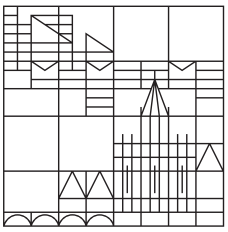
Master Thesis

Query Processing on Dynamic Networks with Customizable Contraction Hierarchies on Neo4j

by
Marius Hahn

at the

Universität Konstanz



Faculty of Sciences
Department of Computer and Information Science

1. Evaluated by	Dr. Theodoros Chondrogiannis
2. Evaluated by	Prof. Dr. Sabine Storandt

Konstanz, 2023

Abstract

The thesis addresses the challenge of speeding up shortest path queries in graph databases, with a focus on Neo4j. It employs an index called Customizable Contraction Hierarchies (CCH), which has been proven to significantly outperform Dijkstra's algorithm, especially in graphs where certain vertices are more important than others in terms of their presence in numerous shortest paths. The research specifically explores the integration of CCH into Neo4j and its performance post multiple edge weight updates. The thesis also investigates the size of the index graph and whether it can be stored entirely in the main memory. It also comments on possibilities to employ further implementations of CCH to other graph databases.

Contents

CHAPTER 1

Introduction

Most applications which handle data have to store these eventually. Simple files do not offer enough semantic structure. Therefore databases have been developed to not only store but query and aggregate data effectively and efficiently. Relational databases that store data in tables have become the unspoken industry standard in the last decades. However there are domains that do not naturally fit into tables and one will need a lot of restructuring to make them fit. As a result, there have been efforts to create databases that use other abstractions to store data, like the graph database Neo4j.

1.1 Background and Motivation

In a relational database the data is organized in tables. Tables have rows and theses rows can be connected to other rows of the same or a different table using joins on matching cells, usually primary- foreign key relationships. To retrieve information that is stored in two table rows one has to scan both an join them on the desired key property. Depending on the size of the tables and the number of joins which have to be done to retrieve the desired information, such queries can be very expensive.
In contrast to relational databases, graph databases treat relationships as first-class citizen. This means a relationship that connects nodes is an entity just as the nodes it connects themselves. As in a graph database, a node has a pointer to its relationship and a relationship has a pointer to its nodes, there is no table scan needed to retrieve connected data, which can make these queries very performant. Such queries are shortest path queries on domains which resembles a graph by their nature, for instance road networks.

1.2 Problem and Objectives

We focus on these shortest path queries and try to make these even quicker by employing an index call Customization Contraction Hierarchies [DSW16], *CCH*. CCH is proven to achieve a tremendous speedup compared to Dijkstra's algorithm in graphs where one can find vertices that are more important than others. Important in this context means that these vertices belong to many shortest paths. One example for such a domain are road networks, which we will also use in our test scenarios.

1.3 Contribution

In comparison to many other papers that examine CCH in their specific details we want to examine if we can adopt it to the graph database Neo4j, and still keep its advantages. Will we be able to store the index in a manner which allows us to keep the performance gain we had

achieved before? Additionally we want to explore how the index behaves after updating multiple edge weights. Will queries still be as quick as they were before the update? This is an essential question as data in databases are usually not only stored but also manipulated. We also want to uncover how big such an index graph will become, as there might be no need to read the index from the disk, because it easily fits in main memory.
Finally we want to keep the integration into Neo4j as small as possible to make it as easy as possible at a later point to port the implementation to other graph databases that might become more relevant in future.

CHAPTER 2

Related Work

The basic algorithm to answer shortest path queries is Dijkstra's algorithm [Dij59]. Section 2.1 gives a basic overview about approaches to make Dijkstra's algorithm quicker. As our research focus is set on databases, we show successful approaches to achieve a fast shortest path query answering with external memory in section 2.2. We are aiming for a suitable index structure for Neo4j shortest path queries, therefore we include a overview over existing shortest path query algorithms in section2.3. Our choice is set Customizable Contraction Hierarchies [DSW16] therefore we also depict the efforts made to integrate Contraction Hierarchies into Neo4j in section 2.4.

