Studying the technique of Contraction Hierarchies for routing on Web-Scale

Tim Vanhee
Student number: 01605252

Supervisors: , Dr. Pieter Colpaert
Counsellors: Ir. Ruben Taelman, Dr. Pieter Colpaert, Julian Andres Rojas Melendez, Brecht Van de Vyvere

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2018-2019

FACULTY OF ENGINEERING
AND ARCHITECTURE

Studying the technique of Contraction Hierarchies for routing on Web-Scale

Tim Vanhee
Student number: 01605252

Supervisors: , Dr. Pieter Colpaert
Counsellors: Ir. Ruben Taelman, Dr. Pieter Colpaert, Julian Andres Rojas Melendez, Brecht Van de Vyvere

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2018-2019

GHENT
UNIVERSITY

Acknowledgements

I would like to thank my supervisor, Dr. Pieter Colpaert and my counsellors Julian Rojas Melendez and Brecht Van de Vyvere for the warm welcome into the research group, the much needed feedback and helpful information when I needed it. Also a thank you to Harm Delva for having multiple meetings with me to face certain problems and work out solutions. Ben Abelshausen for the useful discussions on the subject and the Open Belgium Conference for providing a more in-depth insight into open and linked data.

Finally I would also like to thank my fellow students for the interesting conversations. My brother, Sam Vanhee, for proof-reading my entire thesis, my girlfriend and all my friends and family for supporting me and providing feedback.

Thank you, Dr. Pieter Colpaert for giving me the opportunity to be able to work on such an interesting topic and challenging my knowledge, abilities and myself as a person, for which I am very grateful.

Tim Vanhee

Kortrijk, June 6, 2019

Permissions

Abstract

Performing routing over the web faces different challenges due to the distributed nature of the data. Long-distance routing with such data requires bandwidth and processing power, proportionate to the distance. The correct data must be downloaded to calculate the shortest path between two points. Which data this is, is only known when the shortest path is found. Therefore, more data must be downloaded than just the shortest path. To prevent that a client must download all the data that encompasses the begin and end of the route, a speed-up is proposed: contraction hierarchies. This algorithm calculates shortcuts between two certain points and adds these to the data set. By removing the data that the shortcut bypasses and implementing this algorithm on multiple levels, the total size of downloaded data can be reduced while speeding up the calculations to find the shortest path. **Keywords: Contraction hierarchy, road networks, graph algorithm, shortest path, linked data, web-scale**

Routeren op web schaal brengt de uitdagingen van het routeren zelf en die van een gedistribueerde dataset met zich mee. Routes berekenen over lange afstanden vraagt zowel een bepaalde hoeveelheid bandbreedte alsook de benodigde rekenkracht, afhankelijk van de afstand tussen zowel het start- als eindpunt. Om het kortste pad te berekenen, moeten de correcte data gedownload worden. Men weet pas welke data deze zijn eens het kortste pad gekend is. Hierdoor moet er meer data gedownload worden dan enkel het kortste pad. Om te voorkomen dat de gebruiker alle data moet downloaden die beschreven wordt door de omgeving tussen het start- en eindpunt, wordt er een verbetering voorgesteld: contraction hierarchies. Dit algoritme berekent shortcuts tussen twee bepaalde punten en voegt deze shortcut toe aan de dataset. Door de minder belangrijke gegevens te verwijderen uit de dataset en het contraction hierarchy algoritme toe te passen op meerdere niveaus, kan de totale bandbreedte worden verminderd. Tegelijkertijd wordt ook de totale uitvoeringstijd om het kortste pad te vinden verminderd. **Kernwoorden: Contraction hierarchies, wegennet, graafalgoritmen, kortste pad, linked data, web schaal**

Studying the technique of Contraction Hierarchies for routing on Web-Scale

Tim Vanhee

Supervisor(s): Dr. Pieter Colpaert, Ir. Ruben Taelman, Julian Andres Rojas Melendez, Brecht Van de Vyvere

Abstract— **Performing routing over the web faces different challenges due to the distributed nature of the data. Long-distance routing with such data requires bandwidth and processing power, proportionate to the distance. These problems can be addressed by using contraction hierarchies and a bi-directional Dijkstra algorithm to speed up the execution and preprocessing times as well as reducing the amount of downloaded data when searching for the shortest path over long-distances routes. Because the data set is distributed, tile contractions are calculated on multiple levels, creating a hierarchy of hierarchies, making it possible to reduce the amount of downloaded data. As a result, the file sizes of parent tiles shrink with an average of 23.6% on the regular files and 33.6% on the compressed files. The route planning can be executed in an average time of 550 ms after the graph has been reconstructed. Reconstruction of the tiles takes 2950 ms on average on an intercity route. The file size of the tile on zoom level 9 is around 1.3 MB if compressed and is small enough to be used in a web-scale environment. However, the execution times of the planning algorithm exceed a near-instant time window of 500 ms [1]. The preprocessing time, file sizes and execution times could be reduced significantly by making slight changes in the contraction algorithm.**

Keywords— **Contraction hierarchy, road networks, graph algorithm, shortest path, linked data, web-scale**

I. Introduction

This article is situated in a environment where linked open data, bandwidth and scaling is key to battle growing needs for data to be open and available for everyone at any time, regardless of the amount of requests. An API-endpoint that scales proportionately with an increasing load or one that is able to expand the amount of used data sets requires more than a basic setup. Certain resources are in heavy demand on the web e.g. social media or entertainment content. An other widely used content is road data sets. They are used to calculate the shortest, most efficient routes or even routes with certain characteristics and needs. Route planners use these data sets to plot a route depending on the means of transport, ranging from travelling on foot to sail by boat or go by plane. To calculate the shortest route between a start and end, all the necessary information to determine the result is needed. Since the data set has a distributed nature, it is important to ask ourselves, what data do we need and where do we get it?

An easy and effective way to tackle the increasing load is to use caching[2][3]. Solving the problem of wanting to expand a data set with multiple others can be done using linked data[4][2].

The road data is divided into small pieces called tiles and could be distributed across multiple locations. These tiles contain all necessary information for the roads it encompasses. These tiles are published in such a way that they take advantage of both ends of the spectrum defined in Figure 1. By using linked data, it is possible to determine which information is needed based on the related data that is associated with it.

The main questions that this article answers are the following:

Question 1: How can a route planner be optimized when routing over long distances?

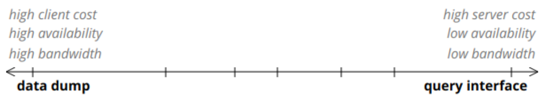


Fig. 1. Spectrum of exposed data[4][2]

Question 2: How do we reduce the amount of data that the client downloads to calculate the shortest path?

To ensure that the questions above are measured correctly, each question will be coupled with one or more hypothesis. The hypothesis are the following:

1. The time between starting the query and supplying the client with an answer cannot take too long. According to [1], in order for a human to perceive a result as instantaneously, the delay may not exceed 500 ms.
2. The application used as a basis for this thesis is Routable Tiles. The user cannot download more data with this solution in comparison to the Routable Tiles application.
3. Caching must be allowed so that repeated request can be answered without the need to contact the server again.

II. Related work

In recent years, much research is done to improve the overall speed of calculating the shortest path between two points. Research has already been performed on contraction hierarchies itself[5, 6]. Time-dependant models have been developed[7] and extensions of these models that allow distributed memory parallelization[8]. Some techniques allow the contraction hierarchies to be metric-independent, giving the contractions the freedom to tweak the metrics in a specific phase and adjust them to current road conditions or traffic jams[9, 10].

All the research performed in the papers mentioned above have the same assumption that the data set is present on the server or is distributed to increase computational performance. The context of this paper is that not all data sets are present on the server at the time of preprocessing. Because of this, most of these solutions are not applicable in this context.

First, the issues this research faces are uncovered. In the next section, the possible techniques to solve these problems are elaborated. Following this, the actual implementation is discussed. After this, the results are gathered and analyzed. Finally, these results are put into perspective and a conclusion is formed.

III. Data set format

Using the data dump format to transport large sets of data requires a high bandwidth and enough client processing power to process this data and use it in an application. Using the API format to expose large sets of data requires a server, which must be able to scale according to the changes in load and have a high availability, making the API a single point of failure. Both ends of the spectrum have their advantages and disadvantages. The point is to find the sweet spot, where advantages outweigh the disadvantages.

IV. Basic idea

This article investigates how the large data set of small pieces of road data can be grouped into bigger, more dynamic parts. These parts function as small data dumps and are exposed through an API, taking advantage of both parts of the spectrum. The initial division of the data sets are already defined as *tiles*[11]. These tiles are defined by an x, y and z coordinate. The z coordinate denotes the zoom level, which is a number that signifies how big the tile is. The smallest zoom level has the highest number. The smallest zoom level is 14.

To tackle the bandwidth issue, the sizes of each tile must be reduced as much as possible. This is done using *Gzip*.

These road data sets can be represented in a graph structure with nodes and edges. At the time of writing, all tiles from zoom level 14 are made publicly accessible by the Open Planner Team[12] on the following URL

https://tiles.openplanner.team/planet/14/{x}/{y}

By using contraction hierarchies, as proposed by [5], it is possible to introduce shortcuts in graph structures. Generating the contraction hierarchies for each tile makes the routing faster because it is able to skip multiple nodes, just by following a single shortcut.

To increase routing speed even more, the tiles on higher zoom levels can be generated using tiles on a lower zoom level combined with the contraction hierarchy algorithm.

V. Solution architecture

As a first step to reduce the sizes of the tiles, a first simplification is proposed in this article. By removing all data of road sections that are not part of an intersection, the size of the tiles can be reduced significantly. This reduces bandwidth use and speeds up the calculation of contractions and routing. The execution time of both the contraction algorithm and a regular Dijkstra algorithm depend on the amount of nodes present in the graph.

To generate a tile on zoom level z, the four tiles on zoom level $z + 1$ are used. After merging the tiles, parts of the bigger tile are removed to make the tile smaller in size. Only the nodes outside the bounding box of the bigger tile and the nodes outside each bounding box of the underlying tiles are retained. Next, the shortest paths between each pair of these nodes is calculated. All nodes on this shortest path are kept. This ensures that the client is able to route over this tile from all directions. Next the shortcuts and the nodes connected to a shortcut are preserved as well. Finally the contractions are calculated and the tile is saved to the file system, ready to be served to a client requesting it.

VI. Results and findings

File sizes depend largely on the location and its contents. If the file contains a small village, the size will be significantly smaller compared to a tile containing a metropolis. The tiles on zoom level 13 that contain parts of Ghent are approximately between 2 and 2.6 MB in size each. This is more than 20 times as much as the tile containing containing parts of Machelen and Olsene. The results discussed below are averages calculated from the tile with ID: x=261, y=171 on zoom level 9. 422 tiles on different zoom levels were generated, excluding the tiles on zoom level 14, which are not altered in any way. Generating these tiles took a little bit more than 9 hours to complete on an Intel Core i5-4210U @ 1.70 GHz. This large figure may be due to the fact that all of the calculations are written in JavaScript, are done synchronously and JavaScript only uses a single thread. Table I shows the average size of the generated tiles.

Zoom level	Amount	Size (kb)	Compressed size (kb)
14	23045	75.59	10.69
13	320	296.19	25.6
12	80	1190.95	89.25
11	17	2540.47	193.41
10	4	8022	604
9	1	16428	1260

TABLE I
Average tile sizes

When calculating a route between Sint-Niklaas and Oudenaarde, the amount of data downloaded is 40 MB. The Routable Tiles demo[13] downloads over 239 MB for that same route. All these values are reduced significantly when the compressed versions are fetched. The total amount of downloaded data is reduced to 0 if the tiles are already in the cache. The reason these values are so big, is because the bounding box of this route includes larger cities. Wallonia is less dense and therefore greatly impact the average.

The Routable Tiles demo downloads all the zoom level 14 tiles in the bounding box around the start and end.

The most computationally intensive task is to fetch and reconstruct the graph from all the different JSON-files from all the tiles. On average, fetching the tiles takes 2950 ms and calculating the shortest path takes 550 ms on average.

VII. Conclusion

A. Preprocessing times

Calculating the zoom level 9 tile that encompasses the city of Ghent takes approximately 7 hours. Because all the edge nodes of the underlying tiles are retained as well as a shortest path between all pairs of these nodes is calculated, the preprocessing times increase dramatically. However, this step is necessary to retain enough data as to keep routing possible. A slight error or loss of data propagates itself into higher zoom levels, creating significant data loss. In this implementation, the data loss only starts showing from a zoom level 9, depending on the region. Calculating a tile on zoom level 12 or 13 takes roughly 500 ms to 2500 ms for a dense area like Brussels. The preprocessing times, in this software implementation, are not viable on a global scale but could be viable on a national or local scale.

B. Bandwidth

The file sizes are within a reasonable range if they are compressed. The compressed file on zoom level 9 only takes up around 1.2 MB. The uncompressed file is 16.8 MB. If a client would download all tiles on zoom level 14 to find a shortest path, a total average of 76.8 MB should be downloaded and processed by the client. Depending on the best-case results and worst-case results, this implementation offers a better solution on long distances in comparison to downloading all the zoom level 14 tiles in the bounding box around start and end. Figure 2 show the comparison between the worst-case bandwidth usage and the usage of the Routable Tiles application. Figure 3 shows the best-case scenario bandwidth usage. These values differ greatly from the 239 MB mentioned in the previous section because the averages are used in these graphs.

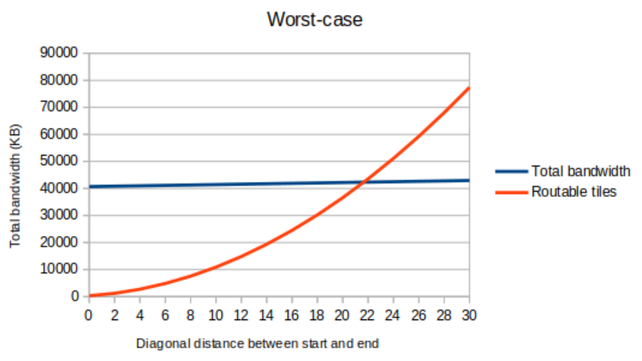


Fig. 2. Worst-case comparison to the Routable Tiles application

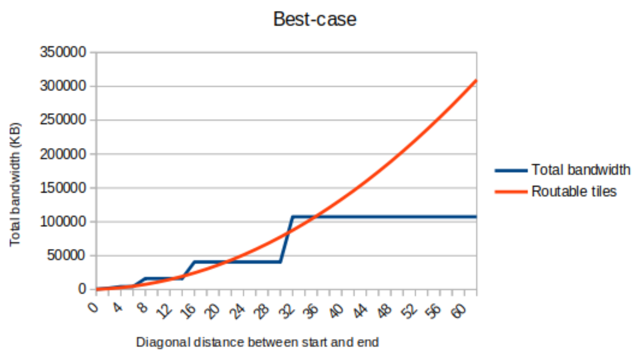


Fig. 3. Best-case comparison to the Routable Tiles application

C. Execution times

The speed of the routing algorithm, around 550 ms, to make an intercity route, is a little above the standard set of 500 ms. An end user perceives a delay of 500 ms [1] as near-instant. The solution, which gives an estimated 550 ms, does not fulfill this requirement.

