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

Word of thanks

Words expressed as a Thank you to everyone involved

Contents

List of Figures **6**

1 Introduction **8**

1.1 Goal of this thesis 9

2 Basic concepts **10**

2.1 Data on a Web-scale level 10

2.2 Resource Description Framework (RDF) and Linked Open Data 11

2.3 Graph routing concepts 13

2.3.1 Dijkstra algorithm 13

2.3.2 Dijkstra algorithm improvements 15

2.3.3 Contraction Hierarchies 18

3 Used tools and technologies **24**

3.1 OpenStreet Map 24

3.2 Routable tiles 25

4 Software implementation **27**

4.1 Graph library 27

4.1.1 Technology 27

4.1.2 Class: Node 28
4.1.3 Class: Edge 29
4.1.4 Class: Graph 29
4.2 First stage simplification 29
4.3 Contraction 30
4.3.1 Sorting implementation 30
4.3.2 Contraction operation 31
4.3.3 Relation between contraction hierarchies 31
4.3.4 Results of the preprocessing phase 31
4.4 Routing planner 31
4.4.1 Shortest path part 1 31
4.4.2 Shortest path part 2 31
4.4.3 Reconstructing the path 31

5 Related work **32**

5.1 Related work 32
5.2 Future work 32

Bibliography **33**

List of Figures

2.1 An RDF triple 12

2.2 Visited nodes by the Dijkstra algorithm 14

2.3 Visited nodes by the bidirectional Dijkstra algorithm 14

2.4 Visited nodes with the A* heuristic 15

2.5 A basic graph 18

2.6 Contraction hierarchy of the basic graph 20

2.7 Shortest path in a contracted graph 22

2.8 Resulting shortest path in a contracted graph 22

4.1 First stage simplification 31

Listings

2.1 a possible CSV notation 11
2.2 Example URIs for a Hawaiian Pizza and an Oven 11
2.3 Pseudo-code for the Contraction Hierarchy algorithm 18
3.1 OSM Source data 24
3.2 Routable tiles exposed URL 25
3.3 JSON-LD example for a routable tile 26

1

Introduction

Searching for answers, communication, services and sharing knowledge are all applications of the web. The older it gets, the more and more is digitized and stored on the web. Nowadays, the ever growing World Wide Web is indispensable and rooted in all aspects of our daily lives. In this day and age, 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 for answering question or queries depends on the type of answer one expects. Answering a question in full is quite difficult because it requires data from different contexts. Take gas-stations as an example. If you are planning a road-trip and require the price of diesel at the different gas-stations you will encounter, you will need the diesel prices of all the companies that own those gas-stations. If your road-trip is international and the search engine you are using only provides prices of domestic gas-stations, you will need a different service to query the prices of the international gas-stations. Linked data tries to avoid this issue by publishing its data in a uniform and structured manner.

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 to the proposed routing question. This thesis deals with some techniques that can be used to speed up the navigation process when computing navigation queries when the data are

published on web-scale.

1.1 Goal of this thesis

The goal of this thesis is to research techniques to speed up long distance routing problems on a web-scale level. 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 of this thesis are so specific, little research on this specific topic could be found during the literature study. The techniques discussed in this thesis have already been researched. These variables include datasets published in a distributed manner that allow scaling on web level and a semantic definition of all object and entities is required in the long run. Additionally, all datasets 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. The combination of all these variables and routing problems is an area that has not been discovered in full and allows addition research to be done to improve the general performance.

2

Basic concepts

2.1 Data on a Web-scale level

According to [1], open datasets are datasets from which the data can be accessed, used, modified and shared for any purpose whatsoever by anyone. Many instances already publish certain datasets as Open Data or host programming competitions based on these datasets. The beauty of open data is its transparency and re-usability. Because it is Open data, a single dataset can be used in all kinds of applications and sectors. Take the Open Data Gent platform as an 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 garages 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 dataset. The datasets that are part of an ecosystem called Linked Open Data (LOD).

2.2 Resource Description Framework (RDF) and Linked Open Data

Linked Data is a term coined by Tim Berners-Lee, the inventor 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[2]:

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 via an example. Consider that we use "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 [3].

