Universiteit Leiden
The Netherlands

Opleiding Informatica

Optimizing and Profiling the Preprocessing Speed of the Contraction Hierarchies Algorithm

Valentijn Stokkermans

Supervisors:
Kristian Rietveld & Luc Edixhoven

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 8/11/2024

Abstract

Shortest paths in road networks are important because they minimize travel time, improving overall transportation efficiency. Most people use some form of navigation in their daily life, these navigation systems use algorithms like Contraction Hierarchies to determine the fastest route to the destination. The Contraction Hierarchies is a promising algorithm for finding the shortest path in a graph. It consists of two stages. First the graph is preprocessed. Then a modified implementation of bidirectional Dijkstra is used to query the shortest path. The preprocessing allows for a significant speedup in query time. Research has been done on decreasing the query time as much as possible. There has not been any focus on trying to decrease the preprocessing stage of the Contraction Hierarchies algorithm at the cost of an increase in query speed. This is interesting for an individual trying to use the Contraction Hierarchies algorithm at a small scale or for personal use. In this thesis we investigate the effects of different heuristic and search methods on the preprocessing speed. We also look for the optimal search range to decrease the preprocessing speed. We managed to significantly lower the preprocessing speed using these optimisations. Additionally, an implementation of Contraction Hierarchies in C# is contributed to the community.

Contents

1 Introduction

Shortest path algorithms are used in daily life by numerous applications. For example when you commute to work, when mail is delivered to your house or when emergency services need to be somewhere as quickly as possible. They help us in many different ways and contribute to a better society. Repeatedly finding shortest paths in large graphs can be time consuming. Therefore, several approaches have been proposed that attempt to optimize the process, either by improving query times or introducing preprocessing. Efficient algorithms are critical when dealing with large-scale graphs, such as road networks or communication systems, where millions of nodes and edges must be processed. In this thesis, we will consider the Contraction Hierarchies algorithm, which aims to significantly speed up query times at the cost of increased preprocessing effort.
Contraction Hierarchies are particularly useful in static environments, where the graph structure remains mostly unchanged, allowing the computational cost of preprocessing to be justified by the significant gains in query efficiency. However, as the preprocessing time can become a bottleneck for dynamic graphs or environments with frequent updates, there is a need to explore methods to streamline this phase without compromising query performance too severely.

The shortest path algorithm we will focus on is the Contraction Hierarchies algorithm, which allows for extremely fast queries. This does however require the graph to be preprocessed extensively, which can take a long time. If a person wants to perform just a few small queries or if the graph is not static this is a major drawback and in those cases Dijkstra's algorithm would be a better choice. Dijkstra's algorithm however also has it's own drawbacks, for routes over a large distance Dijkstra's algorithm does not work well as the search space would increase drastically leading to much slower query times. Even in smaller graphs the Contraction Hierarchies algorithm has a lot faster queries. The results of [2] show that the Contraction Hierarchies algorithm was about 5 times faster than Dijkstra's algorithm and almost 6 times faster than the A* algorithm in a graph of about 5000 nodes and 10000 edges. In large scale commercial use, like Google Maps, having extremely fast query times is very important. There could be thousands of queries done by the server each second, so each query taking 10 to 100 times as long can make a big difference. For personal use the long preprocessing times can be a drawback instead. When only one person is doing a relatively small set of queries, it does not matter much if a query takes 0.01ms or 1ms. That is why we aim to decrease the time it takes to preprocess the graph, even if that leads to a slower query speed.

In this thesis, we describe a C# implementation of the Contraction Hierarchies algorithm. C# is a modern language which sees a lot more use today, especially in business environments. To the best of our knowledge, a C# implementation of this algorithm is currently not available. With this C# implementation we conduct a number of experiments to quantify the effect of different means to reduce the pre-processing time of the algorithm on the query time.

The implementation we created in C# differs from the implementation made by [8]. We have kept the algorithm simple, sticking close to the core of the algorithm. The implementation made by [8] uses additional speed up techniques. We did try to stay as similar as possible when it comes to the data structures, of course with a different programming language this is not completely possible. In Section 3 we go over the differences in the implementations in more detail.

Using our c# implementation, we have investigated methods to reduce the preprocessing time. We managed to reduce the preprocessing time of our simple Contraction Hierarchies implementation compared to the recommended parameters used by [8]. To achieve this we profiled the effects of different search ranges during the node importance and node contraction stages. Then we profiled different heuristics used during the preprocessing of the graph. The heuristics were taken from [8]. We also experimented with a bidirectional approach to node contraction. Finally we combined the fastest methods of the previous tests and compared it to our baseline, a C++ implementation [8] and a java implementation [5]. This also gave us insights into the benefits and difficulties of using C# as the language for our implementation.

This bachelor thesis was written at LIACS under the supervision of Kristian Rietveld. We start with Section 2 by going over definitions and explaining the Dijkstra's and Contraction Hierarchies algorithms. In Section 3 we introduce the implementation we made to conduct our research. In Section 4 we describe our experiments and their outcomes. In Section 5 we discuss future work and conclude our research.

2 Preliminaries

In this Chapter, we introduce the background knowledge necessary to understand the remainder of the thesis. We first introduce the graph notation that will be in this thesis and subsequently explain the Contraction Hierachies algorithm.

2.1 Definitions

We will give basic definitions for graph theory. Let $G = (V, E)$ be a directed graph, so each edge goes in a specific direction. Each edge has a weight, which is a cost to travel along the edge. For an edge $(v, w) \in E$ we usually denote the weight as $w(v, w)$. The weight of the path $P = \langle A, B, C, D\rangle$ would be denoted as $w(P)$. $n = |V|$ is the number of nodes and $m = |E|$ is the number of edges in the graph. A path P in G is a sequence of nodes connected by edges.

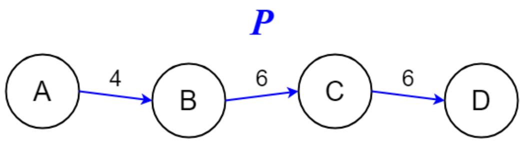


Figure 1: Path $P = \langle A, B, C, D\rangle$. Next to each edge is a number that represents the weight of the edge, so here the total weight of the path is $w(P)$ is 16. [13]

For each pair $(v, w) \in V \times V$ if v and w are connected by edges, so there is a path between them, then there is also a shortest path between them.

2.2 Contraction Hierarchies

The idea behind CH is that by contracting less important nodes first, we preserve the key paths between more central nodes, thus allowing us to simplify the graph without losing critical connections. When a node is contracted, we add shortcuts between its neighboring nodes to ensure that any shortest path that previously went through the contracted node can still be reconstructed. This way, the algorithm ensures that the shortest paths are preserved, even after many nodes are removed. As a result, when performing a query on the preprocessed graph, we can use the simplified structure to find the shortest path much more quickly. We achieve this by finding an ordering of nodes based on their importance in the graph and by contracting them from the graph in the same order. The importance of a node can be gauged by using centrality measures. Degree centrality determines the importance based on the number of direct connections the node has to neighbors. Betweenness centrality determines the importance based on how many shortest paths go through this node. Closeness centrality determines the importance by how close it is to all the other nodes in the graph. This means that a busy intersection node in the city center has a higher importance than some node in a suburb because it has a higher level of closeness centrality. To determine the importance, we use heuristics. We will further explain this in Section 2.2.1.

Once we have an ordering of nodes we contract the nodes in order of importance, the lowest importance first. If the shortest path from an incoming node to an outgoing node travels through the to be contracted node, we need to add a shortcut from the incoming node to the outgoing node. This is done to preserve the shortest paths in the graph. An example of this can be seen in Figure 2. A shortcut is added from v to w when u gets contracted. This shortcut is needed to preserve the shortest path from v to w in the graph.

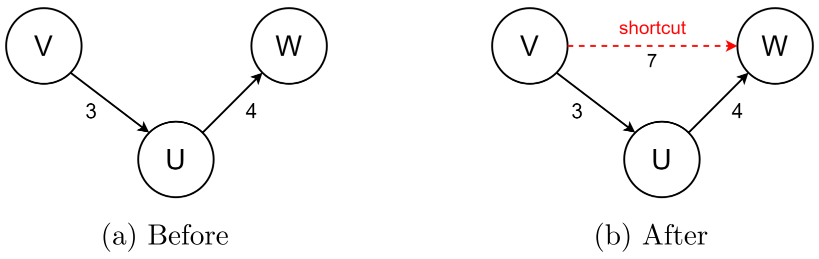


Figure 2: Node contraction of U, shortcut added from V to W to preserve the shortest path [8].