VIII. Perspective

When putting these results into perspective and context, the values depict some other meanings. Small improvements to the file sizes and routing algorithm could make this proposal viable when considering that all data must be transferred to, and be processed by, the client. The most important piece of this puzzle is the contraction algorithm. Slight changes made to the algorithm can have a significant impact on the remaining data and how the structure is saved and transferred to the client. The algorithm determines how much data is kept and thus how large the files will be. In turn, this influences how much data the client downloads and how big the graph is when trying to find a shortest path.

References

[1] Y. Chen, T. Farley, and N. Ye, “Qos requirements of network applications on the internet,” *Information Knowledge Systems Management*, vol. 4, no. 1, pp. 55–76, 2004.

[2] P. Colpaert, “Publishing transport data for maximum reuse,” UGent, 2018. [Online]. Available: https://phd.pietercolpaert.be/

[3] R. T. Fielding, “Architectural styles and the design of network-based software architectures.” University of California, 2000.

[4] L. D. F. collaborators. [Online]. Available: http://linkeddatafragments.org/

[5] R. Geisberger, “Contraction hierarchies: Faster and simpler hierarchical routing in road networks,” Karlsruhe Institute of Technology, p. 70, 2008.

[6] 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.

[7] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, “Time-dependent contraction hierarchies,” in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 2009, pp. 97–105.

[8] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter, “Distributed time-dependent contraction hierarchies,” in *International Symposium on Experimental Algorithms*. Springer, 2010, pp. 83–93.

[9] J. Dibbelt, B. Strasser, and D. Wagner, “Customizable contraction hierarchies,” in *International Symposium on Experimental Algorithms*. Springer, 2014, pp. 271–282.

[10] ——, “Customizable contraction hierarchies,” *Journal of Experimental Algorithmics (JEA)*, vol. 21, pp. 1–5, 2016.

[11] *Tiles*, OpenStreet Map. [Online]. Available: https://wiki.openstreetmap.org/wiki/Tiles

[12] [Online]. Available: https://openplanner.team/

[13] [Online]. Available: https://openplannerteam.github.io/leaflet-routable-tiles/

Onderzoek naar Contraction Hierarchies voor routering op web schaal

Tim Vanhee

Begeleiders(s): Dr. Pieter Colpaert, Ir. Ruben Taelman, Julian Andres Rojas Melendez, Brecht Van de Vyvere

Abstract—**Routeren op web schaal brengt zowel de uitdagingen van het routeren zelf als die van een gedistribueerde dataset met zich mee. Routes berekenen over lange afstanden vraagt zowel een bepaalde hoeveelheid bandbreedte alsook de benodigde rekenkracht, afhankelijk van de afstand tussen zowel het start- als eindpunt. Deze problemen kunnen aangepakt worden door gebruik te maken van contraction hierarchies en een bidirectionele versie van het algoritme van Dijkstra. Dit versnelt zowel de uitvoertijden en de tijd die nodig is voor de preprocessing als de hoeveelheid data die gedownload wordt wanneer het kortste pad over lange afstanden gezocht wordt. Door de gedistribueerde aard van de data, worden tile contracties berekend op meerdere niveaus via het contraction hierarchy algoritme. Na het hiërarchisch samenvoegen en het contracten van de onderliggende tiles, wordt de grootte van de gewone bestanden verminderd met een gemiddelde van 23,6% en die van de gecomprimeerde bestanden met een gemiddelde van 33,6%. Het berekenen van een route neemt gemiddeld 550 ms in beslag nadat de graaf gereconstrueerd is. Het reconstrueren en ophalen van de tiles duurt gemiddeld 2950 ms voor een route tussen twee steden. De bestandsgrootte van de tile op zoom level 9 is ongeveer 1,3 MB indien deze compressed is, wat niet over het hoofd gezien mag worden. De uitvoeringstijd neemt meer dan de beoogde 500 ms in beslag. Door het contraction algoritme te verbeteren, kunnen de preprocessing tijden, bestandsgroottes en uitvoeringstijden significant verlaagd worden. Dit heeft als resultaat dat deze methode om routes op lange afstanden te berekenen een haalbare oplossing kan zijn.**

Keywords—**Contraction hierarchies, wegennet, graafalgoritmen, kortste pad, linked data, web schaal**

I. Context

Dit artikel is geplaatst in een context waar linked open data, bandbreedte en schaling zeer belangrijk zijn om de groeiende nood aan informatie, die vrij toegankelijk is zonder beperkingen, te vervullen. Om een API te maken die proportioneel meeschaalt met een toenemende belasting of een groeiende dataset, is er meer nodig dan een simpele webserver die luistert naar requests.

Resources zoals sociale media of Netflix worden zeer intensief op het web gebruikt. Een ander voorbeeld van een veel gebruikte resource zijn de gegevens van het wegennet. Ze worden gebruikt om de kortste of de snelste route te berekenen die moet voldoen aan bepaalde criteria. Routeplanners van tegenwoordig berekenen de kortste route afhankelijk van het vervoersmiddel. Dit kunnen routes zijn die over de fysieke wegen gaan of zelfs routes per boot of vliegtuig. Om de kortste route te berekenen tussen een start- en eindpunt heeft men de juiste informatie nodig. Nu vragen we ons af welke gegevens we precies nodig hebben en waar we deze kunnen vinden.

Een makkelijke en effectieve manier om het probleem van een toenemende belasting op te lossen is via caching[1, 2]. Het probleem van een dataset die stijgt in grootte door er andere datasets in te integreren, kan opgelost worden met linked data[3, 2]. De gegevens van het wegennet zijn verdeeld in kleine stukjes die tiles worden genoemd. Deze kunnen gedistribueerd opgeslagen zijn over meerdere locaties. Een tile bevat alle nodige informatie over de wegen die er in gelegen zijn. Al deze tiles worden zodanig gepubliceerd dat ze gebruik maken van de voordelen van de beide uiteinden van het spectrum zoals getoond in Figuur 1.

Dit artikel beantwoordt de volgende onderzoeksvragen:

Onderzoeksvraag 1: Hoe kan een routeplanner geoptimaliseerd worden, zodat routes over lange afstanden sneller berekend kunnen worden?

Onderzoeksvraag 2: Kan de hoeveelheid data die de gebruiker downloadt om het kortste pad te vinden, verminderd worden?

Om op deze vragen een objectief antwoord te kunnen formuleren, worden er twee hypothesen geformuleerd.

1. De tijd tussen het starten van de query en het ontvangen van het antwoord mag niet langer dan 500 ms zijn. Volgens [4] wordt een vertraging onder de 500 ms ervaren als instant.
2. De oplossing mag niet meer gegevens downloaden dan de Routabel Tile applicatie, die gebruikt werd als uitgangspunt van dit onderzoek.
3. Caching moet toegelaten worden, zodat bij herhaalde request de server niet opnieuw gecontacteerd moet worden.

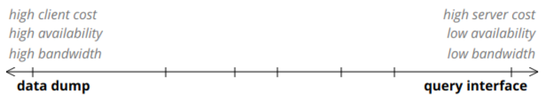


Fig. 1. Spectrum van gepubliceerde data[3][2]

II. Verwant werk

Tijdens de voorbije jaren werd al veel onderzoek gedaan om de algemene efficiëntie van het berekenen van een route tussen twee punten te verbeteren. Er is al onderzoek gedaan naar contraction hierarchies[5, 6] zelf. Tijdsafhankelijke modellen ervan werden ook onderzocht en uitgewerkt[7] met bijkomende uitbreidingen om parallellisatie te voorzien tijdens het berekenen[8]. Een uitbreiding op contraction hierarchies zelf laat toe om het volledige algoritme onafhankelijk te maken van de gewichten om deze daarna dynamisch te kunnen updaten. Dit maakt het mogelijk dat de routeplanner rekening kan houden met vertragingen of files[9, 10].

III. Formaat van de data

Door het datadump formaat te gebruiken, moeten grote hoeveelheden gegevens verzonden en ontvangen worden. Dit vraagt een bepaalde bandbreedte en verwerkingskracht van de gebruiker. Het API formaat heeft als probleem dat alles op de server berekend wordt en dat de server moeilijk mee kan schalen met een plotse toename in belasting. Indien de API op een bepaald moment niet beschikbaar is, kan de applicatie niet meer werken, wat van de API een single point of failure maakt. Beide uiteinden van het spectrum hebben zowel hun voordelen als nadelen. De bedoeling is om de gulden middenweg te vinden waar de voordelen van beide tot hun recht komen terwijl de nadelen beperkt blijven.

Een bijkomend voordeel van linked open data is dat iedereen een routeplanner kan maken zonder specifieke data te moeten verzamelen.

IV. Werkwijze

Dit artikel onderzoekt hoe de grote hoeveelheid data, die opgesplitst is in kleine stukjes, kan gegroepeerd worden in meer dynamische stukken. Deze stukken fungeren als kleine datadumps en worden beschikbaar gesteld via een API. Op deze manier wordt er geprobeerd om een gulden middenweg te zoeken in het spectrum. De eerste opdeling van de dataset is al gedefinieerd en worden *tiles*[11] genoemd. Deze tiles worden gekenmerkt door een x, y en z coördinaat. De z coördinaat stelt het zoom level voor en bepaald hoe groot de tile is. Het kleinste zoom level heeft het grootste getal, namelijk 14.

Om de problematiek rond de bandbreedte aan te pakken, moeten de groottes van de tiles zoveel mogelijk beperkt worden zonder te veel data te verliezen. *Gzip* wordt toegepast om de bestandsgrootte van een tile nog verder te verkleinen door het te comprimeren.

De gegevens van het wegennet kunnen voorgesteld worden in een graaf met knopen en takken. Op het moment van schrijven, zijn alle tiles van zoom level 14 gepubliceerd op de publieke API van het Open Planner Team[12]. De API heeft de volgende structuur

https://tiles.openplanner.team/planet/14/x/y

Door gebruik te maken van contraction hierarchies, die geïntroduceerd is door [5], is het mogelijk om shortcuts toe te voegen aan een graaf. Door gebruik te maken van shortcuts wordt het vinden van het kortste pad versneld. Dit komt doordat er meerdere knopen overgeslagen worden bij het volgen van een shortcut.

Om de snelheid van de routeplanner nog meer te versnellen, kunnen de tiles op een hoger zoom level gegenereerd worden door tiles van een lager zoom level samen te nemen en daar het contraction hierarchy algoritme op toe te passen.

V. Oplossing

Dit artikel stelt een eerste stap voor om de bestandsgrootte te beperken, namelijk door alle tussenliggende punten op rechte wegen te verwijderen. Bij een routeplanner zijn enkel de kruispunten van belang. Hierdoor kunnen de groottes van de bestanden significant verkleind worden. Dit heeft als voordeel dat er minder data gedownload zal worden en de snelheid van zowel het algoritme om het korste pad te berekenen als het contraction algoritme toeneemt. Beide algoritmen zijn afhankelijk van de grootte van de graaf waar het algoritme op uitgevoerd wordt.

De tiles op een lager gelegen zoom level worden gebruikt om de tiles op een hoger zoom level te bepalen. Na het samenvoegen van de onderliggende tiles wordt de samengevoegde tile verkleind, zodat de bestandsgrootte afneemt. De knopen buiten de bounding boxes van de onderliggende tiles en alle knopen die op het kortste pad liggen tussen elk paar van deze knopen, worden gespaard. Dit zorgt ervoor dat de gebruiker nog steeds kan routeren over een bepaalde tile vanuit elke richting. Als laatste wordt het contration hierarchy algoritme uitgevoerd en wordt het resultaat opgeslagen op de schijf.

Zoom level	Aantal	Grootte (kb)	Gecomprimeerd (kb)
14	23045	75.59	10.69
13	320	296.19	25.6
12	80	1190.95	89.25
11	17	2540.47	193.41
10	4	8022	604
9	1	16428	1260

TABEL I
GEMIDDELDE TILE BESTANDSGROOTTES

VI. Resultaten en bevindingen

De bestandsgroottes zijn grotendeels afhankelijk van de locatie en de omgeving. De tile die een dorpje bevat, zal significant kleiner zijn dan een tile die een grotere stad bevat. De tiles op zoom level 13 die een deel van Gent bevatten zijn tussen de 2 en 2,6 MB groot. Dit is meer dan 20 keer zo veel als de tiles die Machelen of Olsene bevatten. De onderstaande resultaten zijn gemiddelden die berekend zijn van de tile met ID $x = 261$ en $y = 171$ op zoom level 9 en alle onderliggende tiles. Hiervoor werden 422 tiles gegenereerd op verschillende zoom levels op basis van 2560 tiles op zoom level 14. Het genereren van de tiles duurde iets meer dan 9 uur op een Intel Core i5-4210U @ 1.70 GHz. Dit grote getal kan het gevolg zijn van de JavaScript implementatie die synchroon uitgevoerd wordt op een enkele thread van een niet zo krachtige CPU. Tabel I bevat de gemiddelde groottes van de tiles die gegenereerd werden op de verschillende zoom levels.

Wanneer men een route berekend tussen Sint-Niklaas en Oudernaarde wordt er ongeveer 40 MB door de gebruiker gedownload. De Routable Tiles demo[13] downloadt voor deze zelfde route meer dan 239 MB. Deze waarden kunnen nog verder verkleind worden door compressie toe te passen. De hoeveelheid kan zelfs teruggebracht worden naar 0 MB als caching toegelaten wordt en alle tiles zich in de cache bevinden. De reden dat de Routable Tiles demo zo veel gegevens downloadt, komt doordat de waarden in Tabel I de gemiddelde waarden bevat en omdat het alle tiles downloadt in de bounding box rond het begin en einde. Wallonië, dat een pak minder dichtbevolkt is dan Vlaanderen, trekt dit gemiddelde fors naar beneden.