The first item describes that URIs must be used as names for things. If we format our data into a CSV style listing, 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"
```

Now, 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 an online database for example, the identifier can easily be reused. If the term "Pizza Hawaii" was defined as an URI by a company that produces pizza's, 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, a crowd-sourced community effort that serves Linked Data, already contains a definition for a "Pizza Hawaii" and a "Oven". The names in our example can be transformed to a more uniform way of describing this recipe. Any ambiguous interpretations are now nearly impossible.

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

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

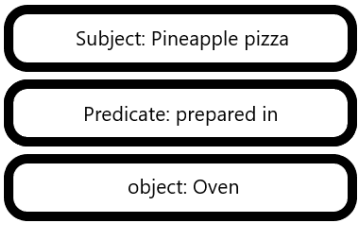

Figure 2.1: An RDF triple

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 lookup the mentioned objects. The instance that defined these items are 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 pizza. The same applies to the Oven object.

As third item, Tim Berners-Lee defined that a URI should provide useful information using the standards e.g. RDF*. RDF, the Resource Description Framework, is a framework for describing resources on the internet. RDF is meant to be machine-readable and thus written in XML. This makes it quite hard for a human to read the RDF definitions. 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 uses a triple format. This triple consists of 3 basic items[3]

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 datasets, the querying algorithm can iterate over multiple datasets. In contrast to an algorithm using Linked Data and RDF, a generic query answering service only accesses its own datasets and the datasets that it was programmed to access.

The RDF framework benefits the client applications and its programmers. 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. The difficulty to create a client

application that implements these different services and protocols increase with the number of services and protocols used.

LOD eliminates the restriction of being confined to the datasets 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 dataset. By using Uniform Resource Identifiers, it is enables the intertwining of different dataset. By using the URIs of other entities in the object properties, multiple datasets can be combined to provide even more information. Data can be added by any instance and can be accessed by any instance. When the user agents queries certain resources, it is able to access related information because of the data being intertwined and 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 dataset. All these uniform definition 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, they use 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. 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 reduces the 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 computation to answer a certain query is shifted from server to client, allowing the servers to scale more evenly and cope with above mentioned challenges.

2.3 Graph routing concepts

2.3.1 Dijkstra algorithm

The Dijkstra algorithm [4][5] is widely used for finding shortest paths 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. The Dijkstra algorithm will start at the node s by visiting it. Dijkstra considers all neighbours 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 the Dijkstra 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 are all the vertices the algorithm visits. To reach the endpoint node, the Dijkstra 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.

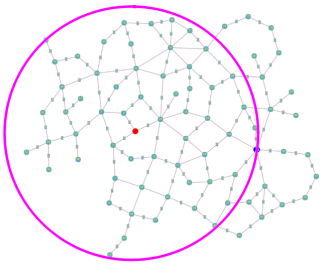

Figure 2.2: Visited nodes by the Dijkstra algorithm

Bidirectional Dijkstra algorithm

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 the Dijkstra algorithm starts

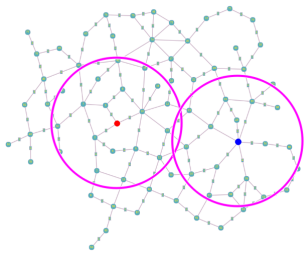