In Figure 3 there is a path $P = \langle v, x, y, w \rangle$ where $w(\langle v, u, w \rangle) \geq w(P)$. We call this path P a witness path. If there is a witness path, a shorter path from v to w not through u, we do not place a shortcut as the path through u has no use in being preserved.

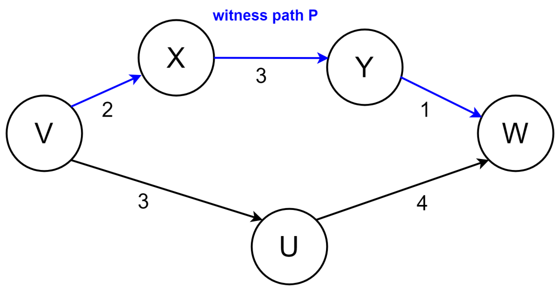


Figure 3: Node contraction of U, no shortcut added [8].

2.2.1 Node Order Selection

The order of the nodes has a big impact on the performance of the algorithm. The Contraction Hierarchies algorithm will give a correct shortest path when queried regardless of the node order. How much time it takes to preprocess the graph can however increase exponentially when a bad node order is used. When an important node is contracted early this leads to many shortcuts being placed. These shortcuts need to be taken into account on further contractions. This might lead to a node having only 4 edges originally to end up having 20 or more by the time we contract it. This also shows that the importance of a node changes when another node is contracted. After contracting a node we need to update the importance of the other nodes. To calculate the new importance of we use a number of existing heuristics described in [8]. We then multiply the different heuristics with weight parameters and sum them together. The default weight parameter values are taken from [8] as they offer a good balance between fast queries and acceptable preprocessing times.

2.2.1.1 Lazy Updates

The node with the smallest importance is contracted first. When we pop a node of our priority queue we can recalculate the importance using the heuristics and if the importance is still the lowest we contract the node, if it is not we put it back in the priority queue. "According to [8] this lazy update procedure is said to yield better results in practical settings and possibly in theoretical scenarios as well."

2.2.1.2 Edge Difference

When we contract a node we want to see the number of edges decrease as well. For this heuristic we simulate the contraction of the node. After the node is contracted we can determine how many shortcuts needed to be placed and how many edges were taken out of the graph. Then we calculate the difference and use the value to determine the importance of the node. This value is multiplied by a weight parameter. We use as a default weight parameter the value 190.

2.2.1.3 Contracted Neighbors

We count the number of neighbors that are already contracted. Whenever we contract a node we increase the contracted neighbors count of its neighbors by 1. This can help reduce the search space when querying the graph [8]. We multiply this number by a weight parameter. As a default value we used 120.

2.2.1.4 Original Edge Space

The target node of a shortcut has a number of edges. This is the number of original edges that the shortcut represents. We sum this for all the shortcuts placed during a contraction and use its value to determine the importance of the node. "According to [8] this property avoids shortcuts that represent too many original edges." We multiply the value with a weight parameter. The default value we use is 600.

2.2.1.5 Search Space

The goal is to minimize the length of the shortest paths trees generated by a modified Dijkstra's algorithm which speeds up the query time. The order in which nodes are considered for contraction matters because the later a node is contracted, the more likely it is to increase the depth of a shortest paths tree. We multiply the search space with a weight parameter. The default value we use is 1.

2.2.2 Node Contraction

To contract a node we use the algorithm shown in Figure 4. Lines 2 to 5 from Figure 4 describe how for each incoming edge of node u we look for witness paths to all outgoing edges of node u. We use Dijkstra's algorithm to do a local search and look for these witness paths. The node to be contracted is taken out of the search graph. Once we find all the target nodes we can stop the search. Line 4 of Figure 4 contains "may be", this is because we can stop the search early by setting a search limit. If we do not find a shorter witness path within the search limit we place the shortcut anyway because the path through u might be the only shortest path. After this is done for all the incoming edges of node u we can remove node u from the graph and set its node level to the order at which it was contracted. After this we continue with the next node.

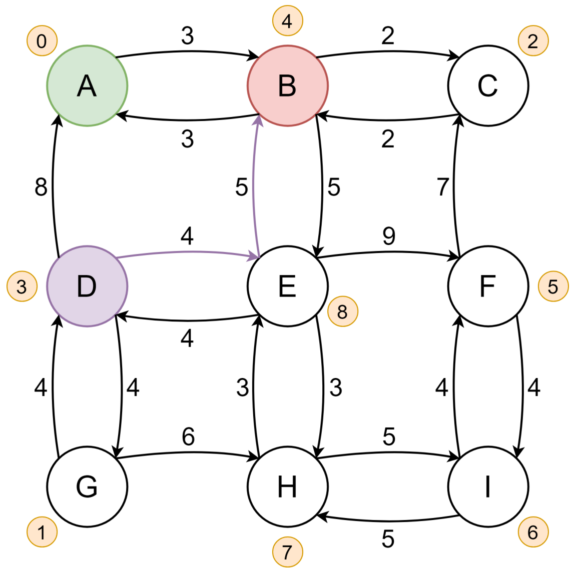
Algorithm 1: SimplifiedConstructionProcedure($G = (V, E)$,$<$)

```
1 foreach $u \in V$ ordered by $<$ ascending do
2     foreach $(v, u) \in E$ with $v > u$ do
3         foreach $(u, w) \in E$ with $w > u$ do
4             if $\langle v, u, w \rangle$ "may be" the only shortest path from $v$ to $w$ then
5                 $E := E \cup \{(v, w)\}$ (use weight $w(v, w) := w(v, u) + w(u, w)$)
```

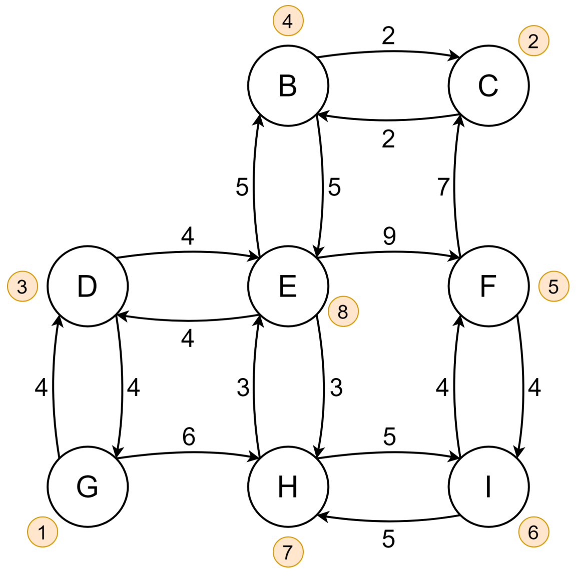
Figure 4: Algorithm for node contraction taken from [8].

Example

We will go over two examples of node contractions. In the first example in Figure 5 we are in the process of contracting node A. We have two incoming edges. One from node D and one from node B. The path from node D to node B has a higher cost through node A than through node E. In this case we found a shorter witness path which means we do not place a shortcut from node D to node B. The second incoming edge goes from node B to node B so we do not have to do anything.



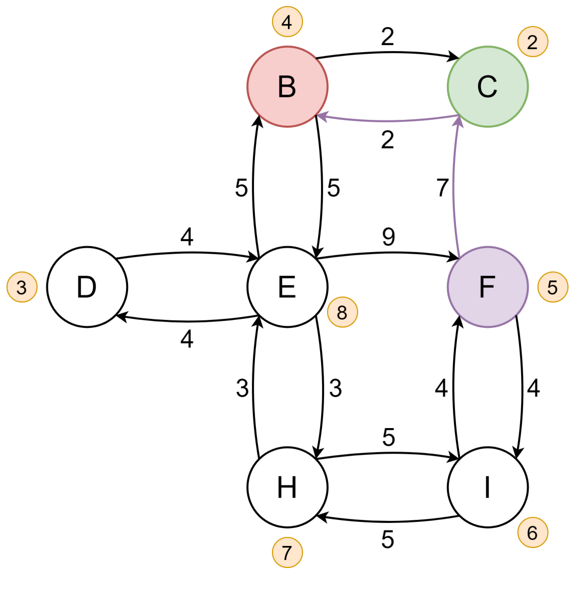
(a) Before



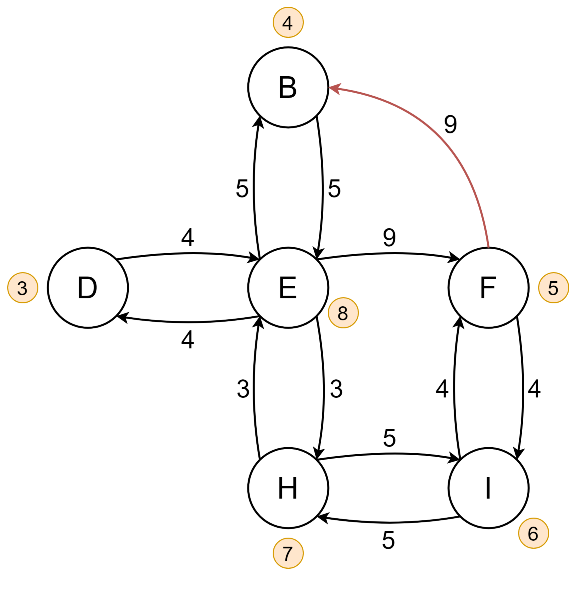
(b) After

Figure 5: D to B through A: $8 + 3 = 11$, D to B through E: $4 + 5 = 9, 9 < 11$ so no shortcut needed for shortest path. Node A is contracted.