De taak die het meeste tijd en rekenkracht vraagt, is het ophalen en opnieuw reconstrueren van de graaf op basis van alle opgehaalde JSON bestanden. Het ophalen van de tiles neemt gemiddeld 2950 ms in beslag en het berekenen van het kortste pad 550 ms.

VII. Conclusies

A. Duur van de preprocessing

Het berekenen van de tile op zoom level 9 die de stad Gent bevat, neemt ongeveer 7 uur rekenwerk in beslag. Omdat het kortste pad berekend wordt tussen elk paar knopen die buiten de bounding boxes vallen van de onderliggende tiles, nemen de uitvoeringstijden drastisch toe. Deze stap is wel noodzakelijk om zeker te zijn dat de gebruiker nog over de tiles kan routeren. Een kleine fout of tekort in een bepaalde tile be¨nvloedt de data op een bovenliggend zoom level, waardoor er veel gegevens verloren kunnen gaan. In deze implementatie wordt, afhankelijk van de regio, het dataverlies pas zichtbaar vanaf zoom level 9. Het berekenen van een tile op zoom level 12 of 13 neemt respectievelijk 500 ms en 2500 ms in voor een drukbevolkte regio als Brussel. In deze implementatie is de duur van de preprocessingsfase niet bruikbaar op grote schaal, maar wel op nationale of lokale schaal.

B. Bandbreedte

De bestandsgroottes zijn aanvaardbaar eens ze gecomprimeerd zijn. Het gecomprimeerde bestand op zoom level 9 vraagt slechts 1,2 MB aan schijfruimte terwijl het originele bestand er 16,8 MB vraagt. Als een gebruiker alle tiles downloadt van zoom level 14 om het kortste pad te vinden, moet hij gemiddeld 76,8 MB downloaden en verwerken. Afhankelijk van de resultaten van de best-case en worst-case situaties, biedt deze implementatie een betere oplossing dan dat alle tegels van zoom level 14 gedownload worden. Figuur 2 toont de vergelijking tussen de worst-case situatie en de Routable Tiles demo op vlak van gebruik van bandbreedte. Figuur 3 toont de vergelijking van de best-case situatie en de Routable Tiles demo.

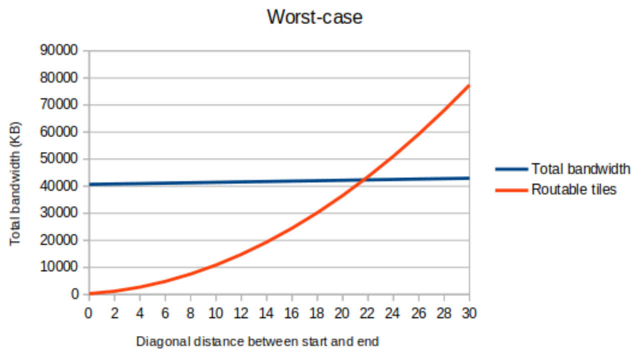


Fig. 2. Vergelijking tussen de worst-case situatie en de Routable Tiles application

C. Duur van de query

De tijd die het routeringsalgoritme in beslag neemt, is gemiddeld 550 ms voor een route tussen 2 naburige steden zoals Gent en Antwerpen. Dit getal bevindt zich boven de near-instant gebruikerservaring van 500 ms. De implementatie voldoet dus niet aan deze verwachting.

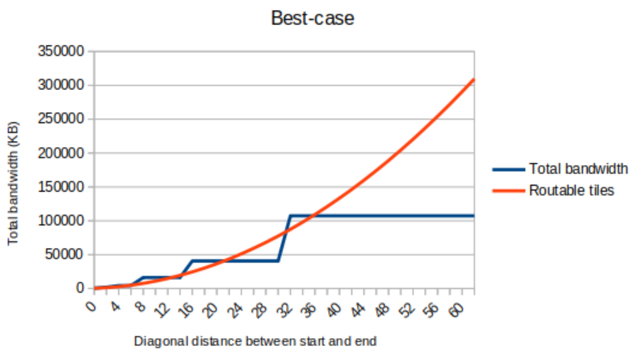


Fig. 3. Vergelijking tussen de best-case situatie en de Routable Tiles application

VIII. Perspectief

Wanneer deze resultaten in perspectief en context geplaatst worden, krijgen ze een andere betekenis. Kleine verbeteringen in de bestandsgrootte en het routeringsalgoritme kunnen deze oplossing bruikbaar maken. Het belangrijkste deel van deze resultaten is het contraction hierarchy algoritme. Kleine veranderingen in het algoritme kunnen een grote impact hebben op de hoeveelheid resterende data en hoe groot de bestanden zullen zijn. Dit zal op zijn beurt een invloed hebben op de hoeveelheid data die de gebruiker moet downloaden, hoe groot de graaf is en hoe lang het duurt om het kortste pad te vinden.

References

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures." University of California, 2000.

[2] P. Colpaert, "Publishing transport data for maximum reuse," UGent, 2018. [Online]. Available: https://phd.pietercolpaert.be/

[3] L. D. F. collaborators. [Online]. Available: http://linkeddatafragments.org/

[4] Y. Chen, T. Farley, and N. Ye, "Qos requirements of network applications on the internet," *Information Knowledge Systems Management*, vol. 4, no. 1, pp. 55–76, 2004.

[5] R. Geisberger, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," Karlsruhe Institute of Technology, p. 70, 2008.

[6] 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.

[7] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-dependent contraction hierarchies," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 2009, pp. 97–105.

[8] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter, "Distributed time-dependent contraction hierarchies," in *International Symposium on Experimental Algorithms*. Springer, 2010, pp. 83–93.

[9] J. Dibbelt, B. Strasser, and D. Wagner, "Customizable contraction hierarchies," in *International Symposium on Experimental Algorithms*. Springer, 2014, pp. 271–282.

[10] ——, “Customizable contraction hierarchies,” *Journal of Experimental Algorithmics (JEA)*, vol. 21, pp. 1–5, 2016.
[11] *Tiles*, OpenStreet Map. [Online]. Available: https://wiki.openstreetmap.org/wiki/Tiles
[12] [Online]. Available: https://openplanner.team/
[13] [Online]. Available: https://openplannerteam.github.io/leaflet-routable-tiles/

Contents

List of Figures

List of Tables

Listings

List of abbreviations

CCH Customizable Contraction Hierarchies. 37

CH Contraction Hierarchies. 32–34, 37, 45

ES6 ECMAScript 6. 42

HH Highway Hierarchies. 31, 32

HNR Highway Node Routing. 31, 32

JSON JavaScript Object Notation. 49, 54

LOD Linked Open Data. 24, 26, 27

MSMD Multiple Source Multiple Destinations. 55

MSSD Multiple Source Single Destinations. 55

OSM OpenStreet Map. 38–40

RDF Resource Description Framework. 26, 27

SSMD Single Source Multiple Destinations. 55

TNR Transit Node Routing. 31

URI Uniform Resource Identifier. 25–27

1 Introduction

Searching for answers, communication, services and sharing knowledge are all applications of the web. More and more is digitized and stored on the web[1]. Nowadays, the ever growing World Wide Web is indispensable and rooted in all aspects of our daily lives. It is used for a number of applications and one very frequently used application is answering questions.

Answering questions or queries is a fairly complex task. The computational power needed depends on the type of answer one expects. Answering a question in full is quite difficult, because it requires data from different contexts. If someone wanted to know how to get from one point to another using public transport, the data from all different public transport companies is needed to provide an answer. These data sets are not guaranteed to be interoperable and thus each data set must be transformed into something with standards and consistency. Linked data[2] avoids this issue by setting standards for all data sets that are published as linked data. This allows the different data sets to be interoperable, contain semantics and have connections to other sets.

Querying a navigation planner is a type of query that is widely used all over the world. A navigation planner deals with different sets of data and requires processing power to compute an answer for the proposed routing question. This thesis deals with some techniques that can be used to speed up the navigation process when computing navigation queries while the data sets are published on web-scale.

1.1 Goal of the thesis

The goal of this thesis is to research techniques to speed up long distance routing problems on web-scale. These techniques may already exist and can be modified for this specific use case or they can be completely new techniques. This thesis will only cover existing techniques that may be modified to fit this use case.

The environment variables that define this thesis are quite specific. Because the conditions are so specific, little research on this specific topic could be found during the literature study but some of the techniques discussed have already been researched. These variables include data sets published in a distributed manner that allow scaling on web level and a semantic definition of all object and entities is required. Additionally, all data sets are woven together as part of the linked data principle. However, this thesis combines those different conditions and this, in turn, makes the topic unique.

1.1.1 Research questions

The main questions that this thesis answers, are the following:

1. How can a route planner be optimized when routing over long distances?
2. How can we reduce the amount of data that the client downloads?

1.1.2 Hypotheses

To measure how qualitative and quantitative the results are, three hypotheses are defined using the research questions mentioned above. The results will be matched against these hypotheses in the conclusion. The three hypotheses are as follows:

1. The route planner's execution times lay within an acceptable time window of 500 ms[3], which is experienced as near-instant by the end-user.
2. The client downloads less data when using the proposed solution in comparison to the Routable Tiles application.
3. Caching is supported.

All information and code can be found at: https://github.com/tvhee/Contracted-Tile-Hierarchies

2
Basic concepts

2.1 Data on web-scale

According to [4], open data sets are data sets from which the data can be accessed, used, modified and shared for any purpose by anyone. Many instances already publish certain data sets as open data or host programming competitions based on these data sets. The beauty of open data is its transparency and reusability. Because it is open data, a single data set can be used in all kinds of applications and sectors. The Open Data Portal from Gent can serve as a good example. One of the sets they publish is the amount of free parking spaces in real-time. This set can be used in an application to find a parking spot, but also to analyse parking behaviour to improve the cities parking spaces and city layout in term of shops and traffic.

Publishing data on a web-scale level is key to ensure that the published data can scale appropriately to its demand. Ideally you want to be able to cope with the challenges that arise if the amount or difficulty of questions increase and when more information is added to the data set. Caching[5] makes it possible to make the API, that publishes the data set, scales better with an increase in load. Data sets that are part of an ecosystem called Linked Open Data (LOD) [2], help solve the problem that new data sets are not interoperable with the current set.

2.2 Linked open data and Resource Description Framework (RDF)

Linked data is a term coined by Tim Berners-Lee et al. [2], one of the main inventors of the World Wide Web. linked open data is linked data that is published with an open license, removing some or all restrictions for reuse. The term defines a set of core principles that indicate best practices to publish data in a structured manner. The four core principles are[6]:

1. Use URIs for names of things.
2. Use HTTP-URIs so that people can lookup those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).
4. Include links to other URIs, so that they can discover more things.

The easiest way to explain the 4 principles is with an example. Consider us using “Pizza Hawaii is baked in an oven”, as an example for all the following core principles. A more in-depth explanation can be found here [7].

The first item describes that URIs must be used as names for things. If we transform our data into a CSV-format, depending on how the data was serialized, we could have the following result:

Listing 2.1: a possible CSV-notation

```
id, prepared_in
Pizza Hawaii,Oven
```

How does an instance use URIs instead of regular names and why? If the object is defined as a URI and the URI is published in, for example an online database, the identifier can easily be reused. If the term “Pizza Hawaii” was defined as an URI by a company that produces pizzas, any store or instance that sells this type of pizza can reuse this URI to define their ‘thing’. This makes object definitions unambiguous: How is an object represented by a URI? DBpedia[8], a crowd-sourced community effort that serves linked data, already contains a definition for a “Pizza Hawaii” and an “Oven”. The names in our example can be transformed to a more uniform way of describing this recipe. Any ambiguous interpretation is now nearly excluded.

Listing 2.2: Example URIs for a Hawaiian Pizza and an Oven

```
http://dbpedia.org/resource/Hawaiian_pizza
http://dbpedia.org/resource/Oven
```

The second item describes the use of HTTP-names. The URIs mentioned in the previous item are already HTTP-URIs. When browsing to the mentioned URIs, anyone or anything can look

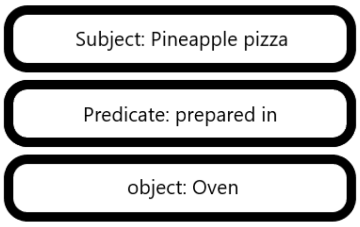


Figure 2.1: An RDF triple

up the mentioned objects. The instance that defined these items is able to further define these objects and specify attributes or specifications. In our example, the “Pizza Hawaii”-URI could mention its recipe, origin or even varieties of this kind of pizza. The same applies to the object representing an Oven.

As third item, Tim Berners-Lee defined that a URI should provide useful information using the standards e.g. RDF*[6]. RDF, the Resource Description Framework, is a framework for describing resources with metadata on the internet[9]. RDF is meant to be machine-processable and human-readable and thus it is written in XML. A good published object can serve a HTTP representation to make it human-readable. RDF is used to semantically define objects and its relations. To facilitate this, RDF a triple format. This triple consists of 3 basic items[7]:

1. Object
2. Subject
3. Predicate

If this is applied to our example, the following triple arises as seen in Figure 2.1. On top of the Object, Subject and Predicate, other attributes are allowed and can thus further define this resource.

The final and fourth item states that a resource should include links to other resources. By adding other URIs, machines are capable of discovering more data. If these URIs lead to other data sets, the querying algorithm can iterate over multiple data sets. In contrast to an algorithm using linked data and RDF, a generic query answering service only accesses its own data sets and the data sets that it was programmed to access.

RDF offers the advantage that the programmer does not need to hard-code all specific protocols and data service or data provider specific actions in its client program. This is because all data sets conform to the same standard.

LOD eliminates the restriction of being confined to the data sets present at the time of development. It does not face this restriction because of the way the data is distributed. LOD links all the different entities used in its data set. By using Uniform Resource Identifiers, it enables the intertwining of different data sets. By using the URIs of other entities in the object properties, multiple data sets can be combined to provide even more information. Data can be accessed and added by any instance. When the user agents queries certain resources, it is able to access related information because of the data being intertwined because of its uniform definition. The querying mechanism shifted from a closed world assumption to an open world assumption. It can now access information that was not part of its own data set. All these uniform definitions require a semantic aspect. If no semantic meaning is given to a certain URI, not a single application or user is able to understand what the object specifically is. For this, the Resource Definition Framework, also called RDF. The LOD-conditions are met as long as the RDF-standard is used in those resources.