Figure 2.3: Visited nodes by the bidirectional Dijkstra algorithm

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 $A_1 \geq A_2$, since A_1 grows exponentially and A_2 grows in a linear fashion.

2.3.2 Dijkstra algorithm improvements

The following techniques can speed up the Dijkstra algorithm when searching for a shortest path. The pros and cons of some technique are discussed and if they are applicable on a web-scale published dataset.
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 trivial to this thesis' routing problem. In this thesis, Contraction Hierarchies will be discussed comprehensively as it is used as a basis for the Proof Of Concept of this thesis.

A* heuristic

The A* heuristic 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 neighbouring 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 planner, 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, will always be an underestimation as the shortest possible path is the Euclidean distance. Further analysis and proof of this heuristic will not be covered in this thesis as this is not part of the scope.

In Figure 2.4, let the red node be the start s and the blue node the end node t. The brown arrow denotes to the shortest path at a certain iteration. The heuristic will look at all neighbours 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

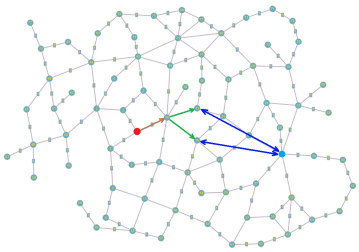

Figure 2.4: Visited nodes with the A* heuristic

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 will precompute all distances between a small amount of landmarks and all nodes v. These precomputations are used in combination with the Triangle inequality. Because this thesis deals with routing problems on a distributed system, knowledge of Landmarks outside the immediate surrounding of a node v is not trivial in this setting. Further analysis is omitted.

Reach-based routing

Reach-based routing 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) 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. This technique can be used to extend and speed-up hierarchical graph structures like Highway Hierarchies, Highway Node Routing and Contraction Hierarchies. Further analysis is necessary to map its impact and possible improvements in performance.

Highway Hierarchies (HH)

In our current road network, hierarchies can be found. Let the maximum speed be the index for the hierarchy. Inner city roads will have the lowest and highways have the highest speed limit. Highway hierarchies will distribute nodes and edges in certain levels of a hierarchy. It groups roads with a smaller importance into roads with a higher importance. All this is based on the fact that planning a long-distance trip will have the least edges and turns in the middle section. It takes a neighbourhood in consideration: highway edges are edges that are neither in the surrounding area of the start and the ending and are thus in the middle section of a trip.

The size of this surrounding area can be chosen and is fixed. Every hierarchy has a sparser representation of certain pieces of the road network. The advantages of Highway Hierarchies

1. Significantly faster than the regular Dijkstra algorithm. Some papers claim it is over 2.000 times faster.
2. The hierarchy of highway networks allows bidirectional searches.

The disadvantages are as follows

1. Preprocessing eliminates the possibility to instantly use this method. The preprocessing phase requires multiple hours for larger road networks.
2. The hierarchies require additional space
3. Changes to the base structure require the preprocessing phase to be rerun

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 (HNR)

Highway Node Routing (HNR) 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 neighbourhood size that is predetermined, but the nodes that are used in the hierarchies are predetermined. The performance of this algorithm depends on the node selection. The advantages of HNR:

1. Faster than a Dijkstra algorithm
2. Less sensitive to changes in graph structure

The disadvantages