In the second example in Figure 6 we are contracting node C. Again we have two incoming edges. This time the path from node F to node B through node C is shorter than the witness path through nodes I, H and E. In this case we need to add a shortcut from node F to node B to preserve this shortest path. The other incoming edge goes from node B to node B again as we only have 1 outgoing edge, so we do not have to do anything.



(a) Before



(b) After

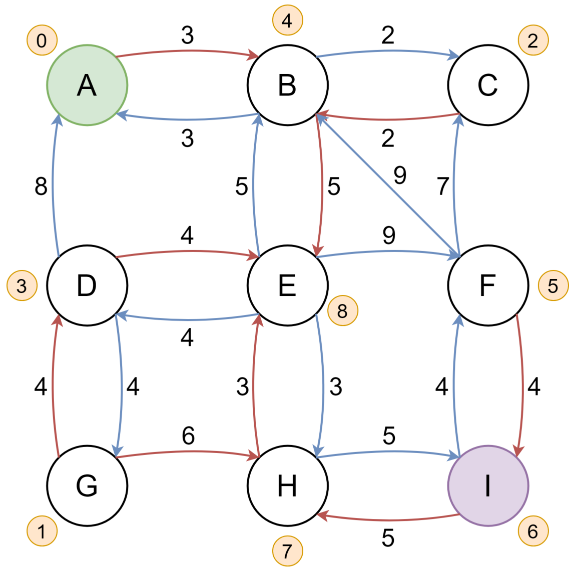
Figure 6: F to B via C: $7 + 2 = 9$, F to B through I, H and E: $4 + 5 + 3 + 5 = 17, 9 < 17$ so a shortcut is needed. Node C is contracted.

2.2.3 Query

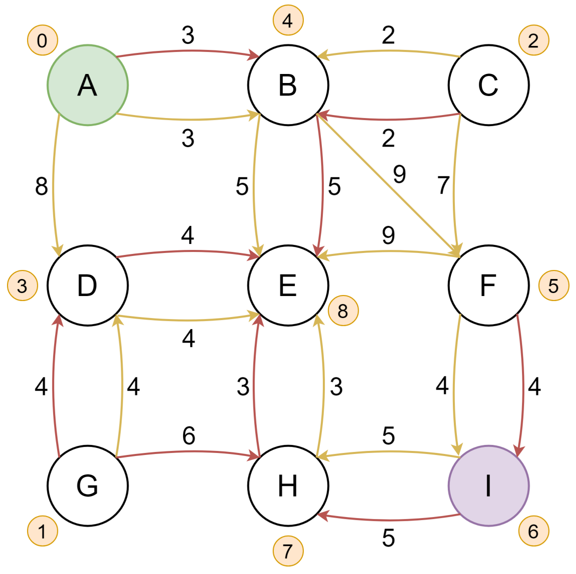
To query a shortest path from the preprocessed graph we use a bidirectional Dijkstra-like algorithm. The nodes have a node level, the order number at which they were contracted. The preprocessed graph is split in two. We call them the upward and downward graphs. The upward graph contains the edges from v to w where the node level of $v > w$. The downward graph contains the edges from v to w where the node level of $v < w$. So if $v > w$ then v was contracted after w. As each edge always leads to a node of a higher level, an edge can only be in one of the graphs, either the upward graph or the downward graph. This allows for extremely efficient searching as will be shown in the examples that follow and is the hierarchy part of the Contraction Hierarchies algorithm.
We do a backward search from the target node and a forward search from the source node. For the backward search we flip the direction of the edges of the downward graph. We then use Dijkstra's algorithm for a complete forward and backward search. We have two priority queues, one for the forward search and one for the backward search. We take the lowest node out of the two and settle it. Once all nodes are settled we have a set of nodes that have been settled by both searches, we call this set L. To determine the shortest path we sum for each node of L the cost of the paths found by both searches to get there. The node in L with the lowest cost is the shortest path. Not all edges and nodes in the upward and downward graph are relevant during a query. Some cannot be reached. This can be seen in Figure 10.

Example
We will look at an example of a query. In Figure 7a we see the upward and downward graphs. We

flip the downward graph by using the reversed edges, we see this is Figure 7b. We will query the path from node A to node I.



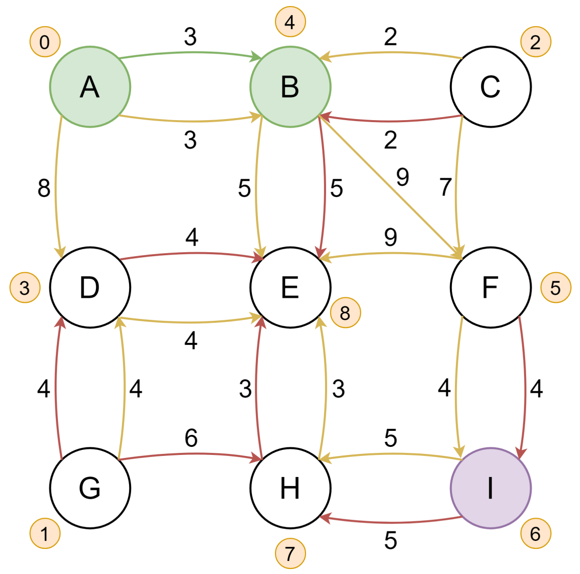
(a) Upward graph (red) and downward graph (blue)



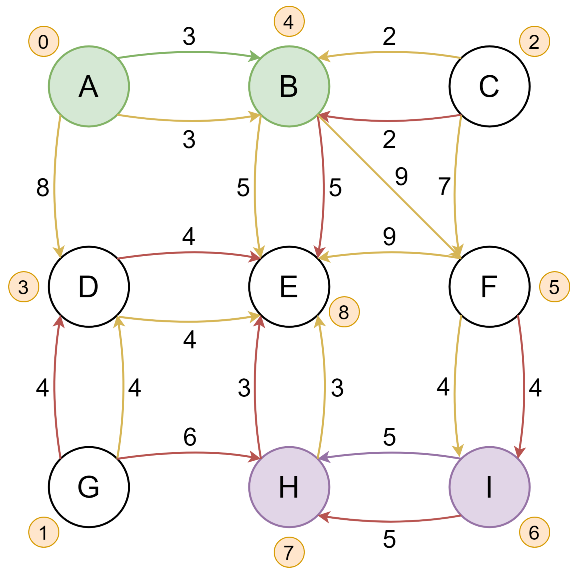
(b) Upward graph (red) and flipped downward graph (yellow)

Figure 7: Node A is the source node and node I is the target node. In orange we see the node order.

We proceed to take the node with the lowest cost of both the upward and downward priority queues. This is node B from the upward priority queue. We visit this node in Figure 8a. Then we take the next node from one of the priority queues. This is node H from the downward priority queue. We visit it in Figure 8b.



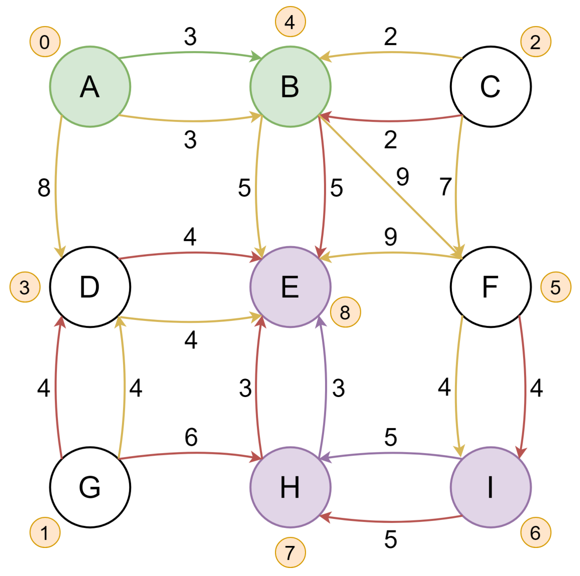
(a) Upward priority queue: 3. Downward priority queue: 5. Take from upward priority queue.