2.1 Speed Up Dijkstra's algorithm

Dijkstra's algorithm is from 1959 and in the following decades there have been many approaches to make it quicker. To name some there is A* [HNR68], which adds a heuristic to each node to help Dijkstra's algorithm to go in the right direct. Furthermore ALT [GH05] with stands for *A* + Landmarks + Triangle Inequality*, which is a approach to improve the A* heuristic with landmarks. Additionally the Arc Flag [BDD09] approach which is based on the idea that many shortest paths in a graph usually overlap, so we can store them in a compact way. At a later point edges which are flagged true for the target direction will be expanded by Dijkstra's algorithm. On top of that is the Transit-Node [BFSS07] which makes use of the observation that if you want to go far, nodes which are spatially close usually pass by the same node. One will try to find such nodes and store their connecting shortest paths in a table. If one looks for the way from one node to another node one only starts to local Dijkstra's algorithm for the source and the target side until one finds a transit node. As the distance between the transit nodes are known, we simply can look up the rest of the path. Moreover Contraction Hierarchies [GSSV12] or CH which is the base for what we will examine further. As transit nodes, CH uses the idea that there are nodes which are more important than others, but instead of selecting some important nodes one ranks them all. CCH, what we will implement for Neo4j, is a fashion of CH that makes it easier to react to changes in the CH index structure. CH goes back to the diploma thesis of Geisberger [GSSD08] in 2008. As all previous mentioned techniques, CH and CCH especially speed up long distance queries.
CH deletes nodes and inserts edges to the graph, so called shortcuts, which preserve the shortest path property in case a node which is deleted resides on a shortest path between the remaining ones. When querying a shortest path, CH uses a modified bidirectional-dijkstra which is restricted to only expand nodes that are of higher importance. This method is able to retrieve shortest paths of vertices that have a high spacial distance. However, it is rather static. In case a new edge is added or an edge weight is updated, it might be necessary to recontract the whole graph to preserve the shortest path property.
In 2016 Customization Contraction Hierarchies [DSW16] or CCH was published. The approach is the same, but in CCH shortcuts are not only added if the contraction violates the shortest path property. Shortcuts are added if there had been a connection between its neighbors through the

just contracted vertex and these neighbors do not own a direct connection through an already existing edge. The shortcut weights are later on calculated through the lowers triangle 4.1.1. Additionally Customization Contraction Hierarchies [DSW16] provides an update approach which only updates edges affected by a weight change.

2.2 Shortest path query processing with external memory

Most shortest path external memory approaches have focused on Hub-Labels like "HLDB: location-based services in databases [ADF$^+$12]" and "COLD. Revisiting Hub Labels on the database for large-scale graphs [EEP15]", with a few notable exceptions like "Mobile Route Planning [SSV]" and "The case against specialized graph analytics engines" [FRP15]. Hub-Labels tries to find important vertices that are encoded in many shortest path, so called "hubs". After that it precalculates a two hop cover for all vertices in the graph. For every source target vertex pair s, t there must be at least one common hub $\forall s, t \epsilon V \exists h \epsilon H(s) \cap H(t)$ [ADGW11] and saves this information to a so called hitting set H. Each vertex has its own hitting set. For query answering on looks in the hitting set of the source and the target vertex. There must be at least one common hub. With adding the distances that are stored in the two hitting sets on gets the distance from s to t. By recursively looking at the neighbors of the target vertex one can finally resolve the original path. This method is faster than CH due to [ADF$^+$12] but the space consumption is higher as well as the preprocessing time. HLDB [ADF$^+$12] uses a relational database to store and query the hitting set of vertices.
Finally there is Mobile Route Planning [SSV] by Peter Sanders, Dominik Schultes, and Christian Vetter. In this paper it is described how one can efficiently store a CH index structure on a hard drive. It includes an interesting technique to how to store edges which are likely to be read sequentially spatially close on the hard drive. As a result read operations that have to be done during query time are efficiently. The motivation of Mobile Route Planning [SSV] through was slightly different. They came up with this idea because computation power on mobile devices is limited, therefore they wanted to find a solution to precalculate the CH index on a server and later on distribute it to a mobile device.
We will use parts of this idea and partly port it to our database context as we suppose there to be many similarities.

2.3 Existing shortest path algorithms in Neo4j

Neo4j contains an implementation of Dijkstra's algorithm , a bidirectional Dijkstra and A*. They are provided by the traversal API such that every plugin can use it. Unfortunately they only provide a *one to one* dijkstra, which made it useless for our purpose. The tests that prove our dijkstra implementation as stated in 3.3 use the Neo4j bidirectional dijkstra and compare the path's weights. By that we also measure the time both need to find the shortest path. It shows our unidirectional dijkstra is slightly faster than the one of Neo4j.
Furthermore the Neo4j graph data science library which provides a *one to all* and *one to one* dijkstra implementation. Moreover it contains an A* implementation, a *Yen's Shortest Path* algorithm and a *Delta-Stepping Single-Source Shortest Path* algorithm. We did not use these either as it turned out that having our own dijkstra is key to accomplish the implementation we are aiming for.

2.4 Contraction Hierarchies in Neo4j

The bachelor thesis by Nicolai D'Effremo [D'E19] which has implemented a version on Contraction Hierarchies [GSSV12] for Neo4j. This implementation shows that even for databases CH is an index structure worth pursuing, as there was a tremendous speedup of shortest path queries paired with a reasonable preprocessing time. Anton Zickenberg [Zic21] showed in his bachelor thesis that it is even possible to restrict these queries with label constraints. Although CH and CCH have only slightly differ, unfortunately we could not use much of the code provided by them. It was deeply integrated into the Neo4j-Platform. Additionally since two major release updates have happened which meant breaking changes have made it nearly impossible to reuse any of this code.

CHAPTER 3

Preliminaries

As the target platform for this work is the graph to database Neo4j, we will mostly consider *directed* graphs. From the terminology we always refer *arcs*, which is an directed edge. In some cases we will refer to *edges*, in these cases the direction irrelevant.