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 this method needs research as this not done in this thesis.

Figure 2.5: A basic graph

2.3.3 Contraction Hierarchies

Contraction Hierarchies (CH) 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). In classic situations, the road network, displayed by a graph, is contracted and a contraction hierarchy is obtained. This is then queried with a bidirectional Dijkstra 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 orders 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 neighbour u of node v and with I_u > I_v
        for each neighbour w of node v with w ≠ 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 searches for a total of two neighbours of node v that we will potentially contract. Some constraints are in effect. The neighbours 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 4 is very important and will be discussed more in depth later on. It requires the path from u to w 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 influences 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 has a complete other result. According to the thesis [6], "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. [6] 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 neighbours 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. Obviously, these values change every time the graph changes its structure. Structural changes in and around a node occur when a neighbour 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. [6] proposed to use a shortcut. When a node is contracted, only its neighbours need to be updated and can thus be sped up. This is called Lazy-updates and it only updates the nodes that are affected by a change in structure. ***Vergelijking door maken van 2 CH's van dezelfde graaf maar met andere sorteervolgordes?***

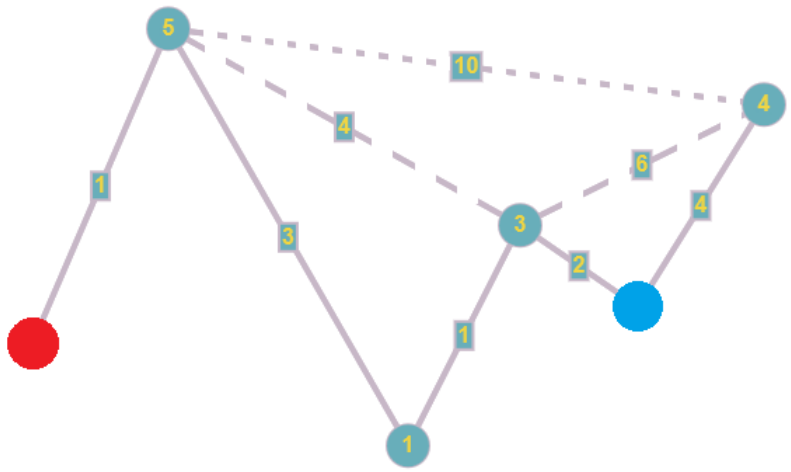

Figure 2.6: Contraction hierarchy of the basic graph

Witness paths

Before we can contract a node v with neighbours 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 [6], *witnesspaths* 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[6].

Contraction

What does contracting 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.

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 at the same time. A bidirectional Dijkstra is thus performed. However, when a node v wants to visit or discover all its neighbouring nodes, an extra restriction applies. The search is limited to the neighbours 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.

Lets 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 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.

Converting the shortest path with shortcuts to the shortest path without shortcuts is the second phase. 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:

1. Long-distance query times are improved [6]

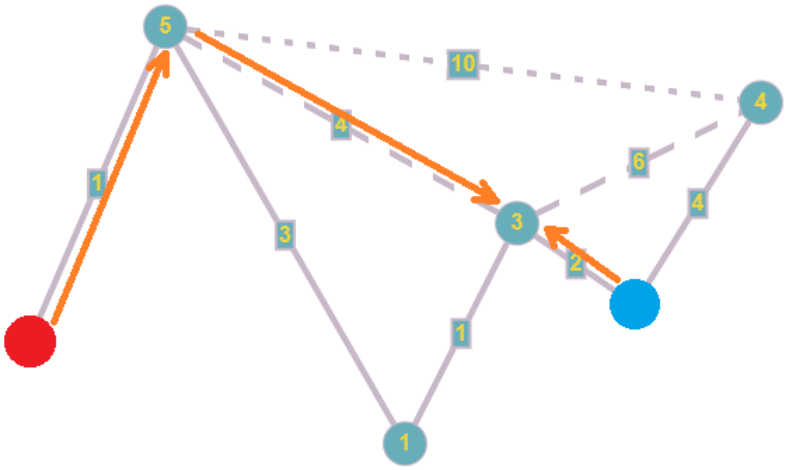

Figure 2.7: Shortest path in a contracted graph

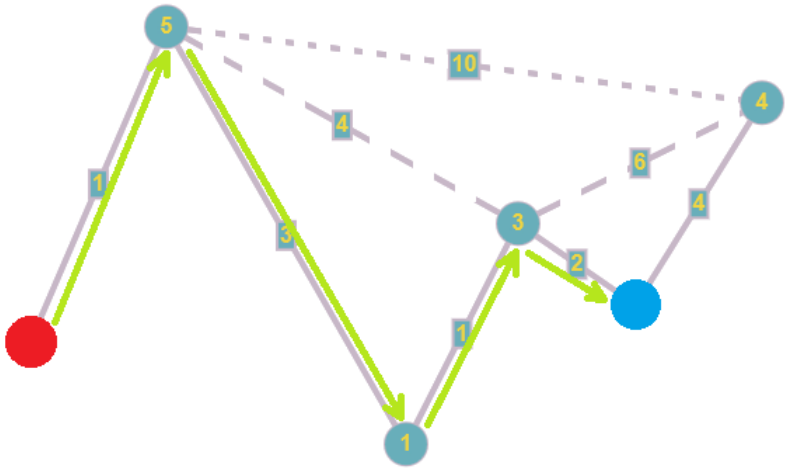

Figure 2.8: Resulting shortest path in a contracted graph

2. Preprocessed information can be reused

The downsides:

1. Changes in the graph structure requires the preprocessing phase to 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

This thesis chose Contraction Hierarchies as a base to try and speed up long distance routing on a distributed dataset on a web-scale level.

Customizable contraction hierarchies

Customizable Contraction Hierarchies (CCH) 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 [7].

3

Used tools and technologies

3.1 OpenStreet Map