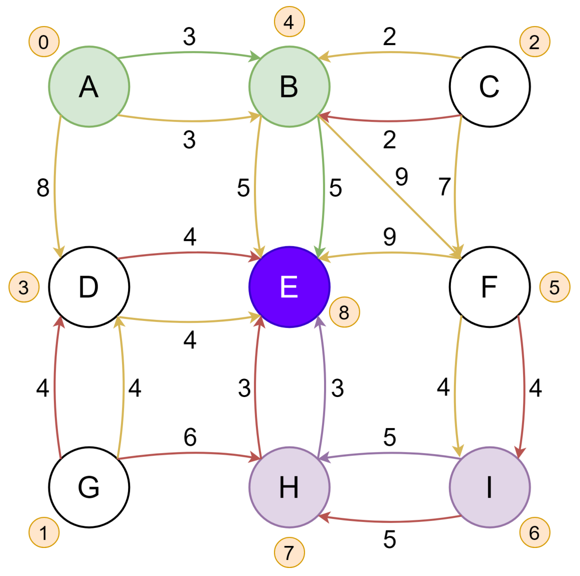
(b) Upward priority queue:5. Downward priority queue: 5. Take from downward priority queue.

Figure 8: Visited nodes B and H

The next lowest node from one of the priority queues is node E from the downward priority queue. We visit the node in Figure 9a. This leaves the downward priority queue empty so we take the next node from the upward priority which is also node E. We visit it in Figure 9b. Now the upward priority queue is also empty so the search is done. We look at the set L of nodes that have been visited by both the upward and the downward search. We calculate for each node the cost to get there from both sides. In our example we only have one node, E, so this is also the shortest path. If there were more nodes in L the node with the combined lowest cost would be the shortest path.



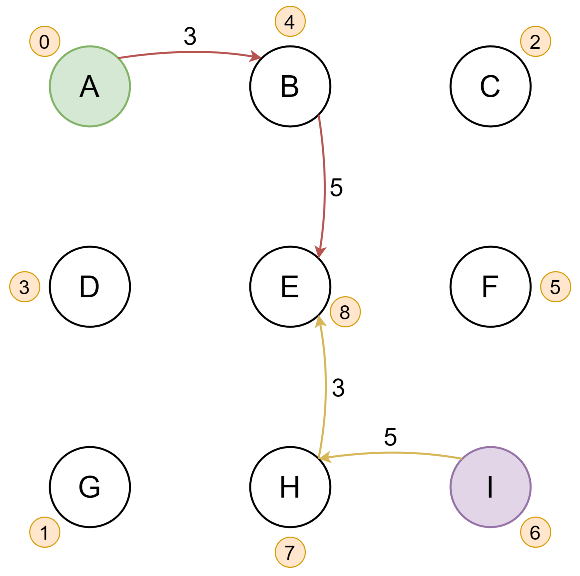
(a) Upward priority queue: 5. Downward priority queue: 3. Take from downward priority queue.



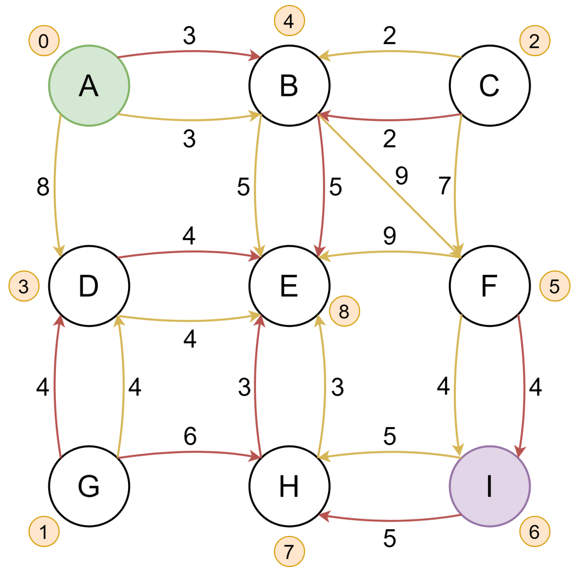
(b) Upward priority queue:5. Downward priority queue: empty. Take from upward priority queue.

Figure 9: Node E is visited by both searches. Save node E in set L with the cost of both sides. After this both queues are empty so the search is finished.

We can see in Figure 10 that we only had to traverse 4 edges in a graph containing 21 edges. This shows the advantage Contraction Hierarchies can have over other non hierarchical shortest path algorithms.



(a) Edges used during query.



(b) All edges in the graphs

Figure 10: Not all edges and nodes are relevant in a search.

2.3 Related work

Dijkstra's Algorithm [7] operates by iteratively selecting the node with the lowest shortest path distance to the source node from a priority queue and updating the distances of its neighbors. The neighbors are updated with the distance of the source to the selected node plus the distance of the selected node to the neighbor. Once all the neighbors have been updated, the selected node is settled and the next node is taken from the priority queue. When the target node is updated the algorithm is done. An improvement on Dijkstra is to do a bidirectional search [9]. In this case we have two priority queues. One for the forward search from the source node and one for the backward search from the target node. Once we find a node that has been settled by both searches we are done. The shortest path is the two shortest paths from both directions combined.

Highway Hierarchies (HHs) [10, 11, 12] divides the nodes and edges in different hierarchical levels based on the importance of the nodes and edges. This is done in two ways, node reduction which bypasses low degree nodes with new shortcut edges and edge reduction which removes edges that only appear on shortest paths close to the source or target. As a result, we have a smaller search space. The query algorithm is similar to bidirectional Dijkstra. When the search is not close to the source or target edges of low importance do not need to be expanded. The difference between Highway Hierarchies and Contraction Hierarchies is that the Contraction Hierarchies focuses more on node-based simplifications and Highway Hierarchies focuses more on edge-based classification and region-based searches. They both use the concept of hierarchies in graphs but differ a lot in implementation.

3 Implementation

We have implemented the Contraction Hierarchies algorithm and heuristics described in Chapter 2 in the C# programming language. In this Chapter we will go over implementation details for the data structures and algorithms used by our C# implementation of the algorithm.

3.1 Graph Data Structures

To store the structure of the graph we use two different data structures. One during the construction of the hierarchy and the other while searching the graph for queries. We used a similar implementation as [8], which uses an adjacency array for both data structures. The adjacency array is a very space-efficient data structure. It also allows for very fast traversal of the graph. We have two arrays, one is used for the nodes of the graph. We know beforehand how many nodes there are in the graph, so this array has a fixed size. We keep a second array for the edges. The number of edges increases as shortcuts are placed, so this array needs to be able to grow. We will go over how we manage this in Section 3.1.1. The edges are grouped by source node. This way we only have to store a target node and weight for each edge. The node stores the position of the first outgoing edge and the last outgoing edge. We need to be able to do a backward search, so for each edge (u, v) we also store a backward variant (v, u). To make this more space efficient we store two flags on an edge, one backward flag and one forward flag. This way an edge can be both at the same time whilst only having the be stored once. As most edges in a road network are two way streets, this saves a lot of space. Where our implementation differs from [8] is in the ordering of the edges. They sort the edges so first all the edges (u, v) have a node ordering of $u > v$ and then all the edges have a node ordering of $u < v$. To split these they have an extra pointer on the node to point to the first edge where $u < v$. This adds extra complexity to the implementation. We solved this by going over all the edges and checking if they have the right node order for our search. This is slightly less optimised but a lot simpler.

3.1.1 Updateable Graph

In the updatable edge array we need to be able to add new edges when a shortcut is needed. We do this by keeping extra space for growth. This means that between the last edge of one node and the first edge of the next node there are a few empty places. If these empty places are full we move the entire edge group to the end of the array. We then update the node pointers to the first and last edge positions. At the same time we increase the amount of empty places this edge group has. Most of the edge groups will see only a small number of shortcuts added and will not be moved as they do not need a lot of space to grow. The edge group section left behind will be made empty and can be used by the edge group before it in the array if needed. The data structure used for the resizable array by [8] is a C++ STL vector. As we worked with C# we had to use a different data structure. For this we used the List from System.Collections, the default List in C#. It has similar features to a STL vector and is also a dynamic array.