3.1 Notation and Expressions

We denote a graph $G(V, A)$ in case we mean a *directed* graph, where v is a vertex contained in the vertices $v \epsilon V$ and a is an arc $a \epsilon A$. An arc is uniquely defined by vertices v_a and v_b such that $v_a \neq v_b$, so there are neither loops nor multi edges. An arc has a weight function $w : A \to \mathbb{R}_{<0}$ that returns a positive integer.
We use A as the arc set and a for a single directed arc. $a \epsilon A$ can be replace with $e \epsilon E$ which refers to edges that are *undirected*.
G represents the input graph. The index graph $G'(V', A')$ is the graph which will be used at contraction for initially building the CCH index structure. A vertex v in will never be actually deleted. Instead the rank property is set to mark this vertex as contracted. So $V \equiv V'$ but $A \subseteq A'$ as arcs are added during the contraction. $S = A' \setminus A$ is the shortcut set that is added.
We define the set of neighbors as $N(v)$, as vertices which are connected to v with an arc. If we only want to get the neighbors of higher rank we will use $N_\uparrow(v)$. Analogously $N_\downarrow(v)$ defines the neighborhood of vertex v which is of lower rank.
We define the degree of a vertex v as the number of neighbors this vertex has $|N(v)|$. The overall degree is the sum of a all upwards and downwards neighbors $|N(v)| = N_\uparrow(v) + N_\downarrow(v)$.
It exists a function on each graph $v : G \to \mathbb{N}_0$ that receives a positive integer and returns the vertex which has this rank assigned.
$G^*(V^*, A^*)$ is the search graph when calculating a shortest path query. Futhermore one query will have two search graphs. $G^*_\uparrow$ representing the upwards search graph and the $G^*_\downarrow$ representing the downwards search graph.
Finally there will be the edge set of edges that are written onto the disk. These $\bigcirc A$ will be separated into sets $\bigcirc A_\downarrow$ and $\bigcirc A_\uparrow$, too.

3.2 Priority Queue

One data structure that is heavily used in this paper is a priority queue. This a very useful structure as it always returns the element with the lowest priority. In Dijkstra's algorithm for example, the priority is calculated on the path weight, thus retrieving an DijkstraState one will always get smallest remaining. A problem arises when updating an element of a priority queue that is already in the queue, because a priority queue can contain the same element multiple times, even with different priorities. The following explanation refers to the Java 17 reference [aiaOPRSCU23], but priority queues which are based on a binary heap [Flo64] all have the same properties.