OpenStreet Map (OSM) 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 datasets are open data and are maintained by the community. According to [8], OSM is used by thousand of websites and apps. At the Open Belgium Conference on March 4th 2019, Joost Schouppe presented a talk on how OpenStreet Map implements dataset updates faster compared to other platform that provide road infrastructure information. According to Joost Schouppe [9] OSM implements update 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 [8] 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. This if of the following format for a node and a way (connection of two or mode nodes) 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, this thesis researches techniques to speed up long distance routing problems on a web-scale level. The routable tiles application 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, 2019, 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 latitude and longitude directions. At the time of writing, only a single *zoom level* is defined, level 14. This indicates the size of the tile and its details. Lower level tiles won't have as much details as higher level tiles e.g. level 14 is more detailed than level 10.

Routable Tiles uses OSM as the source of their information. 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 tile specifiaction

website[10].

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 [11], 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.

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[10].

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

4

Software implementation

The hierarchies can be seen as a representation of how the different tiles are interconnected.

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 so easily extendable. By implementing its own library, extensions can be added pretty easily. The library is written in such a way that it allows the extensions without major changes. One of the basic principles of this thesis is that the calculations and route planning must be done on the client side. The research group for this thesis preferred a JavaScript implementation. Therefore, the graph library was written in JavaScript. Because JavaScript 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 and also introduces inheritance. Because, at the time of writing, most browsers do not support ES6 specific functions, the initial code is translated with Babel to ES5 to which JavaScript conforms

to. By doing this, 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, $x - coordinate$, $y - coordinate$, $edges$ are the principal properties to define a node. The edges stored in the node contain the forward and reverse edges, as a bidirectional Dijkstra requires the node to know its predecessors in a direct graph. To speed up certain operations, additional properties are defined:

1. Neighbours
2. TimesUpdated
3. ContractionLevel
4. EdgeDifference

By keeping track of the neighbours (only the ID), it is not necessary to iterate all the connections to get the collection of neighbours. This is a trade-off between space and time. It requires more space, but saves iterations.

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 neighbours change. Searching the complete priority queue for all the affected neighbours required significant computing power (binary search is $O(log(N))$, with n the number of items in the collection), the neighbours are pushed into the priority queue with the updates values. This 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, we can discard any nodes that are not up-to-date when they are popped from the queue. Every time a neighbour 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. The elements in the queue contain a $updateVersion$ property that defines the version of when it was added to the queue.

$ContractionLevel$ denotes the level of the node in the hierarchy. This value is used in the routing algorithm, as the next possible node in the shortest-path must be a node with a higher level in the hierarchy, and thus a lower value of $ContractionLevel$.

For sorting, as stated in Section 2.3.3, the $EdgeDifference$ is calculated for every node in the preprocessing phase. This value is used in the priority queue to determine which node to

contract next. Every time a node is contracted, the $EdgeDifference$ of all its neighbours is updated and are then pushed to the queue again with a incremented $updateVersion$.