3.1.2 Search Graph

During the searches for queries the edge array does not need to be resizable. As a search in the downward graph is similar to searching the backward edges of the upward graph. We only need to store edges (u, v) with a node ordering of $u < v$. We know exactly how many edges there are so we do have empty spot in the array. The nodes array stays the same, only the pointers to the edges need to change as the edge array is a lot smaller. As the data structure does not need to be resizable we used a normal array instead of a list.

3.2 Priority Queue

We make a lot of use of priority queues. To store the nodes that still need to be contracted, and also for the Dijkstra algorithm used during node contraction and the bidirectional Dijkstra algorithm used during the search for a query. We tried two different priority queues. The standard C# priority queue from System.Collections and a custom priority queue [3]. The standard priority queue is not capable of updating the priority of a key already in the queue. The custom priority queue is able to do this. This is needed because the importance of nodes changes as other nodes are contracted from the graph. What we found was that the standard priority queue is much better optimised and gave significantly faster results. To be able to use the standard priority queue we came up with a work around. On the node we keep track if the node has already been contracted. Nodes that need to have their importance updated are added to the priority queue again. If the importance is lower than it was before, this will ensure that the node is taken from the priority queue at the right time. If the node has a higher importance, this does not have any negative effect as we will just skip it. This is something we do differently from [8] where they do have the capability to update the priority of a key.

3.3 Algorithms

We have made our own implementations for the Contraction Hierarchies algorithm, Dijkstra's algorithm and the bidirectional Dijkstra's algorithm. In this section we will go over their implementation. For a more in depth view please refer to the code [1]. We did not use any techniques to increase the performance of the algorithms and stuck close to their base descriptions. The implementation of [8] does make use of these kinds of techniques. For example by using Hop Search with changing limits and degrees during local searches, reducing edges on the fly and optimising the graph beforehand by reducing nodes that are not needed. During testing they also used cache warm up to achieve the best performance.

3.3.1 Dijkstra's Algorithm

We use Dijkstra's Algorithm to perform local searches in order check if a shortcut is needed. We have a source node, a max cost, a number of target nodes and we set a target flag on the target nodes to true. We do a standard Dijkstra's search from the source node until the max cost is reached for all possible paths. When we find a node that is a target node and the cost of the path is higher than the path through the to be contracted node we place a shortcut. If all target nodes are found we can stop early.

3.3.2 Bidirectional Dijkstra's Algorithm

We use the Bidirectional Dijkstra's Algorithm to search the graph for a query. We start with a source and a target node. We use two priority queues, one for the source node and one for the target node. From the source node we search the forward edges from the target node we search the backward edges. We do a complete search for both, taking the node with the lowest priority of both queues each time. Once these searches have completed we are left with a set of nodes that have been found by both searches. We calculate for each node the combined cost of the forward and backward paths. The lowest combined cost is the shortest path.

3.3.3 Contraction Hierarchies

The first step is preprocessing the graph. We start by calculating the importance of all the nodes and filling the priority queue. To calculate the importance we perform the calculations described in Section 2.2.1. Then we loop over the priority queue until its empty. When popping a node from the queue we check if it has not already been contracted and if the importance with which it was stored is up to date with the importance set on the node. If either condition is false the node is an old duplicate and can be skipped. Then we recalculate the importance of the node to see if it still the lowest. If it is, then we contract the node and set the node level on the node. The node level is the order number at which it was contracted. If it does not have the lowest importance anymore we put it back on the queue. During node contraction we use our implementation of Dijkstra's Algorithm for each incoming node. This places the necessary shortcuts.
The second step is creating a search graph and performing the searches for queries. We create the graph data structure described in Section 3.1.2 and use the Bidirectional Dijkstra's Algorithm described in Section 3.3.2 to search this graph. During the search we set flags on the nodes as they are found. This makes the graph dirty, to solve this we make a deep copy of the nodes array. After the search has been completed we copy these backup nodes to the nodes used during the search so they are clean and ready for another search.

3.4 Utils

We got the OpenStreetMap data to create the graphs from [4]. We needed to do some data engineering to get this into the right format. For this we have used [6] to transform the .bpf files into .csv files. After this we used our Python script, which can be found in [1], to calculate the edge costs and transform the data into the right format described in [8]. This allows us to use the same files in both implementations. This can be reused for future work or experiments on different road networks.

4 Experiments

The Contraction Hierarchies algorithm can take a long time to preprocess the graph. With these experiments we are looking to speedup this process. We will combine the fastest implementations for each part of the preprocessing stage and compare this with our baseline and a different highly optimised implementation. We will see if speeding up the preprocessing stage is possible and if it comes with a cost in query speed. We will also compare the performance of our C# implementation with the performance of the C++ implementation from [8] and the Java implementation from [5].

4.1 Experiment Setup

4.1.1 Environment

We performed the experiments on one core of a single Intel Core i5-13600k with 32GB main memory running Windows 11. The project is written in C# using .NET framework 7.0. The project is compiled by Visual Studio using language version C# 11 and the release settings. The code is then run from an executable file.

4.1.2 Parameters

We use a number of heuristics during the importance calculations. These heuristics are described in more detail in Section 2.2.1. Each heuristic uses a weight coefficient to come to one general importance value when combined. The weights we used for the coefficients can be seen in Table 1. There we also see the limits of the local search ranges that are done during the importance calculations. As default values for these parameters we use the values given by [8] for their economical variant that prioritises a low product of construction time and average query time. The values for the parameters can be seen in Table 1.

Search Range Importance	1000
Search Range Contraction	1000
Contracted Neighbor	120
Search Space	1
Edge Difference	190
Original Edges	600

Table 1: Contraction Hierarchies parameters

4.1.3 Data sets

All our experiments have been done on four road networks. These road networks have been downloaded from [4] and contain OpenStreetMap data. We transformed the .bpf files to .csv using [6]. Then we transformed the .csv file to the .ddsg format described in [8] with our own Python script. For each edge we know its length and its road category. With this we give each edge a cost to traverse, a travel time, by dividing the distance by the speed limit. We only test on road networks because the Contraction Hierarchies algorithm is designed to exploit properties of graphs

that represent road networks. Road network graphs are directed, but as most roads are two way streets also largely symmetric.
The road networks we used are Flevoland, Zeeland, Corse and Guadeloupe. The number of nodes and edges each network contains can be seen in Table 2.

Graph	**Nodes**	**Edges**	**One-way**
Flevoland	99106	265612	13704
Zeeland	100683	329120	11578
Corse	99684	289738	7215
Guadeloupe	70364	260960	8637

Table 2: Road networks used and their contents

4.1.4 Preliminary Remarks

Only the shortest-path length is computed, without outputting the actual route.

4.2 Methodology

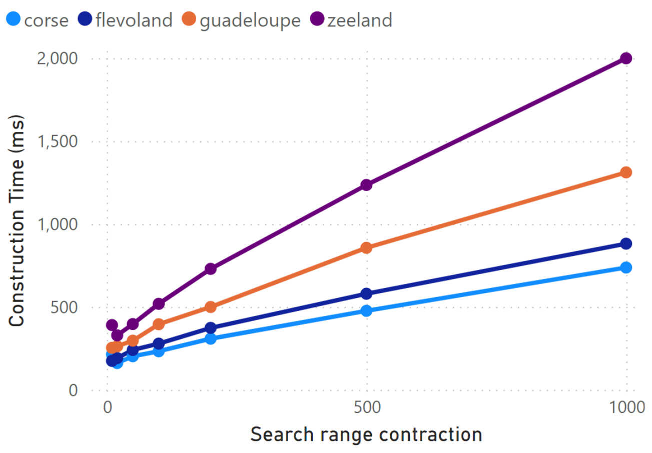
We randomly created a set of a 1000 unique queries for each road network. A query consists of a source-target pair. For each query we measure the time it takes to find the shortest path from the source node to the target node. All the queries are measured together and the result we are interested in is the average query time. The query sets stay consistent over all the experiments. Each experiment is done only once unless specified otherwise.