At the time of writing (2019), more and more client devices are connected to the internet[10]. Those devices have access to more and more processing power. In this dissertation, we aim to utilize this property to its full potential. This results in 2 major advantages. The first major advantage is the reduced load on the server that provides certain services. The second increases user perceived responsiveness. By executing the necessary operations on the client side, the user can be notified on all different stages of execution. This could eliminate the feeling of an unresponsive application. By publishing the data as linked open data, the computational power needed to answer a certain query is shifted from server to client, allowing the servers to scale more evenly and cope with the challenges mentioned above.

2.3 Graph routing concepts

2.3.1 Dijkstra algorithm

Dijkstra's algorithm[11][12] is widely used for finding the shortest path inside graph structures. A shortest path in a weighted graph is the path which has the lowest total cost. The total cost is the sum of all weights that are contained in that path.

Assume a weighted and undirected graph $G = (V, E)$ with V as its collection of vertices and E as its collection of edges. A function $w(a, b)$ defines the weight of the edge which connects node a and node b. Function $d(s, t)$ returns the total cost of the shortest path from node source s to target t. In case the graph is unweighted, each edge can be considered to have a weight of 1 and the shortest path has the least amount of edges.

Presume that the shortest path from a source s to a target t must be calculated in a weighted graph G. Dijkstra's algorithm will start at the node s by visiting it. It considers all neighbors from visited nodes that have not been visited before as a possible next step. A node v will be visited next if the path from the source s to that node is the smallest of all considered nodes. For this, the algorithm uses the following formula:

$$f(n) = g(n-1) + w(n-1, n) \tag{2.1}$$

Where $f(n)$ is a function that describes the cost to node n. Let the node $n-1$ be the predecessor of node n. The cost to go to node n is equal to the cost to go to node $n-1$ with the addition of the weight of the edge going from $n-1$ to n. Function $g(n-1)$ describes the cost of the shortest path to the predecessor node $n-1$. The node with the smallest $f(n)$ will be visited next. This iteration continues until the destination node t has been visited.

Original Dijkstra algorithm

Assume an undirected and unweighted graph as shown in Figure 2.2. Let the red node be the starting point and the blue node the endpoint. If Dijkstra's algorithm calculates the shortest path, it will visit all nodes in a circular form around the starting point. The shape may vary depending on the weights and the graph shape around the starting point. In this example an unweighted and undirected graph is used for the ease of explanation. in Figure 2.2 the area indicated by the purple circle contains all the vertices the algorithm visits. To reach the endpoint node, Dijkstra's algorithm visits a significant amount of nodes that are not a part of the shortest path. In our example, some nodes will be visited by the algorithm and cannot even reach the destination node. This is a significant overhead that is open to improvements.

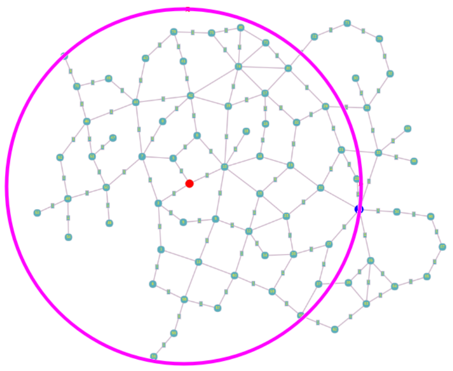


Figure 2.2: Visited nodes by Dijkstra's algorithm

Bidirectional version of Dijkstra's algorithm

[13] One of the improvements does not only calculate the shortest path from source to target, but simultaneously calculates the shortest path from target to source. This is called a bidirectional Dijkstra approach. In Figure 2.3, the 2 purple circles represent the visited nodes by

the bidirectional algorithm. To calculate the surface of a circle, the following formula is used: $A_1 = r^2\pi$. If Dijkstra's algorithm starts from both beginning and end, the circles only have half the radius, resulting in $A_2 = 2(\frac{r}{2})^2\pi$. It is easy to see that $A_1 \geq A_2$.

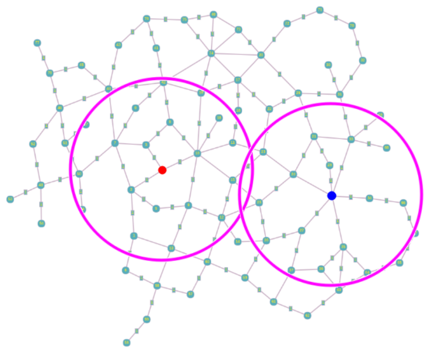


Figure 2.3: Visited nodes by the bidirectional Dijkstra algorithm

2.3.2 Dijkstra's algorithm improvements

The following techniques can speed up Dijkstra's algorithm when searching for a shortest path. The pros and cons of some techniques are discussed and whether or not they are applicable on a web-scale published data set. The impact of most of these techniques was not tested or measured. Some mentioned techniques could increase the speed to a certain degree, but are not always relevant to this thesis' routing problem. Contraction hierarchies will be discussed comprehensively as it is used as a basis for the software implementation.

A* heuristic

The A* heuristic[14] introduces a heuristic function that estimates the remaining distance between a node v and the destination t. By introducing this heuristic function, Formula 2.1 is not applicable and a new formula is introduced:

$$f(n) = g(n) + h(n) \tag{2.2}$$

The A* heuristic must determine which path to follow next. By calculating and sorting the $f(n)$ for all the neighboring and unvisited nodes in an ascending order. The node with the smallest $f(n)$ will be visited next. Function $g(n)$ describes what the cost will be to visit node n and $h(n)$ is an estimation of the remaining distance. For route planning, the Euclidean distance between node n and destination t is a good approach. By using the Euclidean distance, the heuristic function $h(n)$ is just an estimate and not a correct value. The estimate will, if using the Euclidean distance formula, always be an underestimation since the shortest possible path is the Euclidean distance. Further analysis and proof of this heuristic will not be covered in this thesis.

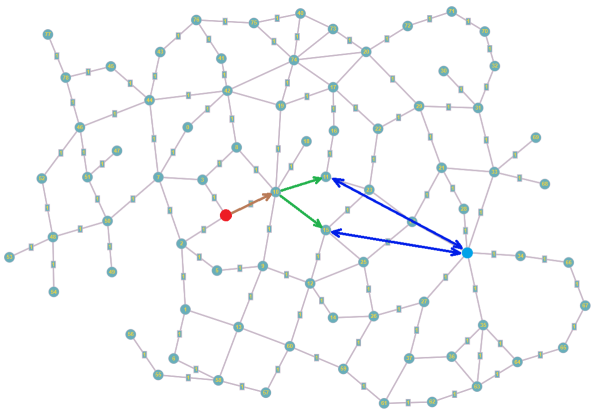


Figure 2.4: Visited nodes with the A* heuristic

In Figure 2.4, let the red node be the start s and the blue node the end node t. The red arrow shows the shortest path at a certain iteration. The heuristic will look at all neighbors of that node, displayed with a green arrow. As stated before, with routing problems, the heuristic function could be the remaining Euclidean distance between the possible next node and the end, displayed as a blue arrow in Figure 2.4. Because both nodes are equally distant from the start node s, the only important term in Formula 2.2 is the heuristic function $h(n)$. Thus the node with the shortest blue arrow will be visited next.

Landmark based A* heuristic

This technique precomputes all distances between a small amount of landmarks and all nodes v[15, 16]. These precomputations are used in combination with the Triangle inequality. Due to the distributed nature of this thesis, knowledge of landmarks outside the immediate surrounding of a node v is not relevant. Further analysis is omitted.

Reach-based routing

Reach-based routing[17, 16] denotes the reach of a node v. Paths can be omitted if the reach is insufficient. Implementing this does not solve the vital long-distance routing problems in a web-scale environment. Further analysis is omitted.

Transit node routing

Transit Node Routing (TNR)[15, 16] assumes that, when calculating a shortest path, the shortest path will most likely enter a set of roads that have faster travel times e.g. a highway or rural road. The nodes that allow access to this faster network are called *transit nodes*. Preprocessing computes the distance between each pair of transit nodes among others.

Highway Hierarchies

Highway Hierarchies (HH) [15, 18, 16], the edges are divided into two categories: highway and non-highway edges. An edge (u, w) is a highway edge if and only if the edge is not fully contained in the area around start s and end t with a radius r. Every node v has a neighborhood radius $r(v)$. These highway edges speed up the calculation of the shortest path . The advantages of Highway Hierarchies are as follows:

1. HHs are significantly faster than Dijkstra's algorithm. Some papers even claim it is over 2.000 times faster.
2. The hierarchy of highway networks allows bidirectional searches.

The disadvantages are as follows:

1. The hierarchies require additional space.
2. Changes to the base structure require the preprocessing phase to be rerun.
3. Preprocessing is required to create the hierarchies.

Highway Hierarchies is a possible option in this use case. Additional research is required to check its viability and performance in a web-scale environment.

Highway Node Routing

Highway Node Routing (HNR)[15, 16] is based on the same premises and principles as HH. In a long-distance trip, fewer different roads are used in the middle section of the trip. One of the differences between HH and HNR is that HNR does not use a fixed neighborhood size that is predetermined, even though the nodes that are used in the hierarchies are. The performance of this algorithm depends on the node selection. The advantages of HNR are as follows:

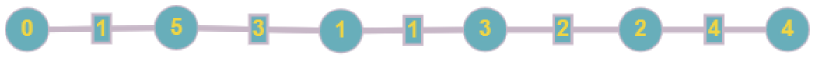


Figure 2.5: A basic graph

1. It is faster than Dijkstra's algorithm.
2. HH is less sensitive to changes in the structure of the graph.

The disadvantages are as follows:

1. Performance is heavily dependant of the node selection.
2. Preprocessing is required to create the hierarchies.

HNR is, just like HH, a possible option. The viability of the solution is not covered in this thesis.

2.3.3 Contraction hierarchies

Contraction Hierarchies (CH) [16, 19, 20, 21, 22, 23] is similar to highway hierarchies. CH can be conceived as a specific case of HH. If we assume a HH, but instead of grouping certain types of roads, every road is its own group of 1. In other words, nothing is grouped and all roads are considered independent. To divide the nodes into different levels (groups), we use a concept called contraction. Contraction is the act of checking if a node can be bypassed by inserting a new edge into the graph. If the graph that needs to be contracted contains n vertices, then the hierarchy contains n levels (groups). Usually, the road network, displayed by a graph, is contracted and a contraction hierarchy is obtained. This is then queried with a bidirectional version of Dijkstra's search algorithm. Contraction hierarchies have two distinct phases: preprocessing and querying. Figure 2.5 will act as an example to illustrate the upcoming explanation.

Preprocessing phase

The preprocessing phase takes a significant amount of time to complete. The time it takes to preprocess a graph is proportionate to its size. Fortunately, this phase only needs a single run for the hierarchies to be calculated. Once calculated, the hierarchies can be reused without another preprocessing phase run. Preprocessing places all nodes in a certain order according to certain conditions. Before continuing, assume a graph $G = (V, E)$ with V its collection of vertices and E its collection of edges. For sorting, assume that every node $v \in V$ has an Importance rating I_v.

This importance describes how interesting it is to contract this node. How this is determined will be discussed further on.

Listing 2.3: Pseudo-code for the contraction hierarchy algorithm

```
for each node $v$ in $V$
    for each neighbor $u$ of node $v$ and with $I_u > I_v$
        for each neighbor $w$ of node $v$ with $w \neq u$ and $I_w > I_v$
            if $\{u, v, w\}$ is the shortest path from $u$ to $w$
                contract node $v$ and insert shortcuts
```

Lets examine the pseudo-code in Listing 2.3 line by line. The first line states an iteration over all nodes inside the collection of nodes. In this iteration, all the nodes $v \in V$ are sorted by their Importance rating I_v. The lower this value, the less important this node is and the more attractive it is to contract it.

Line two and three search for a total of two neighbors of node v that we will potentially contract. Some constraints are in effect. The neighbors cannot be the same node and both nodes need a higher I value, meaning they are more important and thus less interesting to contract. In other words, we want to remove a less important node by strengthening the connection between two more important nodes.

The conditional statement in line four is very important and will be discussed more in depth later on. It requires the path from u to w to be the shortest, otherwise inserting a shortcut does not have an effect. Adding a shorter edge for a path that won't be used because there is an other shorter path available, makes no sense.

At last, if the conditions are met, the node is contracted.

Node iteration and selection

Node order and note selection are probably the most important aspects of CH and highly influence query performance. The order in which the nodes are selected and used in the algorithm define all future actions. Depending on which node was contracted first, the graph structure changes and produces complete other result. According to [16], "Edge Difference" is a promising technique. Edge difference defines the increase or decrease in amount of edges when contracting a node. Every node has this same property and is used as a base for sorting the nodes. The value is calculated as follows: $EdgeDifference = Newedges - Oldedges$. A negative value means that the resulting graph from contracting node v has less edges compared to when that node v was not contracted. [16] also mentioned that a graph contraction using CH is best contracted uniformly. For this, he proposed an additional parameter to use in the node ordering and node selection process. If every node keeps track of the amount of neighbors that have been contracted, the contraction process can be more or less forced to follow a uniform pattern.

Both values are calculated for all the vertices in the graph. Naturally, these values change every time the graph changes its structure. Structural changes in and around a node occur when a neighbor is contracted or edges are removed or introduced. This ultimately changes the order of all vertices. Keeping track of node order and which node to contract first, is (one of) the most computationally intensive tasks of CH. As proposed by [16], there is a way to speed up this process. When a node is contracted, only its neighbors need to be updated and can the process can be sped up. [16] called it lazy-updates and it only updates the nodes that are affected by a change in structure.

Witness paths

Before we can contract a node v with neighbors u and w, it is important that the path from u to w is the shortest. Introducing a shortcut that will not be used is a waste of processing time and space. In [16], witness paths are introduced. When contracting a node, a witness path exists if there is a path $u, ..., w$, not containing v, that is shorter than u, v, w. When such a witness path is found, there is no use in adding the shortcut. If there is no witness path found, u, v, w is the shortest path and a shortcut is useful. To emphasize, a witness path is sufficient and will prevent adding a shortcut. The witness path does not have to be the shortest witness path. The only requirement is that the witness path is shorter than u, v, w.

The process of searching for witness paths and adding shortcuts for a node v is called the contraction of v[16].

Contraction

What does the contraction of a node means? Contracting, as stated before, is the process of checking if there are any witness paths present and check if we can introduce a shortcut.