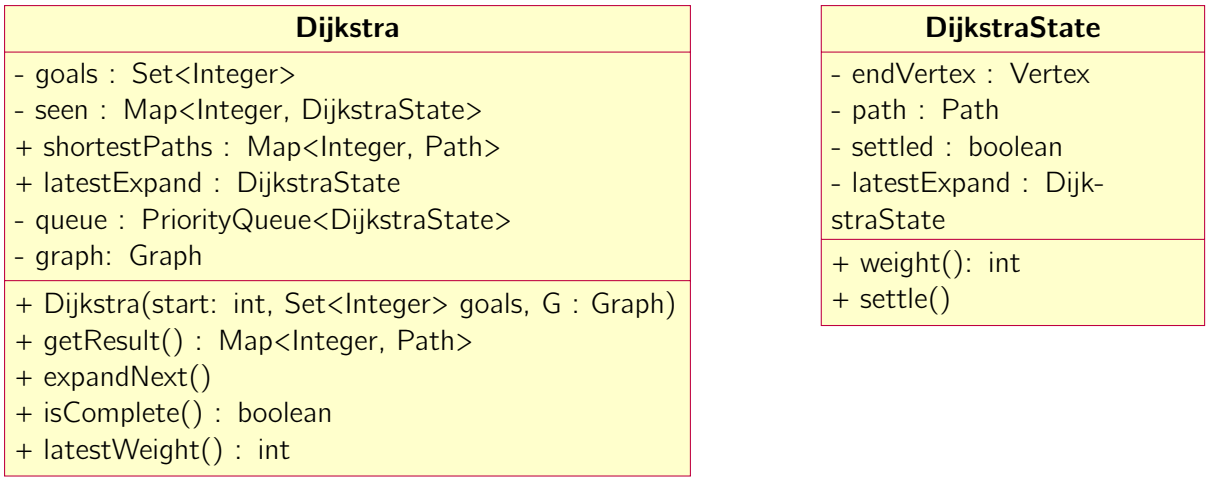


Figure 1 Dijkstra Class Diagram

One might ask why it is impossible remove the element that is already in the queue and then re-push it. That is possible but slow, as the operations *contains(element)*, and *remove(element)* are running in linear time $\mathcal{O}(n)$. It would be better to use *offer(), poll(), remove()* or *add()* which run in logarithmic time $\mathcal{O}(\log(n))$. One possible could be to keep a reference to the element and change the priority as needed at a later point. But the queue will not be notified by such a manipulation, and because the queue eagerly keeps the next element to dequeue at the top one will retrieve the wrong element. So we have to come up with something better. It does not cause a problem if the priority of an element only decreases, as by doing so one will always certainly receive the one with the lowest priority. Therefore one can re-push elements to the queue, as one will always remove the smallest element from the head. To avoid processing an element twice one inserts the elements into a set where one keep track of the already processed elements and if one retrieve an already seen element one poll again. If the priority can decrease and increase and one push updated the elements to again, the problem is a bit more difficult as now one can possibly dequeue an element with the old priority which is lower as the current but not valid anymore. We create a priority queue and a hash table whose key is the element and the value is a integer a version number. If we push an element to the queue we check whether the element is already in the control hash table, if not not we insert it together with the value 0. Then we wrap the element together with the 0 and insert it into the queue. If the element is already in the control hash table we increase its version number in the hash table by one, wrap this new version's number together with the element and push it into the queue. At poll we retrieve the element together with its current version number. If the number is not equal to the one in the control hash table we poll again and check until we find an element that is up to date. As a result we can keep $\mathcal{O}(\log(n))$, because *put(key, value)* and get(key) run in constant time in hash table given there is a suitable hash function.

3.3 Dijkstra' algorithm

Here, we want to explain Dijkstra's algorithm as it is crucial for the ch/cch search. Additionally we want to decouple two things which are usually done together. The initialization of a dijkstra query and the iteration step, which expands vertices until the shortest path is found.
Dijkstra finds the shortest path from a source vertex to a target vertex by always using the *best* path. *Best* means the path that has the shortest distance to the start. Dijkstra starts with expanding the neighborhood of the source vertex and will move on to the vertex that has the smallest total distance to the source vertex. This is done until dijkstra is about to expand the

target node int the next step. In the process of finding the target, dijkstra also finds the shortest path to every node it expands. Therefore dijkstra cannot only be used to do *one to one* shortest path queries, but also for *one to many* and *one to all* shortest path queries.
Looking at Figure 1 we initialize the query with a start ID, a set of target IDs and the graph on which we will use for to do the query. During the construction process the query will take the start ID, create an object of the type *DijkstraState* and push it to the *queue*. This priority queue is organized by the path weight of *DijkstraState* such that one always polls the shortest path that is inside the queue.
After the initialization, we can start expanding nodes by using the *expandNext()* method, which we provided in Algorithm 1. Firstly we poll the next state from the queue. Then we check whether the endVertex of the just retrieved state is in set of targets we are looking for or the set of goals is empty; meaning we will do a *one to all* search. If so we add the just found shortest path to the *shortestPaths* and settle it as we know that we found the shortest path for that endVertex. We iterate over the arcs which are attached to this endVertex and therefore get its neighbors. Afterwards for each neighbor we check if we have to update it with a new state in the queue. This is the case if either we have not seen this vertex so far or the state in the queue is of higher weight as the one we just found.

Algorithm 1 Dijkstra Algorithm

```
1:  procedure expandNext()
2:      state ← queue.poll()
3:      latestExpand ← state
4:      if state.endVertex() ∈ goals ∨ goals = ∅ then
5:          shortestPaths.put(state.endVertex(), state.getPath())
6:      end if
7:      state.settle()
8:      for  arc in state.getEndVertex().arcs do
9:          neighbor ← arc.otherVertex(state.getEndVertex())
10:         if mustUpdateNeighborState(state, neighbor, arc.weight) then
11:             newState ← state.getPath() + arc
12:             queue.update(State(neighbor, newState))
13:             seen.put(neighbor, newState)
14:         end if
15:     end for
16: end procedure
17: function isComplete(forwardQuery, backwardQuery, best)
18:     return queue = ∅ ∨ |shortestPaths| = |goals| ∧ goals ≠ ∅
19: end function
20: function mustUpdateNeighborState(state, neighbor, cost)
21:     nInSeen ← seen[neighbor]
22:     return neighbor ∉ seen ∨ ¬(nInSeen.settled ∨ nInSeen < state.weight + cost)
23: end function
```

The *expandNext()* procedure is usually done in a while loop until the *isComplete()* function returns *true* and all requested shortest paths are found or all vertices have been expanded. We provide this with the function *getResult()* on a dijkstra query object as one can see in Figure 1. Being able to call the *expandNext()* procedure from outside is very powerful. For example if we want to write a bidirectional dijkstra, we can create two instances of the class *Dijkstra* as stated in Figure 1. One dijkstra is initialized only with outgoing Dijkstra($s, [t,], \overrightarrow{G}(V, \overrightarrow{A})$) arcs the other only with incoming arcs Dijkstra($t, [s], \overleftarrow{G}(V, \overleftarrow{A})$). This framework we can use to build our algorithm which tells the bidirectional query when to stop or which side to expand next. The essential finding is that we have reused the logic of the *expandNext()* procedure and not written

it again.

3.4 Contraction Hierarchies

Contraction Hierarchies [SSV] is a method which first augments the graph with some additional edges that later on help Dijkstra's algorithm, such that it needs to expand fewer vertices.