4.3 Search Range

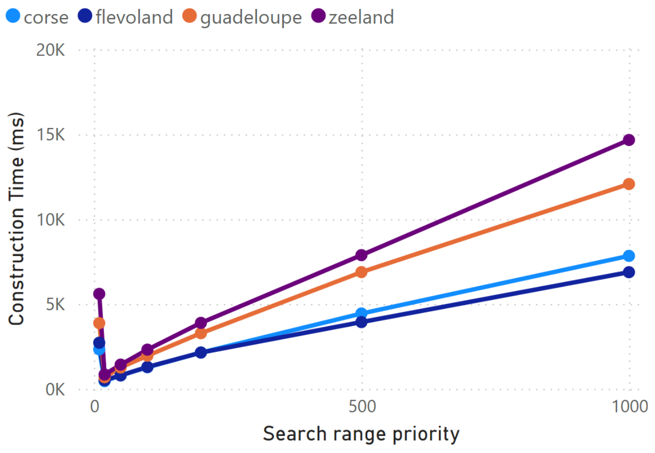
The Contraction Hierarchies algorithm uses Dijkstra's algorithm to perform local searches. These searches have a limit for the total number of nodes that may be visited. This limit is the search range. The local search is performed during the importance calculations for a node and during the contraction of a node. The chance of finding a shorter path decreases as the search range gets smaller. If a shorter path is not found within the search range we add a shortcut. Added shortcuts increase the complexity of the graph. This slows later contractions down. With more shortcuts the query speed should slow down and get worse as well, since the graph gets bigger. So, a smaller search range might reduce the time spent in local search at the cost of more shortcuts potentially slowing the query and future contractions down. With this experiment we hope to find out if the lower search time leads to faster preprocessing or if the added shortcuts will slow the preprocessing down. We will first increase the search range during the importance function from 10 to 1000 nodes while keeping the search range for the contraction function at 1000 nodes. We report the total preprocessing time. Next, we will do the same for the contraction function. The default search range for both is 1000. We will also increase the search ranges for the contraction and the importance functions together. They will have the same search range going from 10 to 1000 nodes.

4.3.1 Results

We can see in Figure 11a and Figure 11b that both importance and contraction have the shortest construction time across all road networks at a search range of 20 nodes. From 20 to 1000 nodes we see a steady increase in construction time and from 10 to 20 nodes we see the construction time get smaller. This means that with a search range of 20 nodes we still find shorter paths, but with 10 nodes we do not find them as often. This leads to a larger increase in shortcuts and slows down the construction. This difference is especially large in the importance function where we see a big drop in construction time between 10 and 20 nodes. A good importance order is essential to the algorithm and if done improperly it can cause a snowball effect, leading to even more shortcuts needed. We can see see this snowballing effect in the differences between the graphs. The construction time for Zeeland and Guadeloupe grows faster than the others.



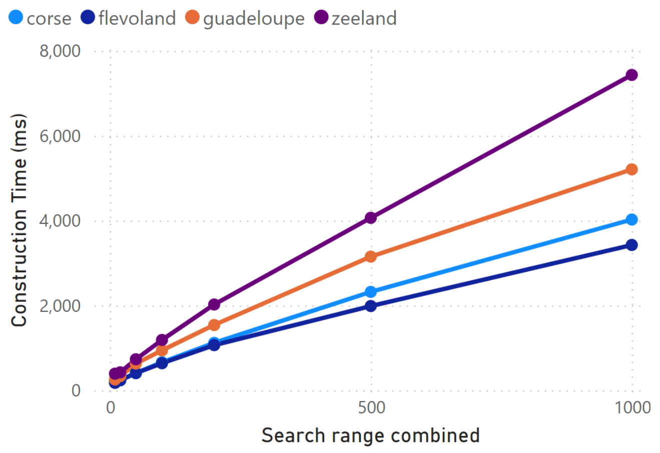
(a) Contraction



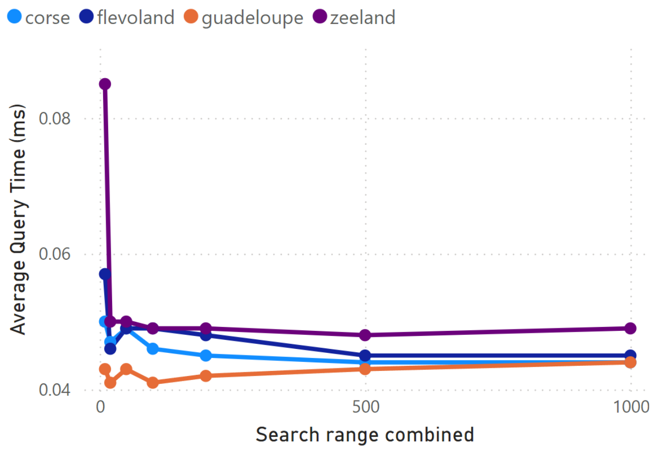
(b) Importance

Figure 11: Construction time for different search ranges for the contraction and importance calculations.

Now we combine the search ranges and set both to the same value. The results of which can be seen in Figure 12a. When we look at the results of the contraction and importance function together, we see that Figure 12a looks more like the results of the contraction test in Figure 11a. This would suggest that the contraction stage has a larger impact on the construction time than the importance stage. The drop in construction time we saw in Figure 11b going from 10 to 20 nodes is also no longer there. The average query speed can be seen in Figure 12b and is at its lowest with a search range of around 500 nodes when taking all four road networks into account. Only Guadeloupe shows a lower average query time at a range of 20 to 200 nodes. For all four road networks we see a drop in average query time going from 10 to 20 nodes. Zeeland's drop is however significantly larger.



(a) Construction time



(b) Query time

Figure 12: Search range results for contraction and importance together

4.3.2 Conclusion

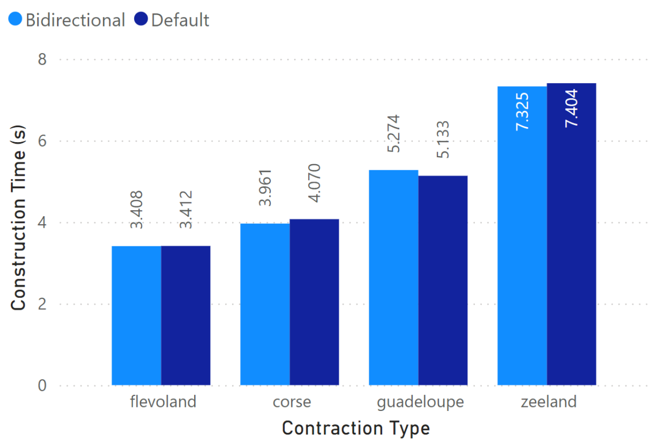
The best performance in construction time is at a search range of 20 nodes for both contraction and importance. We did see some big changes in query time. Going from 10 to 20 nodes we see Zeeland's query time decreased by roughly 41%. After this is stayed roughly the same going from 20 to 1000 nodes. Flevoland had a similar, but smaller, drop in query time going from 10 to 20 nodes. It decreased by roughly 19%. Guadeloupe did not see a significant change in query time across all number of nodes, staying between 0.041ms and 0.044ms. Corse did not change much either, decreasing from 0.05ms to 0.044ms while going from 10 to 1000 nodes. The big change in Zealand's query time going from 10 to 20 nodes seems like an outlier, the reason for this could be some missed shortest paths because of the low search range which caused other bad importance decisions.

4.4 Bidirectional Contraction

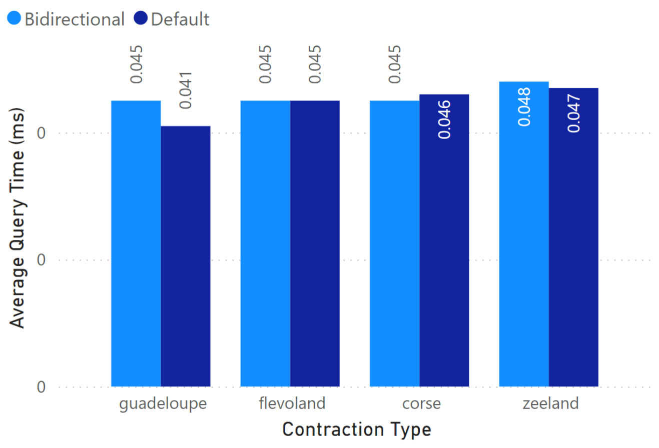
Most edges in a road network are bidirectional. This is also shown in Table 2, where the percentage of one-way streets ranges from 2.5% to 5.2%. We aim to improve the construction time by adding a shortcut in both directions if both edges to the node to contract are bidirectional. If there is a shorter path that uses one-way streets this leads to an unnecessary shortcut being placed. In return we only need to do one search instead of two.

4.4.1 Results

In Table 3 we can see that this leads to a large number of shortcuts being added, roughly 10% more. For Corse, Zeeland and Flevoland this decreased the construction time only very slightly. Guadeloupe however saw its construction time increase very slightly. The average query time did not change by much. In Figure 13b we can see that Zeeland and Guadeloupe saw an increase of construction time, Corse saw a decrease and Flevoland did not see any change. The differences are so small however that they are insignificant.



(a) Construction time



(b) Query time

Figure 13: Execution time of bidirectional contraction compared to the default.