In the first paragraph of Subsection 2.3.3 the word *removed* is used and in the previous subsection the word *bypassed* is used. Contracting a node does not physically remove it. It is bypassed and shifted to the lowest available level in the hierarchy. However, when executing a query, all nodes with a lower hierarchy-level are ignored, “removing” it if viewed from the perspective of nodes higher in the hierarchy. In Figure 2.6, the graph from Figure 2.5 has been completely contracted. The lower the node, the lower its importance, and thus the lower its level in the hierarchy. The shortcut $(3, 5)$ is introduced when contracting node 1, $(3, 4)$ when contracting the blue node and $(4, 5)$ when contracting node 3.

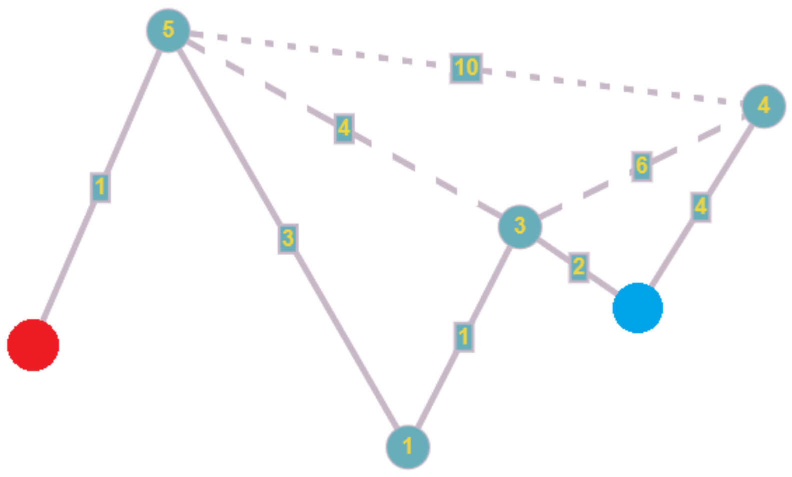


Figure 2.6: Contraction hierarchy of the basic graph

Querying phase

Querying a graph which has been processed by a contraction hierarchy algorithm is basically the same, but with a restriction and complemented by a secondary part.

When querying or routing through the graph, the search starts at the start node s and at the end node t simultaneously. A bidirectional version of Dijkstra's is thus performed. However, when a node v wants to visit or discover all its neighboring nodes, an extra restriction applies. The search is limited to the neighbors that have a higher hierarchy level. This way both start and end paths will climb their way up the hierarchy. At a certain height, both paths constructed from start and end will meet. When they meet, the shortest path has been constructed. However, because we are traversing a hierarchy, which introduces shortcuts, the shortest path that was constructed does not really exist in the original graph.

The secondary part that complements a normal shortest-path search needs to convert the constructed shortest path to a path that actually exists. One possible way to achieve this is by adding a property to a shortcut when it is introduced. This property could be the node that the shortcut bypasses. Because one shortcut edge contracts a single vertex, we can descend our way down the contraction hierarchy using this property. When descending, we recursively replace a shortcut edge e that connects node a and node p by the node c it replaced. We then follow the shortest path from node a to c and from c to p. The actual route is obtained when all the shortcut-edges are eliminated.

Let us use Figure 2.6 as an example to calculate the shortest path from the red node to the blue node. As stated before, the height of the nodes is a graphic representation of their level in the

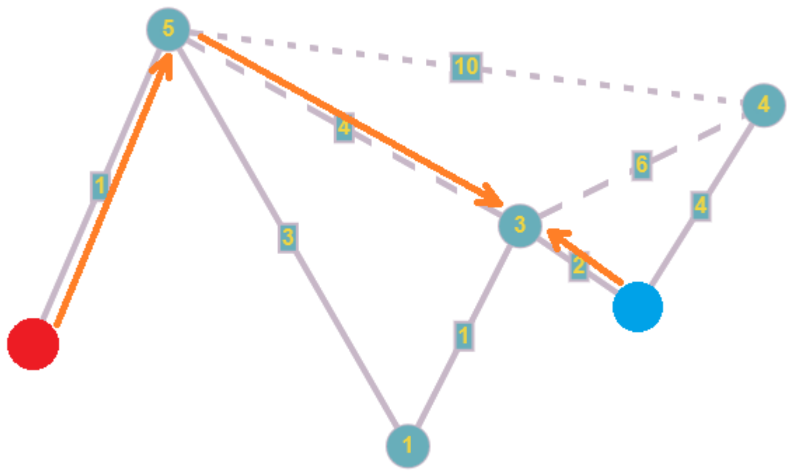


Figure 2.7: Shortest path in a contracted graph

hierarchy. When both nodes start climbing in the hierarchy, they will meet at some point at a certain node. Figure 2.7 shows what the shortest path will be, before the second part of the querying phase is executed.

The next phase is to remove any shortcuts in the shortest path. Figure 2.8 shows what the final shortest path will look like. The shortcut $(3, 5)$ is changed to the path $(3, 1, 5)$, containing the node 1 that the shortcut contracted.

Some of the advantages of contraction hierarchies are:

1. Long-distance query times are improved [16].
2. The preprocessed information can be reused.

The downsides:

1. Changes in the graph structure requires the preprocessing phase to be rerun.
2. Performance highly depends on the node order and node selection.
3. The most processing power is spent on searching witness paths and on ordering the nodes.
4. A contraction is made specifically for a small amount of transportation methods.

We chose contraction hierarchies as a base to try and speed up long distance routing on a distributed data set on a web-scale level.

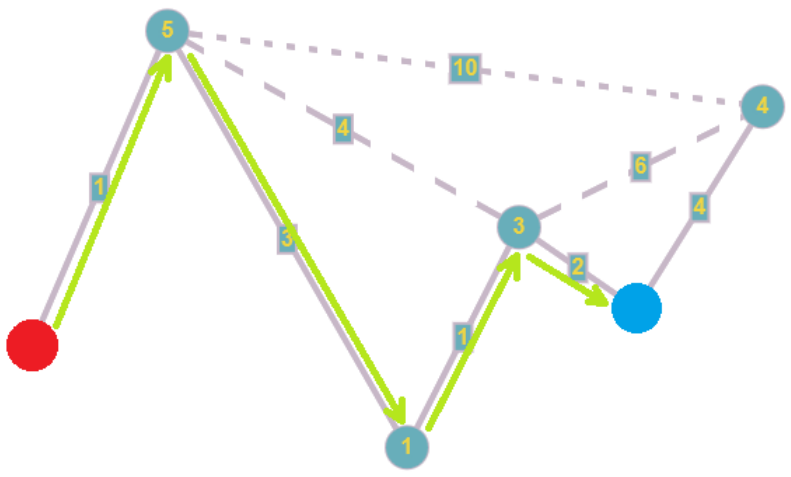


Figure 2.8: Resulting shortest path in a contracted graph

Customizable contraction hierarchies

Customizable Contraction Hierarchies (CCH) [16] is based on CH and it allows quick changes in edge-weights without the need to recalculate the complete hierarchy. To accommodate this feature, CCH is based on three stages instead of two. In between the preprocessing phase and the querying phase, a customization phase is added. The Querying phase is the only phase that is not altered and is the same as in the CH querying phase. In CH, the preprocessing phase used the edge-weights to search for witness paths. Changing edge-weights results in other possible witness paths and thus a completely different graph structure. The CCH algorithm accomplished this by performing the preprocessing on the unweighted and undirected version of the graph. A more detailed analysis and explanation of this technique can be found at [24].

3
Used tools and technologies

3.1 OpenStreet Map

OpenStreet Map (OSM) [25] is an open source initiative that published road networks as open data. All road network data that are used in this thesis are obtained through OSM.

OpenStreet Map is a community-driven project. The data sets are open data and are maintained by the community. According to [25], OSM is used by thousands of websites and apps. At the Open Belgium Conference on March 4th 2019, Joost Schouppe presented a talk on how OpenStreet Map implements data set updates faster compared to other platform that provide road infrastructure information. According to Joost Schouppe [26] OSM implements updates faster than any other provider by collaborating with local governments, using areal imagery and GPS devices accompanied with user input.

OSM publishes their data on their website [25] and offers multiple sources and APIs to access that data. Apart from using an API or accessing the online interface, it is possible to download the date of a certain area as a XML-file. A node and a way (connection of two or mode nodes) have the following format respectively respectively:

Listing 3.1: OSM Source data

```
<node id="..." visible="true" version="1" changeset="..." timestamp="..."
user="..." uid="..." lat="..." lon="..."/>

<way id="..." visible="true" version="..." changeset="..." timestamp="..."
user="..." uid="...">
  <nd ref="node id"/>
  <nd ref="node id"/>
  <tag k="bicycle" v="yes"/>
  <tag k="highway" v="track"/>
  <tag k="name" v="..."/>
  <tag k="surface" v="paved"/>
  <tag k="tracktype" v="grade1"/>
  <tag k="vehicle" v="no"/>
 </way>
```

These contain all known data about that piece of road network. This is the raw data before it is parsed through Routable tiles.

3.2 Routable tiles

As stated before, techniques to speed up long distance routing problems on a web-scale level are discussed in this thesis. The Routable Tiles application [27] and its specification was created by the OpenPlanner Team, consisting of Ben Abelshausen, Pieter Colpaert and Julian Rojas. The application, developed in C#, exposes a URL with documentation in the RDF format. At the time of writing, the URL is defined as follows:

Listing 3.2: Routable tiles exposed URL

```
https://tiles.openplanner.team/14/{x}/{y}
```

The x and y values in Listing 3.2 define the tile identification in the latitudinal and longitudinal directions. Until now, only a single *zoom level* is defined, level 14. This indicates the size of the tile and its details. Lower level tiles do not have as much details as higher level tiles e.g. level 14 is more detailed than level 10.

Routable Tiles uses OSM as the information source. The data from OSM is transformed from the format from Listing 3.1 to the format described in Listing 3.3.

When fetching a resource from the server, a JSON-LD-file is returned. JSON-LD is a JSON format, tailored for Linked Data. The file is constructed using URIs and two default attributes: id and context. The following listing is an example taken from the Routable Tiles specification website[27].

Listing 3.3: JSON-LD example for a routable tile

```
{
    @context={
                "tiles": "https://w3id.org/tree/terms#"
             },
    @id="https://tiles.openplanner.team/14/8363/5483",
    "tiles:zoom": 14
    "tiles:longitudeTile": 8363,
    "tiles:latitudeTile": 5483,
    ...
}
```

According to [28], the context defines multiple terms that are mapped to IRIs, Internationalized Resource Identifiers. The @*id* defines the resource itself and its location.

Routable tiles also defines ways, nodes and relations to expose the tiled version of OSM. These nodes and ways are used to create a graph representation of the data provided by OSM. Nodes consist of an id, longitude and latitude. Optional properties like *barrier*, *foot* or *bicycle* can be present to define additional restrictions. Ways are comprised of ids, data type, road type, and if the road is a one-way road. Ways can also contain multiple nodes.

It is important to note that a single tile not only contains the nodes inside the tile, but also the nodes outside the tile that have a connection to a node inside the tile.

Routable tiles uses OSM as an underlying data source. It tries to stay as close to OSM as possible. Tags and values are not validated[27].

Routable Tiles is used as a basis for this thesis and its software implementation. Routable Tiles is still in a early implementation and definition phase. Our goal in writing this thesis is to help shape its future. The contractions generated by the software implementation of this thesis, will be used for the less detailed zoom levels.

4 Solution architecture

The hierarchies can be seen as a representation of how the different tiles are interconnected. The software implementation which accompanies this thesis, is split into two parts: the client and server application. The server merges tiles and calculates the contractions, while the client requests these tiles and calculates a shortest route. Calculations of the contractions must be done beforehand because the time it takes to calculate such contraction can range from a couple of seconds to multiple hours.

Figure 4.1 shows the basic setup of this software implementation. A preprocessing server gets input from the routable tiles API and calculates the different tiles on different zoom levels. These tiles are saved on a storage space, in this case a subdirectory of the web server. The files are stored using their x and y IDs. These files are accessed by the web server, here in another directory on the same machine, and exposes an API. This API lets a client request tiles and use them in a route planner.

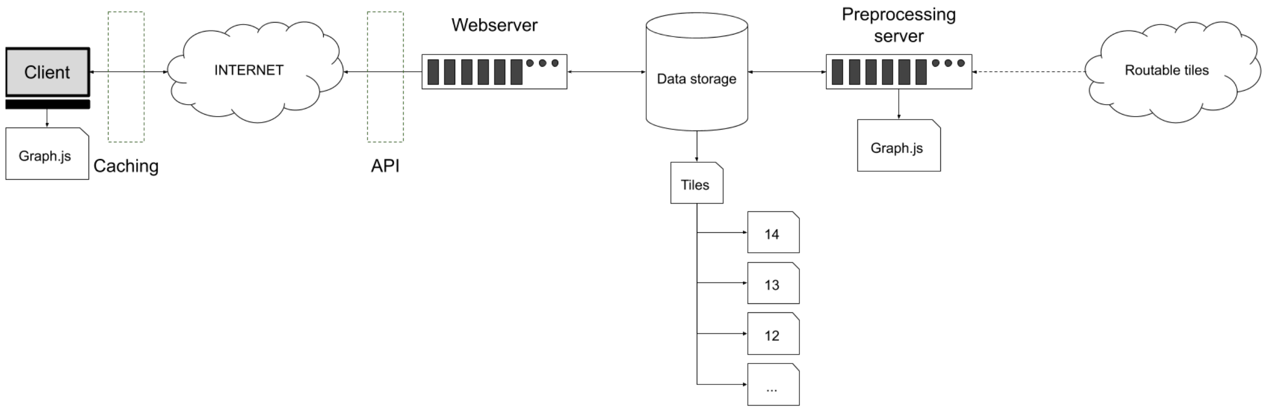


Figure 4.1: Infrastructure of the software implementation

4.1 Graph library

4.1.1 Technology

The software implementation uses its own implementation of a graph library. Although many graph libraries are available on the internet, they are not always extendable. By implementing its own library, extensions can be added more easily. The library is written in such a way that it allows extensions without major changes. One very important requirement of the implementation is that the calculations and route planning must be done on the client side. The research group for this thesis preferred a JavaScript implementation. Because it is a scripting language and not a object-oriented programming language, it is not possible to create classes and methods. As a result, the initial code is written in ECMAScript 6 (ES6), the specification on which JavaScript is based on. ES6 introduces the ability to create classes, constructors, member functions and properties as well as inheritance. Because, nowadays, most browsers do not support ES6-specific functions, the initial code is translated with Babel to ES5, which JavaScript conforms to. By doing so, most browsers can execute the script without any problems.