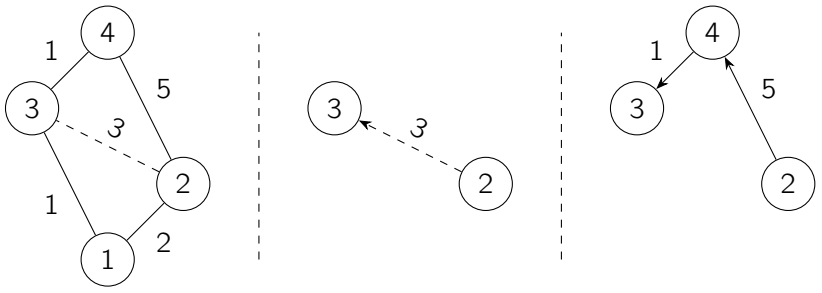


Figure 2 Contraction Hierarchies simple example

Reading Figure 2 Contraction Hierarchies contracts or deletes vertices in a given order. The order we take is defined by the number one sees inside the vertices: CH contracts it in this order $v(1)$, $v(2)$, $v(3)$ and finally $v(4)$. Only the solid lines are edges of the original graph. The dashed line is a shortcut. While contracting vertices it preserves the shortest path among the remaining vertices. At first $v(1)$ is contracted and as $v(1)$ is on the shortest path between $v(2)$ and $v(3)$ a shortcut is inserted between them. This is the only shortcut CH inserts in this small graph. In a shortest path dijkstra will now be restricted to only expand edges which are of higher rank. To include all possible path CH does this from the source vertex side expanding all outgoing edges and from the target side with all ingoing edges. If we want to find the shortest path from $v(2)$ to $v(3)$, CH will expand all outgoing of $v(2)$ where it finds $v(3)$. It also expands all ingoing of the target vertex $v(3)$ where it finds $v(4)$. Then both search side are merged at the meeting point which is in this case $v(4)$. This results in the path on the right in Figure 2. The upwards search also find the a direct path from $v(2)$ to $v(3)$, which one can see in the middle of Figure 2. Out of all these paths one take the shortest, which is also the shortest path in the original graph. A shortcut does not only contain a weight information but also which vertex was contracted as it was inserted. With this information one can resolve the actual path in the graph.

CHAPTER 4

Customizable Contraction Hierarchies

In this section we will present the basic idea of Customization Contraction Hierarchies [DSW16] and also work out the main difference between CCH and Contraction Hierarchies [GSSV12]. We will provide examples to help understand concepts which are crucial to comprehend our implementation.

4.1 Contracting

Algorithm 2 provides our contraction algorithm. We do a called *metric dependent* contraction in Customization Contraction Hierarchies [DSW16]. This is a greedy algorithm which always takes the next best vertex to contract. Some use the simple edge difference as Contraction Hierarchies [GSSV12], but we will use a more advanced technique which assigns an importance to each vertex. This importance is further described in section 4.5.2, through the equation 4.1.
Before starting we have copied G into G' such that both are identical but refer to their own arc and vertex set. The input parameter of the *contractGraph()* function is the set of vertices V into the index graph G'. At first we calculate the so called *contraction* for each vertex, which is the set of shortcut arcs which has to be inserted if that vertex is contracted next. From this contraction object we can determine the importance of the vertex. We push the vertex together with its importance into the queue. The queue is a priority queue which is organized by the importance, such that *Q.poll()* will always return the vertex with the lowest importance in the queue. As long as the queue is not empty we pull the next vertex and calculate its contraction. We assign the rank on the current vertex, initially starting at 0, and add this information to G which resides in Neo4j and increase the rank for the next vertex that will be pulled from the queue.
Then we iterate over the shortcut set of the contraction and insert a shortcut into G' where there is yet no arc between vertices. If there is already an arc which connects the vertices of a shortcut, we update the weight of that arc. If the shortcut weight is smaller we update the middle vertex to be able to reconstruct the actual path in the input graph G. In comparison to [DSW16] we merge two steps in one. The first is adding shortcuts and the sections is basic customization, through lower triangle enumeration 4.1.1. This is needed because at this point we already have the lower triangles at hand, this we do not need to iterate over them again to set the correct weight and middle vertex of the arcs. Then we iterate over the neighbors of the recently contracted vertex $N_{\downarrow}(v) \cup N_{\uparrow}(v)$, recalculate the contraction of these vertices and repush their importance together with the vertex back into the queue. This is needed because

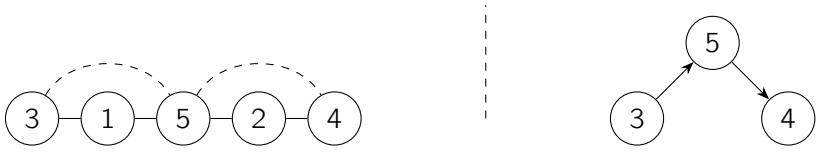


Figure 3 The numbers inside the vertices represent their contraction order. Shortcuts are dashed edges