Graph	ContractionType	Shortcuts	AvgQueryTime (ms)	ConstructionTime (s)
Flevoland	Default	54589	0.045	3.412
	Bidirectional	58876	0.045	3.408
Zeeland	Default	91818	0.047	7.404
	Bidirectional	101867	0.048	7.325
Corse	Default	44389	0.046	4.070
	Bidirectional	49034	0.045	3.961
Guadeloupe	Default	63438	0.041	5.133
	Bidirectional	70916	0.045	5.274

Table 3: Execution time of bidirectional contraction compared to the default.

4.4.2 Conclusion

The differences are so small that the added complexity to the code is not worth the effort. The construction time only decreased by 1.07% for Zeeland, 0.12% for Flevoland, 2.68% for Corse and -2.75% for Guadeloupe. This averages out to an 0.28% decrease across all 4 road networks. Bidirectional contraction did not deliver the expected results and did not really improve upon the default contraction method.

4.5 Simplification of Importance Computation

To determine the importance we use a number of heuristics. We discussed these heuristics in Section 2.2.1. In this experiment we will add the heuristics to our importance calculation one by one to see the effect they have. By making the importance simpler and faster to calculate we hope to decrease the construction time. The heuristics and their weight can be seen in Table 4. Choosing the right weights is very important. Each heuristic calculates a value. To make sure the importance of a heuristic is reflected by their value, we use these weights to correct them. The weights we have chosen come from [8]. Each number contains all previous heuristics as well. As some heuristics depend on the results of other we hope to see a positive or negative impact after a new heuristic is added. Other orders of heuristics are possible as well, but heuristic 2 and 5 need the result of heuristic 1.

Number	**Heuristic**	**Weight**
1	The number of shortcuts added	1
2	The edge difference	190
3	The number of contracted neighbors	120
4	The original edge space	600
5	The search space	1

Table 4: Considered heuristics and their weight in importance calculation

4.5.1 Results

We can see in Figure 14 that the differences between heuristics are quite small. On each graph the effect is different. In Zeeland we the construction time increasing for each heuristic we add. Guadeloupe increases sharply when adding heuristic 2, but decreases again when adding heuristic 4. Corse sees its construction time decreasing slightly for each added heuristic until we add heuristic 5 after which it rises again every so slightly. Flevoland sees an increase in construction time when heuristic 2 is added and decreases again for each heuristic added afterwards, ending up with a slightly faster construction time using all 5 heuristics. In Table 5 we can see the number of shortcuts. Heuristic 2 and 4 cause an increase in the number of shortcuts added. When heuristic 3 is added we do not see any change in the number of shortcuts, the construction time however is slightly lower. Heuristic 5 decreases the number of shortcuts added for all 4 road networks. In Figure 15 we can see an outlier for Flevoland at heuristic 5. What might have caused this is not clear. When we combined just heuristics 1 and 5 for our optimal implementation in Figure 16b we did not see a spike in average query time. In the rest of the cases we only saw small changes in the average query time with no clear pattern between the different road networks.

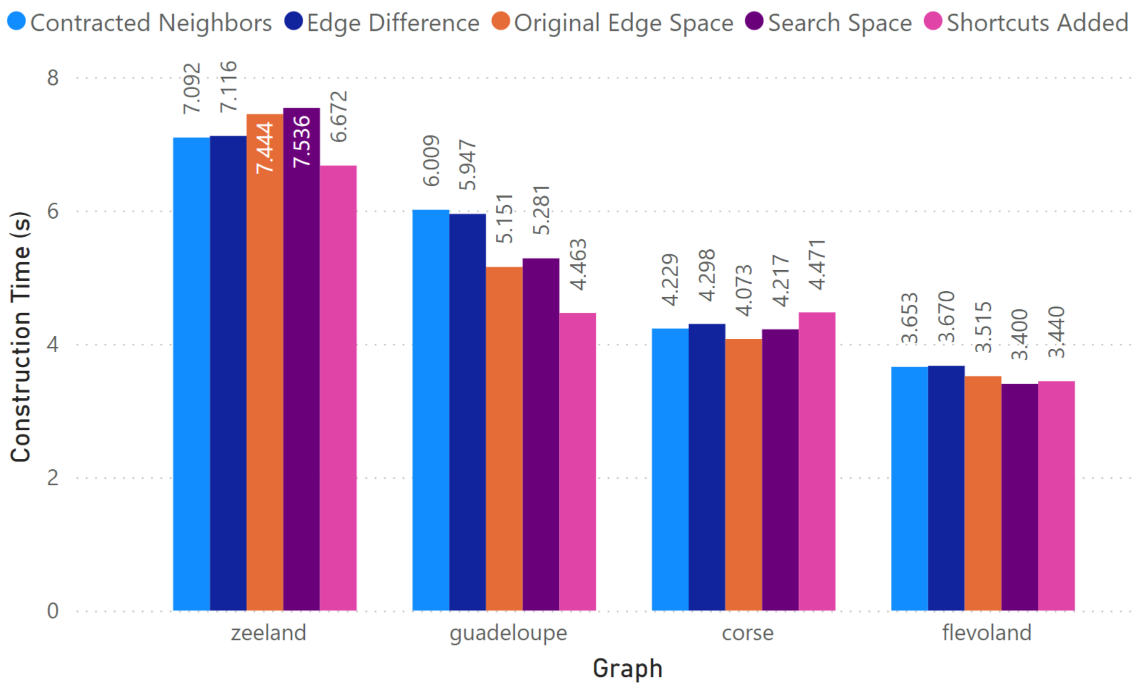


Figure 14: Construction time for different combinations of heuristics

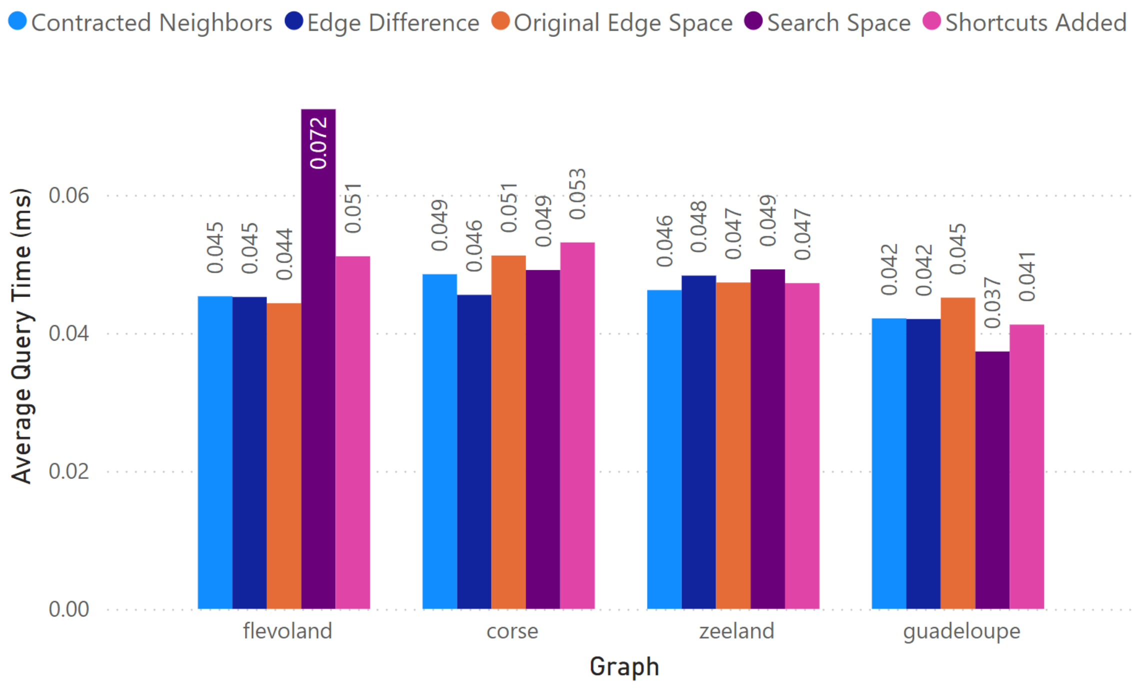


Figure 15: Query time for different combinations of heuristics

Graph	**Heuristic**	**Shortcuts**	**Avg. Query Time** (ms)	**Construction Time** (s)
Flevoland	1	48195	0.0511	3.440
	2	50307	0.0452	3.670
	3	50307	0.0453	3.653
	4	56068	0.0443	3.515
	5	54589	0.0724	3.400
Zeeland	1	75872	0.0472	6.672
	2	81731	0.0483	7.116
	3	81731	0.0462	7.092
	4	92731	0.0473	7.444
	5	91818	0.0492	7.536
Corse	1	41142	0.0531	4.471
	2	39844	0.0455	4.298
	3	39844	0.0485	4.229
	4	44946	0.0512	4.073
	5	44389	0.0491	4.217
Guadeloupe	1	55528	0.0412	4.463
	2	65824	0.042	5.947
	3	65824	0.0421	6.009
	4	63514	0.0451	5.151
	5	63438	0.0373	5.281

Table 5: Importance type results

4.5.2 Conclusion

Heuristic 1 has the lowest construction time for Zeeland and Guadeloupe. In the case of Corse and Flevoland the results are all very close to each other. The construction time is at its lowest for Corse with Heuristic 4 added. In the case of Flevoland this is with all 5 heuristics added. We can see that heuristic 5 has a very positive effect on the number of shortcuts, this is lower for all four road networks. Going from heuristic 1 to 5 we saw an increase in construction time of 12.95% for Flevoland, 18.33% for Guadeloupe, -5.68% for Corse and -1.16% for Flevoland. Across all 4 road networks this gives us an average increase in construction time of 6.11%. We expected the construction time to be lower with a simpler importance calculation. This is reflected by our results. The query time did not change much, we expected the query time to be larger with fewer heuristics used.