4.1.3 Class: Edge

An edge object contains four basic properties: id, $from$, to and $weight$. Each edges 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 base graph.

4.1.4 Class: Graph

As for the encompassing class, it contains a collection of nodes and edges and two boolean value indicating whether the graph is undirected or directed and if it is an unweighted or weighted graph. Because an undirected graph is basically a directed graph with edges in both directions with the same cost, the implementation is actually 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 add and remove functions for nodes and edges. Additionally, a unidirectional and bidirectional Dijkstra as well as a function to calculate the edge difference in a directed and a function to calculate the edge difference in an undirected graph is implemented.

4.2 First stage simplification

The main problem with executing the Dijkstra algorithm on a graph is the size. The bigger the graph, the bigger the distance between 2 nodes and thus the bigger the amount of 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 the Dijkstra 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. This explains the efficiency of $O(nlog(n))$ of the algorithm. To cope with this problem, we introduce a first simplification. If planning a route

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

from a point A to a point B in Ghent for example, the most important steps in following the route are turns. A route is defined by the actions which change our direction. Lots of GPS systems and online route calculators offer a summary of the planned trip. This consists almost only out of turns, roundabouts and change 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 the following results were obtained:

When eliminating nodes from a graph, we remove data from our dataset. The result of bypassing or removing the intermediary nodes is achievable through creating a Contraction Hierarchy of that dataset. The downside of using a Contraction Hierarchy is that a lot more of new edges are introduced. See Figure 4.1 for a comparison between the result of the proposed simplification and calculating a CH. The top part shows the initials graph before contraction or 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 less new edges and results in a simpler graph, compared to contraction. When simplifying the graph, the weights of the bypassed edges are added together and represent the weight of the newly added edge.

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. Therefore, the initial step of querying consists of finding the closest node that has not been removed by the first simplification step.

4.3 Contraction

4.3.1 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 by determining the value for each node before the contraction preprocessing phase starts.

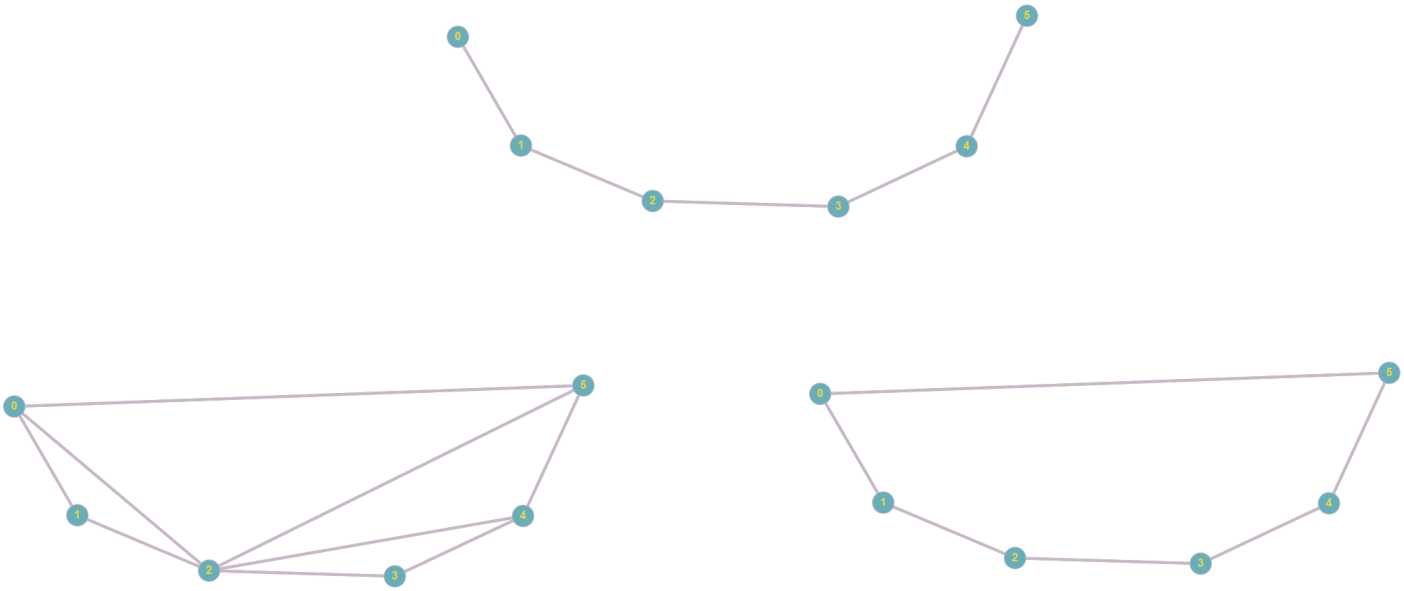

Figure 4.1: First stage simplification

4.3.2 Contraction operation

4.3.3 Relation between contraction hierarchies

4.3.4 Results of the preprocessing phase

4.4 Routing planner

4.4.1 Shortest path part 1

4.4.2 Shortest path part 2

4.4.3 Reconstructing the path

5

Related work

5.1 Related work

5.2 Future work

Bibliography

[1] "Open definition 2.1," Tech. Rep. [Online]. Available: https://opendefinition.org/od/2.1/en/

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

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

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

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

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

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

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

[9] 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/

[10] J. R. Pieter Colpaert, Ben Abelshausen, *Routable Tiles Specification*, Open Planner Team, 2019. [Online]. Available: https://openplanner.team/specs/2018-11-routable-tiles.html

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