The graph library implements three classes: Graph, Node and Edge.

4.1.2 Class: Node

Properties like *id, x coordinate, y coordinate and edges* are the principal properties to define a node. Storing the edges in the node is key to ensure that all algorithms are able to look up the related information at any time with a reasonable efficiency. The edges themselves are not stored in the nodes, instead the ids are stored. The most important data is present, like the id, weight and destination of the edge. Additional information can be consulted in the collection of edges of the graph itself. The node keeps track of the connected edges in two directions, forward and reverse. The forward edges are outgoing edges, while the reverse edges are incoming edges. The incoming edges are used by the bidirectional Dijkstra algorithm to determine the reverse path, initiated by the destination. To speed up certain operations, additional properties are defined:

1. neighbors
2. EdgeDifference
3. TimesUpdated and UpdateVersion
4. isContractable

The contraction algorithm iterates all the *neighbors* of a node. Without storing the neighbors of a node, the contraction algorithm would need to iterate over all incoming and outgoing edges in the edge collection of the node to find all the neighbors of that node. This is not obligated, as the iteration can be done using the incoming and outgoing edges. This is simply a trade-off between memory and time, requiring more space but saves iterations.

As stated in Section 2.3.3, the contraction algorithm contracts all the nodes in a specific order. As proposed in [16], the edge difference is used as an index. The edge difference is calculated for all nodes during the preprocessing phase. The priority queue sorts all elements that are pushed to the queue, depending on this value. When a shortcut is introduced, the edge difference of the neighbors change, manipulating the topology of the graph. This will in turn change the order of the elements in the priority queue as the values of the concerned neighbors change.

The TimesUpdated is a property defined for the contraction process. As mentioned in Section 2.3.3, every time a node is contracted, the number of incident edges of all its neighbors change. Searching the complete priority queue for all the affected neighbors required significant computing power e.g. binary search is $O(log(N))$[29], with n the number of items in the collection. New nodes are created, with the same properties as the neighbors, except for the properties of the edges, which could be changed by the contraction algorithm. By pushing these nodes to the queue, there is no need to search the queue and update the existing neighbors. This also causes the priority queue to contain duplicates of a single node with slightly different values. By keeping track of how many times the node was updated or how many duplicates are contained in the queue, we can discard any nodes that are not up-to-date when they are popped. Every time a neighbor is updated, its TimesUpdated on the original node collection is incremented and the new element that is pushed to the queue contains this same value. All elements in the queue contain this property.

The property isContractable is introduced to ensure that, during the contraction of a graph, we do not contract certain edges. The edges that may not be contracted get a *True* value for this property. This is primarily used to identify the edge nodes of a tile, outside the bounds of that tile. Further information and motivation for this choice can be found in Section 4.3.2. The primary reason is that, by retaining the nodes outside the bounding box of the tile will insure that each tile has an overlap with its neighboring tiles.

4.1.3 Class: Edge

An edge object contains four basic properties: id, from, to and weight. Each edge contains the source and destination node that it connects. The weight denotes the cost to travel across the edge from the source to the destination. Additionally, a “contracts” property is introduced. This property contains the id of the node it contracts. This should improve the performance of the algorithm when it tries to expand the calculated shortest-path to an actual path that exists in the original graph. The expansion of the shortcuts is not a part of this thesis and any impact on the efficiency is not researched.

The weight of an edge currently consists of the haversine distance between the two nodes that define the edge. The most logical next step is to include the speed limit in all edges and replace the haversine distance with the actual time it takes to travel across this edge.

4.1.4 Class: Graph

As for the encompassing class, it contains a collection of nodes and edges and a boolean value indicating whether the graph is undirected or directed. Because an undirected graph is a directed graph with edges in both directions with the same cost, all methods are implemented using a directed graph. When adding an edge in an undirected graph, the class adds two edges: one in the forward direction and one in the backward direction.

Functions

The graph library offers the basic functions of adding and removing nodes and edges. Additionally, an unidirectional and bidirectional version of Dijkstra’s algorithm as well as a function to calculate the edge difference in a directed and an undirected graph is implemented.

X range	Y range	Nodes before	Nodes after	Reduction ratio
8400-8430	5450-5480	468053	128382	73,855225 %
8000-8030	6000-6030	411906	35014	91,49951688 %

Table 4.1: First stage simplification ratio

4.2 First stage simplification

The main problem with executing Dijkstra's algorithm on a graph is the size. The bigger the graph, the bigger the distance between the start and end nodes resulting in more intermediary nodes. More intermediary nodes means adding more nodes to the priority queue and sorting the elements inside the queue takes the most computing power of Dijkstra's algorithm. When using a Fibonacci heap for the queue implementation, removing a node takes $O(log(n))$ time, with n the amount of elements in the queue[30]. When all n items are popped from the queue, the efficiency of $O(nlog(n))$ is determined. To cope with this problem, we introduce a first simplification. If planning a route from a point A to a point B in, for example, Ghent, the most important steps in following the route are turns. A route is defined by the actions which changes our direction. Lots of GPS systems and online route calculators offer a summary of the planned trip. This summary consists almost only out of turns, roundabouts and changes in direction. As a result, the nodes that connect only 2 points are superfluous. By removing or bypassing these nodes, the graph shrinks to a more simple form. This technique was tested on certain tiles of the city of Ghent and a rural area of Spain. These were chosen at random and yield the results displayed in Table 4.1.

When eliminating nodes from a graph, we remove data from our data set. The result of bypassing or removing the intermediary nodes is achievable through creating a contraction hierarchy of that data set. The downside of using a contraction hierarchy is that many more new edges are introduced. See Figure 4.2 for a comparison between the result of the proposed simplification and calculation of a CH. The top part shows the initial graph before contraction and simplification. The bottom left graph has been contracted in the following order: 1, 3, 4, 2. The bottom right picture displays the proposed first stage simplification. It is clear that the first stage simplification introduces fewer new edges and results in a simpler graph, compared to CH. When simplifying the graph, the weights of the bypassed edges are preserved.

Simplifying the graph presents a trade-off between improving contraction hierarchy calculation speed and querying speed. By defining such a simplification, the starting point of the end-user might be removed after simplification.

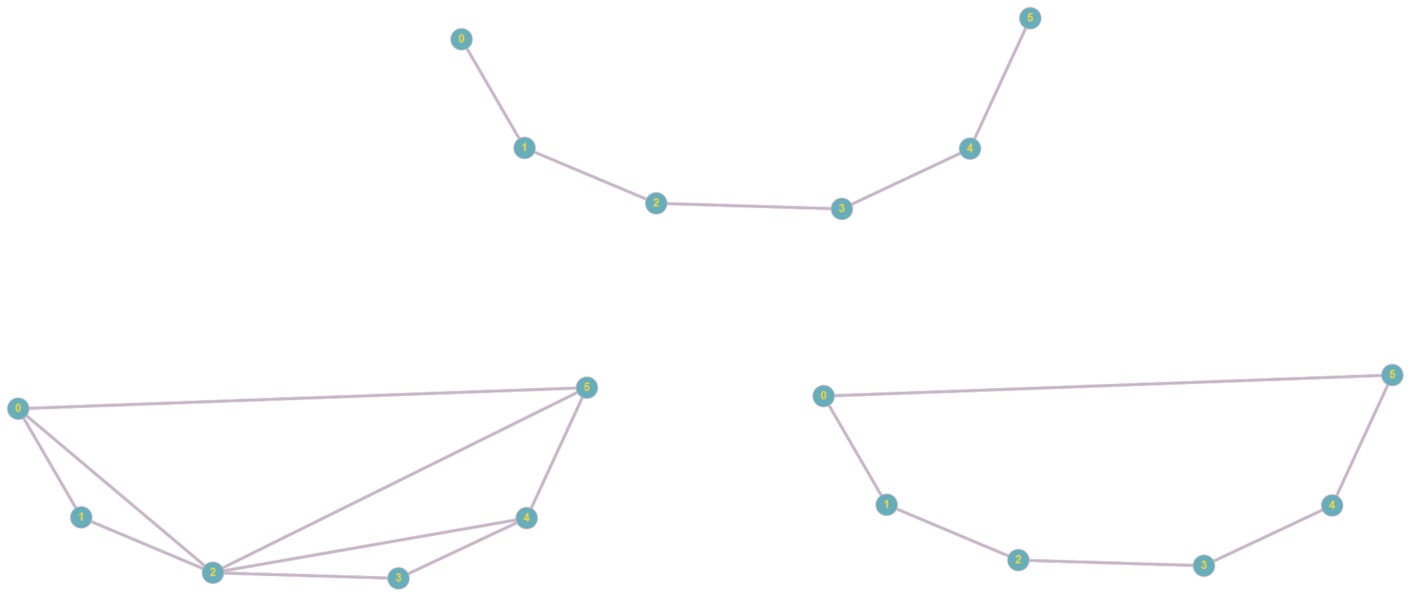


Figure 4.2: First stage simplification

4.3 Server application

The server application is responsible for creating the tiles on zoom levels 0-13. The tiles on zoom level 14 are provided by the Routable Tiles API. To create a tile for a zoom level z, four tiles from zoom level $z - 1$ are fetched. Once fetched, the contraction algorithm is executed on this tile and it is stored on the file system. The stored file contains the data of all four tiles from which it is generated as well as all the shortcuts that are generated by the algorithm. If this logic is used to calculate the tile on zoom level 0, the file would contain all data from the whole planet and all generated shortcuts. This is not a usable size and therefore the tiles need to be reduced in size before or after they are merged into a bigger tile.

A complete view of the flow of the server application can be found in Figure 4.3.

4.3.1 Receiving requests

The server application exposes a URL similar to the URL of the Routable Tiles application, with the exception that this API also accepts a z parameter, indicating the zoom level of the requested tile.

Listing 4.1: Exposed API URL server application URL

```
https://.../tiles/{z}/{x}/{y}
```

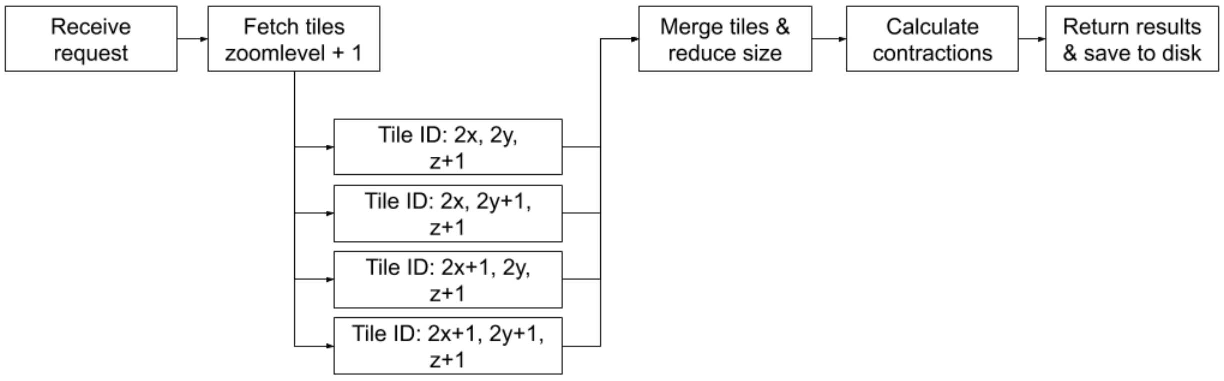


Figure 4.3: Server application flow

Once the request is received, it fetches the requested tile and returns it to the client. However, if the tile has not been generated yet, it has to be generated at run-time. This is not recommended in a production environment because calculating these tiles take a significant amount of time.

4.3.2 Merging and reducing tile sizes

When a tile needs to be generated, the underlying tiles must be fetched. These tiles are fetched and parsed into a graph. These four graphs are then all put together into a single graph. As stated before, a tile contains the nodes inside the tile and the ones outside the tile that are connected to a node inside the tile. This way, when we merge the four underlying tiles, an overlap is present and ensures that the separate tiles are interconnected with one another. If the newly generated tile is not reduced in size, it will only grow and would not deliver any advantages in terms of bandwidth when downloading this tile.

Reducing a tile in size is a critical step in generating a new tile. If too much data is removed, the tile could end up with too little information, causing the route planner to be unable to find a shortest path. However, if too much data is retained, the sizes of the files would be too large, causing the client to download a large amount of data. The following three steps are proposed to determining which data to keep and which to remove from the set:

1. Outside nodes and their neighbors

2. Shortest paths between pairs

3. Nodes with a shortcut edge

Outside nodes and neighbors

To ensure that the underlying tiles have an overlap, all the nodes outside the bounding box of the underlying tile and the neighbors inside the tile are preserved and cannot be removed. If this is done in all four of these tiles, an overlap is guaranteed. These overlaps are essential in preserving the correct data for the following step, which is calculating shortest paths between pairs.

Shortest paths between pairs

If the source node or destination node is not inside the data set of a certain tile, but the tile is fetched by the client, it means that that tile is only used to navigate through it to reach another tile. The data needed to travel across a certain tile must be retained, and preferably, be the shortest of all possible paths. Therefore, the shortest path between all nodes lying outside of the tile is calculated. This returns all nodes that are needed to route across a certain tile.

Nodes with a shortcut edge

Finally, all nodes that have a shortcut connected to them cannot be removed from the data set, otherwise we wasted precious processing power when calculating the shortcuts. Therefore, all nodes which have an edge that is a shortcut, introduced by the contraction algorithm, are retained.

Shrinking tile

When the previously mentioned three steps are executed, a set of nodes is obtained. All nodes in the tile, that are not included in this set, are removed. After removing the nodes, edges without a source an/or destination remain. Any edge without a source or destination only adds data to the graph without any contribution whatsoever. Because of this, all edges connected to these nodes are removed as well.

The reason tiles are being shrunk before contracting them, is to reduce the amount of nodes that can be contracted. The less nodes, the smaller the priority queue and the faster the algorithm finishes. It also has an impact on possible witness paths. Removing information, possibly removes witness paths with a shorter length, but it also removes the chance that the witness path is calculated using non-shortcut edges, even if there is a shortcut that bypasses this witness path. This makes the calculation of a witness path longer, because more nodes need to be discovered.

