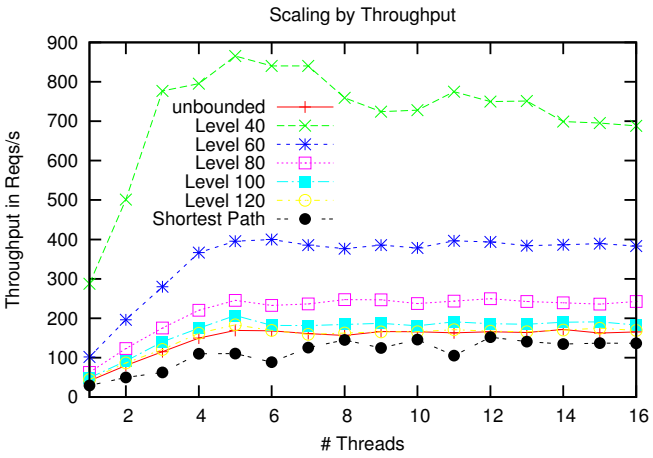

Figure 2. DORC throughput dependent on different CORE parameters.

3) Measurements and Analysis: Figure 2 shows the performance in requests per second of DORC while sending $G_s^{up} \cup G_t^{down}$ for each request and how it scales in the number of threads. To put the data in perspective we also show the current server side shortest path implementation as a baseline. We see that even without making use of a CORE graph, throughput is improved compared to pure server side computation. For few cores, we also observe linear speed up, using more than 4–5 cores does not pay off likely due to the memory bandwidth becoming the bottleneck. Employing a CORE graph yields considerably higher throughput, e.g. when using a CORE graph above level 40, a single server can handle close to 900 requests/second.

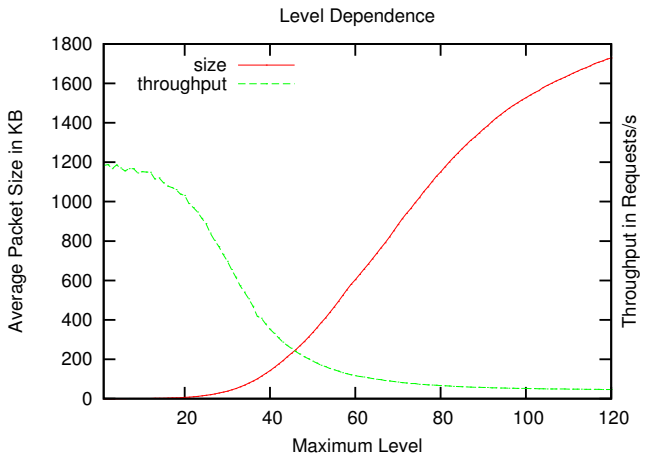

Figure 3. Average result sizes and throughput with different level parameters for the CORE graph; *one single* thread.

Figure 3 depicts (for *one single thread*) the correlation of the size of the transmitted graphs and the throughput versus the choice of the level parameter for the CORE graph. As expected graph sizes increase dramatically with growing level parameter whereas throughput improves drastically.

C. Client Side Performance

Finally we provide some performance data on the prototype Android client. As the capabilities of mobile devices varies broadly, we can focus only on one device which we consider 'standard' by todays standards. Our measurements were taken on a Galaxy Nexus with a 1.2 GHz ARM Cortex-A9 processor running Android 4.0. Because the performance on the client doesn't impact the overall throughput, number of serviceable clients or cost of the entire system it's performance matters mostly in terms of usability. If a mobile app can compute and display a route fast enough to make it feel right for the user there isn't really any point in putting much work into scraping of another millisecond.
The transfer of the CORE graph with level parameter 40 takes about 1.4 seconds (this has to be done only *once*!). Adding the upwards and downwards graphs for source and target takes another 115 milliseconds. The time for actual Dijkstra computation on the augmented graph depends on the distance of the path to be computed but was measured at about 500 milliseconds for a cross-country path in Germany.

D. DORC for Privacy Preservation

How could a client user hide its position and intended route of travel towards from server? A straightforward strategy is to not only issue the single request he/she is interested in, but let's say additional k *random* source-target requests. Then the server does not know which of the $k+1$ source-target pairs the user is actually interested in, e.g. for $k = 15$ there are 16 possible routes the user might take. The cost of achieving this '16-fold privacy' is the issuing and computation of 16 route queries on the server. Using our DORC scheme, we can do much better: additionally to the true source s and target t the client chooses 3 additional random sources and 3 additional random targets and queries the server for the 4 up-graphs (down-graphs) associated with the sources (targets). Upon delivery of the up-/down-graphs, the server cannot tell which one of the $4 \times 4 = 16$ source-target-pairs the client is actually computing and following. Hence using DORC the overhead compared to a single source-target query to achieve 16-fold privacy, is only a factor of 4. In general, by requesting κ-times more data from the server, DORC can achieve a κ^2-fold privacy, which for the naive obfuscation strategy would require $k = \kappa^2$-times more queries to be solved by the server.

V. Outlook

We have presented the DORC scheme which extends our existing online route planner by a mechanism to distribute the computation load amongst its requesting clients. DORC not only leads to a considerable increase of throughput on the server but also allows for more efficient ways of obfuscating the clients' routes of travel. While for ordinary shortest/quickest paths the computational load on the client was not a real issue, more complex queries like constrained shortest paths, or travelling salesperson tours (as already present in the purely-online version of our route planner) are not computable on the client side in reasonable time without negatively affecting the user experience. The current focus of our work is the development of schemes to balance the work of these computationally more challenging tasks between clients and servers. Furthermore, changing traffic conditions might require an update of the CORE subgraph which is usually transmitted only once to the clients. We are currently developing update strategies avoiding the full retransmission of the CORE each time traffic conditions change.

Acknowledgement

This work was partially supported by the Google Focused Grant Program on Mathematical Optimization and Combinatorial Optimization in Europe.

References

[1] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transportation Science*, vol. 46, no. 3, pp. 388–404, 2012.

[2] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 0029-599X, pp. 269–271, 1959.

[3] R. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks," in *6th Workshop on Alg. Engineering and Experiments (ALENEX'04)*, 2004.

[4] H. Bast, S. Funke, and D. Matijevic, "Ultrafast Shortest-Path Queries via Transit Nodes," *DIMACS Series in Discr. Math. and Theor. Computer Science*, vol. 74, pp. 175–192, 2009.