Algorithm 2 Insert Shortcuts Algorithm

```
1: function contractGraph(V)
2:     for v ∈ V do
3:         Q.offer(getContraction(v))
4:     end for
5:     while Q ≠ ∅ do
6:         contraction ← getContraction(Q.poll())
7:         v ← contraction.v
8:         v.rank ← rank
9:         updateNodeInNeo4J(vertex, rank++)
10:        for shortcut ∈ contraction.shortcuts do
11:            createOrUpdateEdge(vertexToContract, shortcut)
12:        end for
13:        for neighbor ∈ N↓(v) ∩ N↑(v) do
14:            Q.update(getContraction(neighbor))
15:        end for
16:    end while
17:    return v
18: end function
19: function getContraction(v)
20:    shortcuts ← []; outerCount, innerCountTimesOuter ← 0
21:    for inArc ∈ v.inArcs do
22:        if inArc.start.rank = Vertex.UNSET then
23:            outerCount++; inNode ← inArc.start
24:            for outArc ∈ v.outArcs do
25:                if outArc.end.rank = Vertex.UNSET then
26:                    innerCountTimesOuter++; outNode ← outArc.end
27:                    if inNode ≠ outNode then shortcuts.add(Shortcut(inArc, outArc))
28:                    end if
29:                end if
30:            end for
31:        end if
32:    end for
33:    ED ← outerCount = 0 ? 0 : |shortcuts| − outerCount − innerCountTimesOuter / outerCount
34:    return Contraction(v, ED, shortcuts)
35: end function
```

the importance of a vertex depends on its neighbors. As the neighbors of v just lost a neighbor their importance has to have changed. After the queue is empty and the algorithm is finished, the function return the top vertex, which is the one with the highest rank. This vertex is later on needed to store the index onto the disk.
To get a contraction of a vertex we have to iterate over all connected vertices of v which are not yet contracted. We initialize a collection of shortcuts to an empty list. Afterwards we iterate over all incoming arcs of v, where the start vertex has not yet been contracted. For each of these incoming arcs we iterate over all outgoing arcs of which the end vertex has not yet been contracted. Then we create a shortcut container object which is inserted into the shortcut collection. Into the shortcut we insert the incoming and the outgoing arcs.
Finally we calculate the edge difference and return the vertex together with the edge difference and the shortcuts, in a *Contraction* object.

4.1.1 Lower Triangles

To determine an arc weight of G' we need to look at its lower triangle as shown in Figure 4. This is the two loops in *getContraction(v)* in Algorithm 2. They enumerate all lower triangles of the vertex z, which has a lower rank than x and y, $r(z) < r(x) < r(y)$. The shortcut which is created, contains the arcs b and c. The weight of a in the G' is determined by the shortest lower triangle or by the weight of the input graph.
The lower triangles are important for the basic customization process but also needed for the update process. We are interested in the arc a. As shown in Figure 4 it follows:

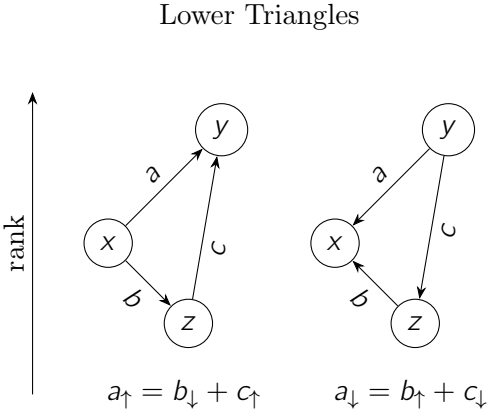


Figure 4 Lower Triangle

$$a_\uparrow = b_\downarrow + c_\uparrow \qquad a_\downarrow = b_\uparrow + c_\downarrow$$
$$a(x,y) = a(x,z) + a(z,y) \quad a(y,x) = a(z,x) + a(y,z)$$

4.1.2 Contraction Example

In Figure 3 you can see a contracted graph $G'(V, E')$ on the left. The solid lines represent the original edges E of a graph G. The dashed lines between vertices are shortcuts S which have been added while creating the CCH index graph $G'(V, E')$. The numbers inside the vertices reflect the contraction order. Contracting a vertex means deleting it. If a vertex which is contracted resides on a simple path between two vertices of higher rank, and there is no edge $e \epsilon E'$ between these vertices a shortcut has to be inserted. We will now reconstruct the contraction of Figure 3. At first vertex $v(1)$ is removed. As $v(1)$ resides on a simple path between $v(3)$ and $v(5)$ and there is no edge $e(v(3), v(5)) \notin E'$. Hence a shortcut is added to keep the connection. The same applies after contracting $v(2)$ for the vertices $v(4)$ and $v(5)$. For all the other vertices we do not need to insert shortcuts.

Algorithm 3 Find Search Path

```
1:  function find(start, goal)
2:      pickForward ← true;
3:      forwardQuery ← Dijksta(start, [goal], G*↑(V, ⋃A↑));
4:      backwardQuery ← Dijkstra(goal, [start], G*↓(V, ⋃A↓));
5:      while ¬isComplete(forwardQuery, backwardQuery, candidates.peek()) do
6:          query ← pickForward?forwardQuery : backwardQuery
7:          other ← pickForward?backwardQuery : forwardQuery
8:          pickForward ← ¬pickForward
9:          if ¬reachedTop(query) then query.expandNext()
10:         else continue
11:         end if
12:         latest ← query.latestExpand()
13:         if other.resultMap().containsKey(latest.rank) then
14:             forwardPath ← forwardQuery.getPath(latest.rank)
15:             backwardPath ← backwardQuery.getPath(latest.rank)
16:             candidates.offer(forwardPath + backwardPath)
17:         end if
18:     end while
19:     return candidates.poll()
20: end function
21: function isComplete(forwardQuery, backwardQuery, best)
22:     reachedTop ← reachedTop(forwardQuery) ∧ reachedTop(backwardQuery)
23:     return reachedTop ∨ w(best) < w(forwardQuery) ∧ w(best) < w(backwardQuery)
24: end function
```