4.3.3 Contraction

This implementation uses the contraction algorithm discussed in Section 2.3.3 with multiple changes. The main differences are the following:

1. When a witness path is found, a shortcut is introduced with the weight of the witness path.
2. Only the nodes with a negative edge difference are contracted
3. The contracted level is not used in the implementation

In contrast to the algorithm described in Section 2.3.3 which was proposed in [16], not all nodes are contracted, but only the nodes with a negative edge difference. Moreover, when contracting a node and a witness path is found, a shortcut is introduced anyway with the weight of the shortest path. This makes the graph *shrink* in the long run. This is because the contracted node will most likely be removed by the step, defined in Subsection 4.3.2. The contracted level is not used because not all nodes are contracted. Nodes that are not contracted would not receive a contraction level property. The most important type of the contracted level property would be to define in which hierarchy they were introduced: the zoom level.

Sorting implementation

The sorting of the nodes is based on the edge difference, which is explained in Section 2.3.3. The edge difference is obtained through determining the value for each node before the preprocessing phase starts. When calculating the edge differences, we can observe values ranging from negative integers to large positive integers. This integer defines how much edges are introduced to the graph when contracting it.

Contraction operation

Contracting a tile is similar to the techniques described in Section 2.3.3 with the exceptions described in Subsection 4.3.3. Each pair of neighbors is checked for witness paths and shortcuts are introduced. This is done for all nodes with a negative edge difference.

After contracting the graph, it is transformed to a JSON-LD-format and stored as plain JSON to the storage location. Table 4.2 shows the file sizes of the tiles when they are exported. Note that these are averages and the values are highly dependent of the area they encompass.

Zoom level	Amount	Size (kb)	Compressed size (kb)
14	23045	75.59	10.69
13	320	296.19	25.6
12	80	1190.95	89.25
11	17	2540.47	193.41
10	4	8022	604
9	1	16428	1260

Table 4.2: Average tile sizes

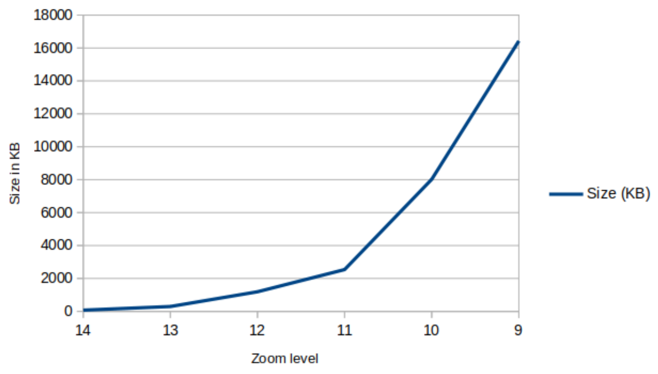


Figure 4.4: File sizes for zoom level 14 through 9

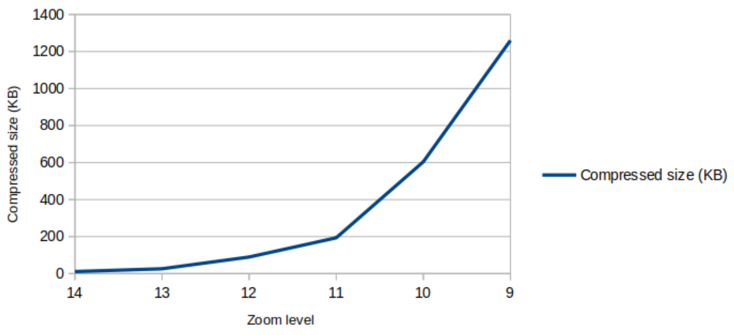


Figure 4.5: Compressed file sizes for zoom level 14 through 9

An important aspect of these tiles is that the total size, shrinks in a way that downloading tiles on multiple levels uses less bandwidth than downloading all tiles on the most specific level. Figure 4.6 displays that, when trying to calculate a shortest route between the two green tiles, the tiles on a lower zoom level will be downloaded and the complete graph will be reconstructed. Additionally, the tiles on zoom level 14 that encompass the actual shortest route without shortcuts, need to be downloaded as well. If this total is more than the total size of downloading all tiles on a zoom level 14 of the bounding box, there is no use in calculating the contractions on different zoom levels. Table 4.3 displays the averages of the total bandwidth

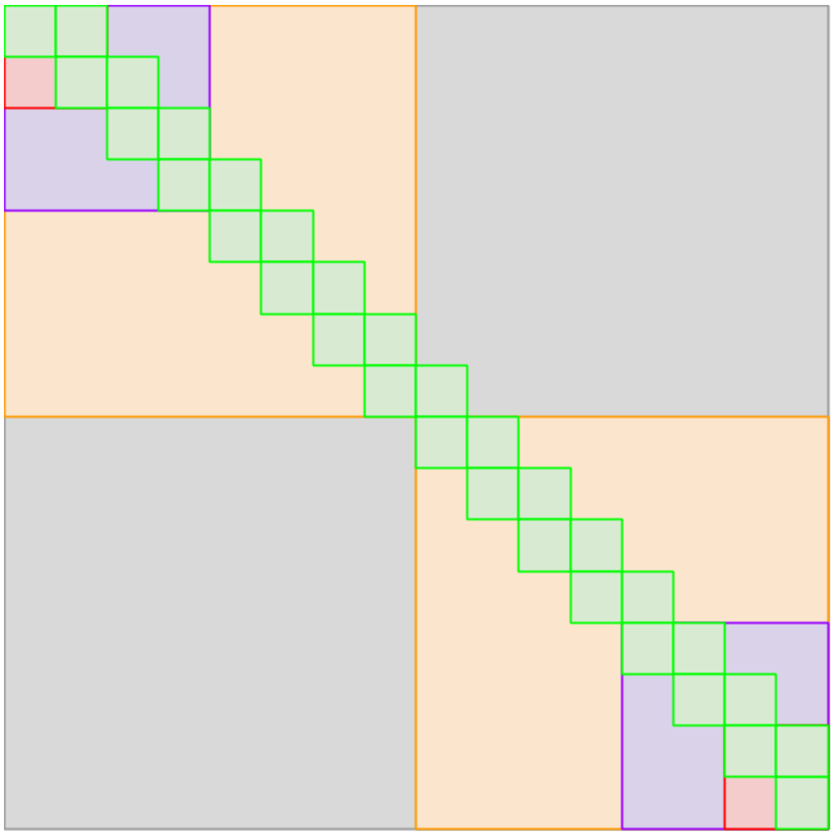


Figure 4.6: Tile hierarchy for route on zoom level 9

needed to calculate a shortest path between two green tiles which have the maximum distance in between them on a tile of zoom level 9.

Note that in Table 4.3, the amount of downloaded the amount of tiles on zoom level 14 equals 64. Only 2 are needed to calculate the shortest path over the contractions, but to return a viable path to the user, the actual tiles with actual roads, that do not contain any shortcuts, are needed.

The total bandwidth shown in Table 4.3, is consistent for all routes that require one zoom level 9 tile. It will always, on average, download the total amount displayed in that table. Short distance-routing on tiles with a start and end in different zoom level 8 tiles, will also download that zoom level 9, even if they are situated diagonally. The difference between this case and the case in Table 4.3 are the amount of tiles that are downloaded on zoom level 14. Table 4.4 displays the total download size if the tiles are positioned in a worst case scenario. The worst case scenario is also visualized in Figure 4.7.

Zoom level	Tile count	Average size per tile (KB)	Total size (KB)
14	64	75.59	4837.76
13	2	296.19	592.38
12	2	1190.95	2381.9
11	2	2540.47	5080.94
10	2	8022	16044
9	1	16428	16428
		Total	**45364.98**

Table 4.3: Total average size for route on zoom level 9

Zoom level	Tile count	Average size per tile (KB)	Total size (KB)
14	3	75.59	226.77
13	2	296.19	592.38
12	2	1190.95	2381.9
11	2	2540.47	5080.94
10	2	8022	16044
9	1	16428	16428
		Total	**40753.98**

Table 4.4: Total average size for worst case route on zoom level 9

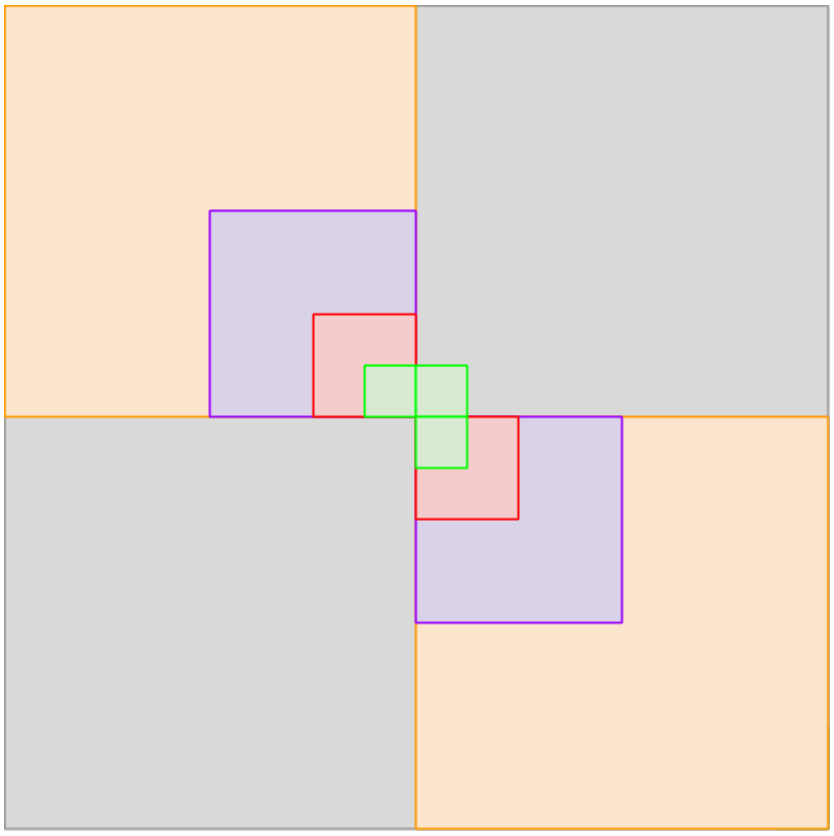
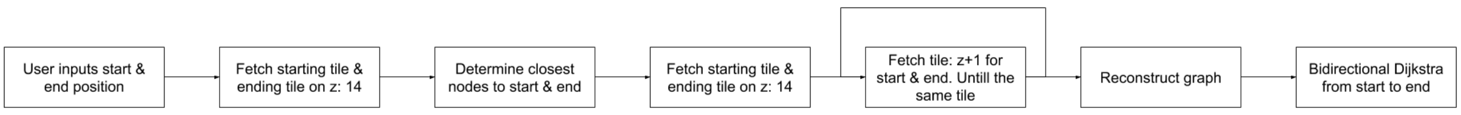


Figure 4.7: Tile hierarchy for worst-case route on zoom level 9

4.4 Client application

The client application fetches tile in two stages. First, it downloads one tile for the start and one for the finish. When the user chooses a spot on the map, it is not certain that the chosen point is actually part of the road network. Therefore, the node closest to the chosen point is used as the start or finish. The flow of the client application can be found in the figure below.


User inputs start & end position
Fetch starting tile & ending tile on z: 14
Determine closest nodes to start & end
Fetch starting tile & ending tile on z: 14
Fetch tile: z+1 for start & end. Untill the same tile
Reconstruct graph
Bidirectional Dijkstra from start to end
Figure 4.8: Client application flow

4.4.1 Fetching tiles

As a first step, the application determines which tiles to download. Because each tile has a mathematical relation with the tiles on zoom level $z+1$ and zoom level $z-1$. Use $x' = \frac{x}{2}$ and $y' = \frac{y}{2}$ to get the x' and y' id for the tile on zoom level $z' = z-1$. Use $x'_1 = 2x$, $x'_2 = 2x+1$, $y'_1 = 2y$, $y'_2 = 2y+1$ to determine the four tiles on zoom level $z' = z+1$. The application will list all IDs for x and y for all zoom levels $z' = z-1$ and will stop until both start and finish reach the same tile. Figure 4.9 displays this in a graphic manner. Let green be the start and finish tiles on zoom level $z = 14$, red $z = 13$, purple $z = 12$ and orange $z = 11$.

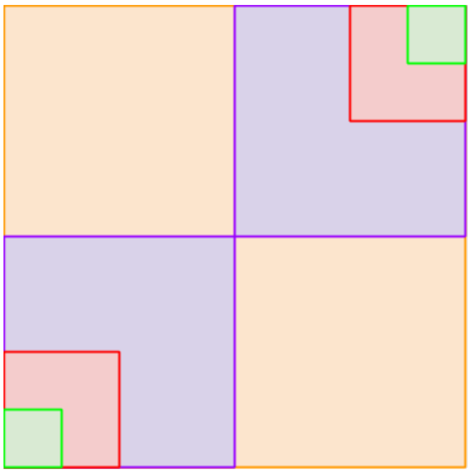


Figure 4.9: Tile hierarchy example

4.4.2 Reconstruction and shortest path

Once all tiles determined and fetched, the graph can be reconstructed. For this, all tiles are processed from a JSON-LD format to the same graph structure using the Graph library. Once the graph is reconstructed, all that remains is to calculate the shortest distance between the start and end node using a bidirectional Dijkstra algorithm.

To calculate the shortest path between start and end, two different methods could be utilized. The first being the method used in this thesis.

The first method is the reconstruction of the complete graph by gathering all tiles in the hierarchy. This makes the implementation of the Dijkstra algorithm straightforward but gives little to no room for improving the amount of downloaded data.

The second method is to initiate a tile-to-tile routing algorithm from start and end. This introduces three different stages in a regular Dijkstra algorithm and two stages in a bidirectional Dijkstra algorithm. With the regular implementation of the Dijkstra algorithm, the third stage is the first stage but in reverse and the bidirectional implementation does not need the reverse direction. The first stage is a Single Source Multiple Destinations (SSMD) step. The second stage is defined by a one or more Multiple Source Multiple Destinations (MSMD) steps. The third stage is a Multiple Source Single Destinations (MSSD) step. The algorithm starts with a SSMD at the start and and at the end, depending if it is a bidirectional Dijkstra or not. For all tiles in between the source tile and destination tile, the MSMD step is repeated. A bidirectional implementation stops when both parts have reached the same node, creating a shortest path. The regular implementation does the MSSD step in the last tile to each the destination node. This process makes the tile-to-tile routing algorithm more complex than the bidirectional Dijkstra on the reconstructed graph.