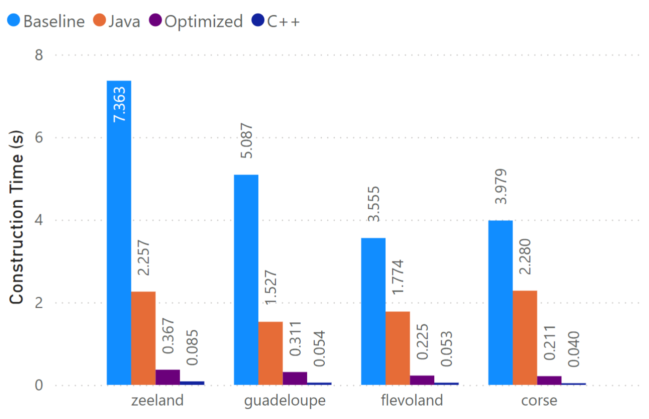
4.6 Comparison

We compare our baseline implementation that uses the default contraction type and all 5 heuristics with our optimized version that only uses heuristics 1 and 5. We kept 5 as it has a very positive effect on the number of shortcuts added, as we saw in Figure 14. We did increase the weight of heuristic 1 to 10 instead of 1. Heuristic 1 had the best construction times so we want to make it the most important.

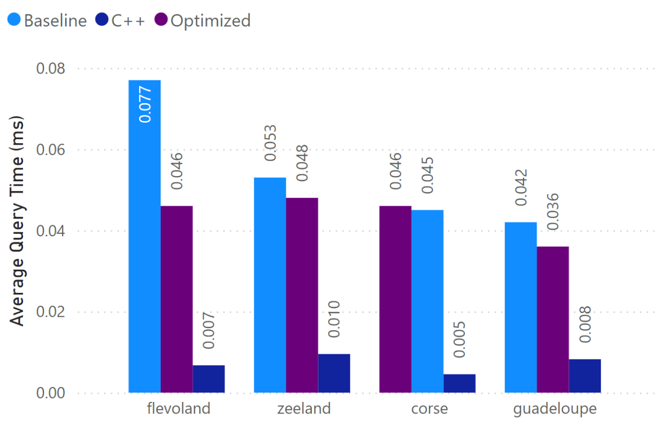
We also compare these two C# implementations with a C++ implementation from [8] and a Java implementation from [5]. The C++ implementation uses similar data structures but uses additional techniques to speed up the process. Differences between our implementation and the implementation from [8] are discussed in the Section 3. The heuristics used by the C++ implementation are edge difference, original edges term and search space size. The Java implementation uses different data structures. It saves the nodes on an array and on each nodes stores all outgoing and incoming edges.

4.6.1 Results

We can see in Figure 16a that our optimized version performs a lot better in terms of construction speed compared to our standard version. Compared to the Java implementation we can see that our default implementation is quite a bit slower, but our optimized implementation is a lot faster. The C++ implementation however is still much quicker than even our optimised implementation. If we look at Figure 12b we can see that the query time has also improved with the optimized implementation. In Table 6 we can see that the number of shortcuts with the optimized implementation is lower in most cases, the exception being Guadeloupe. If we compare the number of shortcuts between our implementation and the C++ implementation we see that for Corse we have a lower number of shortcuts, for Zeeland and Guadeloupe we have a higher number of shortcuts and for Flevoland we have a similar number of shortcuts.



(a) Construction time



(b) Query time

Figure 16: Comparison between implementations

Graph	Version	Shortcuts	AvgQueryTime (ms)	ConstructionTime (s)
Flevoland	Default	54589	0.052	3.532
	Optimized	54581	0.046	0.225
	C++	58604	0.0067	0.0529
	Java	-	-	1.774
Zeeland	Default	91818	0.052	7.417
	Optimized	83905	0.048	0.367
	C++	71731	0.0095	0.0848
	Java	-	-	2.257
Corse	Default	44389	0.045	4.192
	Optimized	41634	0.046	0.211
	C++	52616	0.0045	0.0403
	Java	-	-	2.280
Guadeloupe	Default	63438	0.042	5.212
	Optimized	65574	0.036	0.311
	C++	53041	0.0082	0.0543
	Java	-	-	1.527

Table 6: Comparison results

4.6.2 Conclusion

The optimized version has reduced the construction time across all graphs by a significant amount. The construction times decrease by approximately 93% for Flevoland, 95% for Zeeland, 95% for Corse and 94% for Guadeloupe. This did not come with a cost in query speed as we expected beforehand. The reason this did not happen is because we expected to end up with significantly more shortcuts, which would have slowed the queries down. We do see however that the C++ implementation was significantly faster than our optimised version when ran on the same hardware. It is hard to compare the run times of different languages with different implementations. Looking at the number of shortcuts we can see that our implementations are relatively close to the C++ implementation. The execution time of the Java implementation sat somewhere in the middle of our standard and optimised implementations.

5 Conclusions

Faster preprocessing speeds are possible, this might come at the cost of the query speed. However in our results the query speed changed very little when the construction time was reduced. We saw very good results by making the search range smaller and by using less heuristics. Our experiment with a bidirectional node extraction turned out to not be faster than the normal node extraction. We did not manage to outperform the c++ implementation, however to make a better comparison changes would need to be made to the implementation which we discuss in Section 5.1. Using C# brings some challenges with it. The performance does not look as good as with C++ and also the priority queue was lacking some functionalities that would have been nice to have. It however does also bring good things with it. The development experience is a lot smoother and its more widely used. We already got positive feedback on our GitHub about our C# implementation. This shows that there is demand for it.

5.1 Further Work

To get a better view on the improvements made we would need to create a similar implementation in C++. Dijkstra's algorithm is what takes the most time. We would need to compare a C# and C++ implementation of Dijkstra's algorithm to know if C++ is really faster than C#. To improve upon the preprocessing speed further we think that machine learning could be used to determine the importance of a node. By training the machine learning model on the values given by the heuristics even faster results could possibly be achieved.

References

[1] Code repository. https://github.com/Valentijn-Stokkermans/ContractionHierarchiesCsharp. (accessed: 29.10.2023).

[2] Contraction hierarchies results. https://github.com/tungduong0708/Contraction-Hierarchy. (accessed: 21.10.2024).

[3] Custom priority queue. https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp. (accessed: 29.10.2023).

[4] geofrabrik. https://download.geofabrik.de/. (accessed: 29.10.2023).

[5] Java contraction hierarchies. https://github.com/navjindervirdee/Advanced-Shortest-Paths-Algorithms/blob/master/Contraction%20Hierarchies/DistPreprocessSmall.java. (accessed: 29.10.2023).

[6] osm4routing2. https://github.com/Tristramg/osm4routing2. (accessed: 29.10.2023).

[7] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959.

[8] Robert Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks, 2008.

[9] Ira Pohl. Bi-directional and heuristic search in path problems, 1969.

[10] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. *13th European Symposium on Algorithms (ESA), volume 3669 of LNCS, pages 568–579.*, 2005.

[11] P. Sanders and D. Schultes. Engineering highway hierarchies. *14th European Symposium on Algorithms (ESA), volume 4168 of LNCS, pages 804–816.*, 2006.

[12] P. Sanders and D. Schultes. Engineering fast route planning algorithms. *6th Workshop on Experimental Algorithms (WEA), volume 4525 of LNCS, pages 23–36.*, 2007.

[13] D. Schultes. *Route Planning in Road Networks.* PhD thesis, Universität Karlsruhe (TH), 2008.