4.2 Searching

Algorithm 3 depicts our search algorithm. It finds the shortest path between two vertices. As input parameter is takes two integer values, which represent the rank of the start vertex and the rank of the target vertices. At first we create a boolean variable which helps to choose whether to continue with the forward or with the backward search. Then we initialize one forward query which receives the start vertex as input and all upwards arcs and one backward query that receives the target vertex as input and all downward arcs. We continue until we find the shortest path. We definitely have found the shortest path if either both queries have expanded the top vertex with the highest rank or the next vertex to expand to in both queries is further than the shortest path merge we have seen so far. This is the functionality of *isComplete* function. All shortest path pairs will be merged and pushed to the priority queue which is called *candidates*. It is organized by the path weight and will return the shortest path merge that has been found on *candidates.peek()*. If the search is complete we simply peek at the head of *candidates* and return it as the shortest path; or none if the vertices are not connected.
If the search is not complete we continue. If *pickForward* is set to *true* we will continue expanding the upward forward query otherwise we will expand the backward query. Afterwards we flip *pickForward*, such that in the next iteration the respective other query will be expanded. If the query we are about to expand already reached the top vertex we continue with next iteration step, otherwise we tell the query to expand the next vertex. If the vertex which has been expanded last in the query also appears in the set of already expanded vertices in the other query, we merge both their paths and add them to the priority query *candidates* of the shortest path pairs found so far. As two merged shortest paths do not necessarily result in a shortest path, we still have to continue as described before.

4.2.1 Example

Regarding Figure 3, as we preserved all shortest paths during the contraction the shortest path can be retrieved by a bidirectional Dijkstra which is restricted such that it only expands vertices of higher rank. Therefore if one wants to retrieve the shortest path between $v(3)$ and $v(4)$ there will be a forward search from $v(3)$ and a backward search from $v(4)$. As we restrict theses searches to expand only vertices of higher rank, the only vertices to expand are the start and target vertex. Both will find only one vertex $v(5)$, the highest vertex and also the meeting point. Finding at least one meeting point in the forward and backward search means there exist a path between them. After merging these paths at the middle vertex $v(5)$ one will obtain the shortest path.
For an arbitrary contracted graph it is possible that there is more than one meeting point. As merging two shortest paths will not necessary lead to an other shortest path, one has to merge all possible meeting points and take the path among the merged ones which has the smallest distance.
In the example of Figure 3, backward and forward search both reach the top vertex, so the search can stop.

4.3 Difference between CH and CCH

The left graph in Figure 5 has been contracted in the CH way, whereas the right is the CCH way. We explicitly state this here because we have found a paper [OYQ+20] which mixes up these well known names, claiming they do Contraction Hierarchies, CH, while actually doing Customizable Contraction Hierarchies CCH. The main difference is, CH will only insert a shortcut between two vertices if the vertex that is contracted resides on the shortest path between two of its neighbors. When vertex $v(1)$ is contracted there is no shortcut inserted as vertex $v(1)$ is not on the shortest path between $v(3)$ and $v(4)$.
Whereas in the CCH case the edge weights do not play a role while contracting. If a vertex is contracted and there is no direct connection between two of its neighbors, one has to insert a shortcut. This gives the advantage that we can later on easily update edge weights without inserting new shortcut, as all possibly needed shortcuts already exist.

4.3.1 Example

We will complete this example by updating the edge $e(v(2), v(4))$ which currently has the weight of $w(e) = 1$ to $w(e) = 5$. Now the vertex $v(1)$ is on the shortest path between vertex $v(2)$ and $v(3)$.

To update the CH graph we have to insert an edge between vertex $v(2)$ and $v(3)$ whereas the topological structure of the CCH remains the same, one only needs to update the weight and the middle vertex of the already give shortcut edge.

4.4 Metric Dependent Vertex Order

There are two ways to receive a suitable vertex order. A so called *metric independent* and a so called *metric dependent* one. The metric independent recursively looks for minimum balanced separator to determine a vertex ordering[DSW16]. Although this is the superior method, it is not used in this paper. Writing an algorithm that calculates balanced separators is not trivial,