Both methods offer their own advantages and disadvantages. The first method allows an easier implementation but because all data must be downloaded before running the algorithm, it offers, without heuristics, no possible improvements on the amount of downloaded data. The tile-to-tile routing has a much more complex implementation but offers much more improvements in the amount of downloaded data. Only the data needed for the algorithm is fetched, making it more bandwidth efficient.

This thesis opted for the first method, which offers an easier implementation than the tile-to-tile algorithm.

Reconstructing the graph takes between 500 and 4000 ms, depending on the area. The test route from Sint-Niklaas to Oudenaarde takes 3500 ms on average.

Shortest path speed up

If the shortcuts that were calculated using the contraction algorithm are not prioritized over regular nodes, there would be no improvements. The Dijkstra algorithm visits a node n with the smallest total distance from the start node s to that node n. The shortcuts have a weight equal to the sum of the individual edges that the shortcut bypasses. This means that the individual edges will be discovered first and only then, depending on the implementation of the priority queue, the shortcut or the last individual edge will be used to discover a certain node n.

To increase the speed of the Dijkstra algorithm, shortcuts need to be prioritized above regular edges to improve the overall speed. Therefore, edges with a lower zoom level have priority over edges with a higher zoom level or no zoom level at all. If all shortcuts are prioritized above certain regular edges and the shortcuts tend to go in a direction away from the target, the resulting route will not be a shortest path. To counter this, the priorities are changed slightly. If an edge e, with a higher zoom level z than shortcut s, has a weight that is less than half of the shortcut, the shortcut s is not prioritized over the edge e.

The priorities can easily be extended by simply adjusting the priority function for the route planner. It is possible to add certain restrictions to the planner, like height differences, wheelchair accessibility and so on.

5 Conclusions

In this conclusion, the results from the software implementation will be compared to the Routable Tiles application as well as being tested against the hypotheses as defined in Section 1.1.2.

5.1 Execution times

5.1.1 Contraction algorithm

Calculating the contraction hierarchies for the tiles takes an acceptable time on certain zoom levels. Table 5.1 displays the average time of each activity for the zoom levels of the area surrounding Bergen and Table 5.2 shows the averages of the area surrounding Brussels. It is clear that the average for the processing and merging are slightly bigger for Brussels. However, there is a huge difference in the time it takes to simplify the graph. Brussels gives an average of 2390 seconds, which equals to 39.8 minutes for zoom level 11. The calculations of the tile on zoom level 9, which encompasses an area from Sint-Niklaas to Oudenaarde, takes roughly 7 hours. The reason it takes so long is because the shortest path between each pair of nodes outside the bounding box is calculated. The amount of times the shortest path between two nodes is calculated increases drastically with an increase in the amount of nodes. This operation is the bottleneck on the server application and any and all improvements made on this stage are

Zoom level	Sample size	Fetching tiles (ms)	Process subtiles (ms)	Merge subtiles (ms)	Simplify graph (ms)	Edge difference (ms)	Contract graph (ms)	Save to disk (ms)
13	23	0	90	98	72	19	189	14
12	6	1268	112	111	0	83	645	52
11	1	8887	555	408	577786	120	2499	92

Table 5.1: Average server execution times for area surrounding Bergen (Mons)

Zoom level	Sample size	Fetching tiles (ms)	Process subtiles (ms)	Merge subtiles (ms)	Simplify graph (ms)	Edge difference (ms)	Contract graph (ms)	Save to disk (ms)
13	64	0	130	148	88	29	328	21
12	16	2941	209	189	0	145	1779	81
11	3	16831	754	767	2390587	360	8400	212

Table 5.2: Average server execution times for area surrounding Brussels

good. On zoom level 12 and 13, the total time is acceptable and does not pose a threat to the overall performance, because the zoom levels 11 through 0 outweigh these execution times.

No comparison can be made because there is no real reference for these results. The calculation highly depends on the CPU of the machine running the code. These results are obtained from running the calculations on an Intel Core i5-4210U @ 1.70 GHz, which is not a heavy-duty machine. The duration to calculate tiles on zoom level 13 and 12 are reasonable and could be calculated during run-time.

5.1.2 Reconstruction and shortest path

The total duration for the client is the total time it takes to fetch the tiles, reconstruct the graph and calculate the shortest path in that graph. Fetching the tiles and reconstructing the graph takes the longest amount of time, being between 3000 and 3500 ms for an intercity route. Both of these figures were obtained by calculating a shortest path between Sint-Niklaas and Oudenaarde. The execution time of the routing algorithm, around 550 ms to make an intercity route, is a little above the standard of 500 ms. An end user perceives a delay of 500 ms as near-instant. The solution, which gives an average 550 ms, does not fulfill this requirement.

5.2 Bandwidth

To measure the amount of used bandwidth, without taking caching into account, is performed by looking at the JSON-file sizes. As mentioned before, the average tile sizes can be found in Table 4.2. These file sizes are acceptable in a linked open data context. Every improvement is a win for the client and the overall speed.

To even further reduce the file sizes, the files are compressed using *gzip*. This compression results in relatively small file sizes, considering they encompass a large portion of the roads in that area. The files shrink to about $\frac{1}{10}$ to $\frac{1}{13}$ of their original size. The compressed file on zoom level 9, which contains the area of Ghent, only takes up around 1.2 MB. The uncompressed file size is 16.8 MB.

One of the hypotheses proposes the comparison between this software implementation and the Routable Tiles application. As mentioned in Section 4.3, there is a possibility for a best-case and a worst-case. Both cases will be compared to the Routable Tiles application. Figure 5.1 show the worst-case and Figure 5.2 shows the best-case comparisons to the Routable Tiles application in terms of bandwidth. The size of a tile on zoom level 8 is an estimate based on the averages of four tiles on zoom level 9 that will be merged and no data will be removed, making it a very conservative estimate.

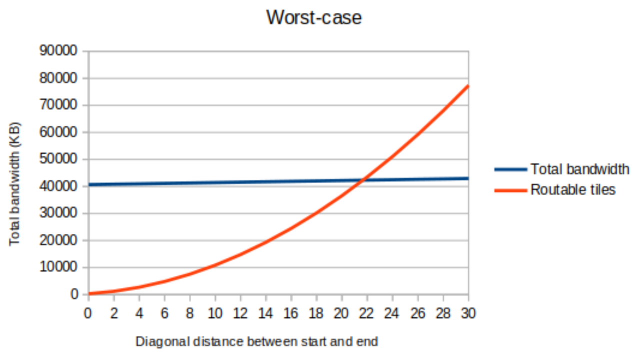


Figure 5.1: Worst-case comparison to the Routable Tiles application

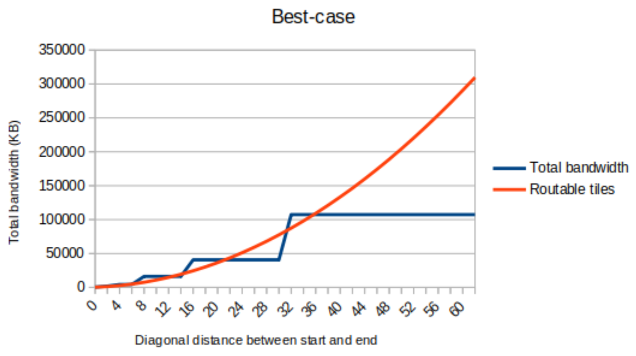


Figure 5.2: Best-case comparison to the Routable Tiles application

It is clear that, without any improvements, the Routable Tiles application is much more efficient in terms of bandwidth on short distances. However, the larger the distance between start and end, the more this software implementation will provide better results. As a conclusion, this software implementation lacks efficiency on short-distance routing but it is much more data-efficient on long-distance routing compared to the Routable Tiles application.

5.3 Caching

The Express web server used by this implementation offers an easy way to enable caching. Setting the the lifetime value in the *max-age* property of the *Cache-Control* object, will enforce caching for each request. The download times for each file drops from $\frac{filesize}{internetspeed}$ to 20 ms on average if the file was present in the cache.

6
Future work

6.1 Algorithm speed increases

It is possible to extend the contraction and the shortest path algorithm to better serve certain needs.

The implementation of the contraction algorithm shrinks tiles after fetching. To determine what information to preserve and what to remove, the shortest path is calculated between each pair of nodes that are outside the bounding box. Currently, the shortest path in the contraction algorithm does not take shortcuts into account. A way to speed up this process is to use shortcuts, speeding up the Dijkstra algorithm it uses.

To decrease the amount of downloaded data in a worst-case scenario, as shown in Table 4.4, additional tiles can be downloaded. By downloading additional tiles, the total amount of downloaded data can be reduced. When searching for a route between two tiles, all tiles in zoom levels above that tile will be downloaded until they meet in a certain tile on a certain zoom level. If the neighboring tiles are downloaded for each tile on a zoom level and the algorithm stops when both start and end want to download the same tile, the total amount of data can be reduced. However, this can have an impact on best-case or average-case long-distance routing, as unnecessary tiles are downloaded as well.

When calculating a route from a point that lies at the edge of a tile certain routes, e.g. the highway around Ghent, will be ignored because they are not contained in the hierarchy of that tile. Sometimes it is faster, not shorter, to go around a city like Brussels instead of driving through it. By fetching the surrounding tiles, it is possible to remove this restriction. Fetching those tiles does increase the overall amount of downloaded tiles and graph size, slowing down the shortest path algorithm.

6.2 Extensions

As of now, the implementation does not take turn restrictions into account. Turn restrictions are attributes in a *osm:Relation* attribute. A relation defines how ways and nodes are intertwined. There are 9 types of turn restrictions and each restriction defines what cannot be done.

- osm:no_left_turn
- osm:no_right_turn
- osm:no_straight_on
- osm:no_u_turn
- osm:only_right_turn
- osm:only_left_turn
- osm:only_straight_on
- osm:no_entry
- osm:no_exit

Implementing these restrictions can further refine the route planning and make it more correct.

The weight of edges is defined by the haversine distance between the two nodes it connects. The haversine distance is the distance between two points on a sphere, in this case the planet earth. To make highways and other roads with a higher speed limit more attractive to use in the routing algorithm, the haversine distance can be replaced by calculating the time it takes to travel over a certain edge in the graph. Calculating the time needed to travel across an edge is possible with the haversine distance and the speed limit, embedded in the routable tiles object.

When a shortest route is calculated, it still uses shortcuts. Shortcuts are roads that do not exist and possibly even pass through buildings. The next step in this route planner would be to recursively convert all shortcuts back to regular ways.

6.3 Future visions

Using Linked Open Data, this route planner and its contraction algorithms could be extended to facilitate a wide variety of applications. A route planner specifically designed to help people with a disability find the train stations and bus stops that are wheelchair accessible is a good example of a possible future use-case. It also enables developers, companies and even individuals to develop their own custom route planner based upon the tiles, which are freely usable, accessible and extendable.

Bibliography

[1] N. Noy, M. Burgess, and D. Brickley, “Google dataset search: Building a search engine for datasets in an open web ecosystem,” 2019.

[2] L. D. F. collaborators. [Online]. Available: http://linkeddatafragments.org/

[3] Y. Chen, T. Farley, and N. Ye, “Qos requirements of network applications on the internet,” *Information Knowledge Systems Management*, vol. 4, no. 1, pp. 55–76, 2004.

[4] “Open definition 2.1,” Tech. Rep. [Online]. Available: https://opendefinition.org/od/2.1/en/

[5] R. T. Fielding, “Architectural styles and the design of network-based software architectures.” University of California, 2000.

[6] T. Berners-Lee, *Linked Data - Design Issues*, W3.org, 2006. [Online]. Available: https://www.w3.org/DesignIssues/LinkedData.html

[7] P. Colpaert, “Publishing transport data for maximum reuse,” Ph.D. dissertation, Universiteit Gent, 2018. [Online]. Available: https://phd.pietercolpaert.be/

[8] DBpedia. [Online]. Available: https://wiki.dbpedia.org/

[9] E. Miller, “An introduction to the resource description framework,” *Bulletin of the American Society for Information Science and Technology*, vol. 25, no. 1, pp. 15–19, 1998.

[10] D. Evans, “The internet of things: How the next evolution of the internet is changing everything,” 2011.

[11] E. W. Dijkstra. [Online]. Available: http://www.cs.yale.edu/homes/lans/readings/routing/dijkstra-routing-1959.pdf

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms.* MIT press, 2009.

[13] D. Delling, P. Sanders, D. Schultes, and D. Wagner, “Engineering route planning algorithms,” in *Algorithmics of large and complex networks.* Springer, 2009, pp. 117–139.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[15] D. Schultes, "Route planning in road networks." in *Ausgezeichnete Informatikdissertationen*, 2008, pp. 271–280.

[16] R. Geisberger, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," p. 70, 2008.

[17] R. J. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks." *ALENEX/ANALC*, vol. 4, pp. 100–111, 2004.

[18] P. Sanders and D. Schultes, "Engineering highway hierarchies," in *European Symposium on Algorithms.* Springer, 2006, pp. 804–816.

[19] 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.

[20] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-dependent contraction hierarchies," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments.* Society for Industrial and Applied Mathematics, 2009, pp. 97–105.

[21] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter, "Distributed time-dependent contraction hierarchies," in *International Symposium on Experimental Algorithms.* Springer, 2010, pp. 83–93.

[22] J. Dibbelt, B. Strasser, and D. Wagner, "Customizable contraction hierarchies," in *International Symposium on Experimental Algorithms.* Springer, 2014, pp. 271–282.

[23] ——, "Customizable contraction hierarchies," *Journal of Experimental Algorithmics (JEA)*, vol. 21, pp. 1–5, 2016.

[24] T. Columbus, "Search space sizeincontraction hierarchies," 2012.

[25] OpenStreet Map. [Online]. Available: https://www.openstreetmap.org/about

[26] J. Schouppe, "Lecture: Being the first to map new things," Open Belgium 2019. [Online]. Available: https://osmbe.github.io/presentations/talks/2019-03-04-on-being-first-jschouppe/

[27] P. Colpaert, B. Abelshausen, and J. Rojas, *Routable Tiles Specification*, Open Planner Team, 2019. [Online]. Available: https://openplanner.team/specs/2018-11-routable-tiles.html

[28] "Json-ld v1.1, draft community group report 05 september 2018," Tech. Rep., 2018. [Online]. Available: https://json-ld.org/spec/latest/json-ld/

[29] D. E. Knuth, *The art of computer programming: sorting and searching.* Pearson Education, 1997, vol. 3.

[30] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.