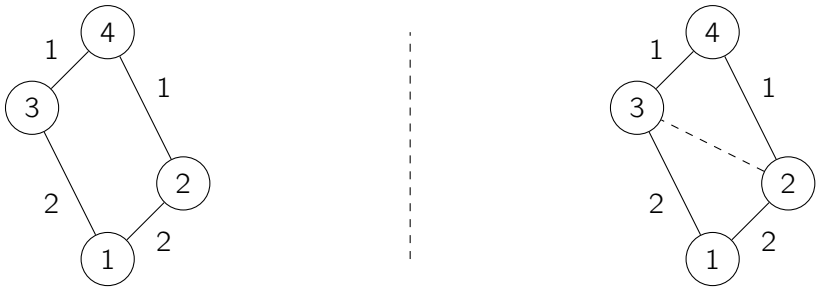


Figure 5 The left represents a CH and the right a CCH contracted graph

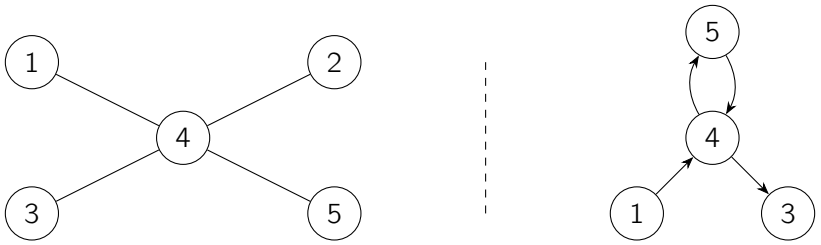


Figure 6 The numbers inside the vertices represent their contraction order

and we are not aiming for optimizing the contraction process.

The *metric dependent* order mainly uses the edge difference ED to determine which vertex is to be contracted next. The ED is determined as the $|edgesToInsert| - |edgesToRemove|$. The fewer edges are inserted during contraction the fewer edges will be contained by the final graph, therefore fewer edges will have to be expanded during the search. However using only the edge differences does not lead to desired result. This is because during contraction there will be areas which becomes sparser than others.

There are two problems which can arise. One is that important vertices are not contracted last. The other is the search space of the query becomes linear although it could be logarithmic.

4.4.1 Important Vertices not contracted last

Figure 6, this is a possible contraction order, if the ED is to contract vertices. At the beginning the vertices with rank 1, 2, 3, 5 have the same edge difference, which is $ED = -1$. Vertex after vertex is removed and no shortcut is inserted. This happens until there are only $v(4)$ and $v(5)$ left. Now $v(4)$ has also an $ED = -1$, similar as $v(5)$. Thus chance is $\frac{1}{2}$ the algorithm contracts $v(4)$ before $v(5)$. This is not the desired result. There are six
$e(v(1), v(2)), e(v(1), v(3)), e(v(1), v(5)), e(v(2), v(3)), e(v(2), v(5)), e(v(3), v(5))$ shortest paths that involve $v(4)$. All the other vertices do not encode any shortest path, therefore $v(4)$ should be contracted last. The search graph on the right of Figure 6 shows the reason. Imagine we we do a shortest path query between $v(1)$ and $v(3)$. After expanding both, the forward and the backward search to $v(4)$, there is yet another vertex we will have to expand $v(5)$. Although as visible in the original graph on the right, it is not possible that $v(5)$ is on the shortest path. Therefore a better contraction order would be contracting $v(5)$ before $v(4)$. This issue can be overcome by the method that is explained in section 4.5.

4.4.2 Linear Query Search Space

Figure 7 shows three possible index graphs G' of one and the same base graph G. The numbers inside the vertices represent the contraction order.

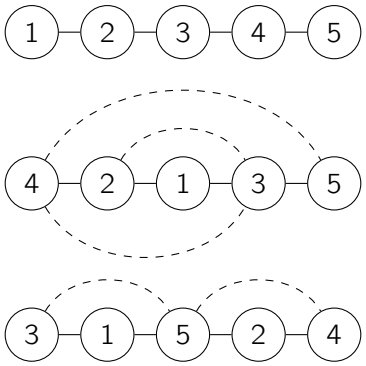


Figure 7 Linear Contraction

The first one could be contracted using the edge difference ED, as always one of the outer vertices with $ED = -1$ was contracted. On the one hand it reaches the optimum in case for *least shortcuts inserted*. On the other hand it has the worst search space among the three vertex orderings. To get from vertex $v(1)$ to $v(5)$ we have to expand four vertices.
The second G' is contracted from the inside to the outside, which encodes the shortest path, first and therefore inserts three shortcuts. Although this example has a lot of shortcuts, there are still a lot of vertices to expand in some cases. In case every vertex of G has a weight of 1, and one wants to go from $v(1)$ to $v(5)$ the forward search will have to expand three to four vertices as in the upper first example.
The third example contracts the middle vertex last. At first it contracts the vertices right next to the middle vertex. Therefore we have to insert shortcuts between $e(v(3)v(5))$ and $e(v(4), v(5))$. Thus no matter what source-target-pair we are trying to find in this example, the forward and the backward search will have to expand at no more than one single vertex each.

4.5 Vertex Importance

As discussed in section 4.4.1 and 4.4.2 that are vertices that are more important than other vertices. Contracting these vertices late is key to results in a efficient search lateron.

4.5.1 Suitability of CCH

As it is important to contract important vertices last, the advantage one gets in doing a CCH search over a simple dijkstra run depends on whether the base graph $G(V, E)$ has vertices which are more important than others.

A vertex $v \epsilon E$ is important if there are many shortest paths containing this very vertex. Therefore if it is possible to calculate a small balanced separator on G, CCH will be able to show its whole advantage. To dive deeper into this topic, see "Lower Bounds and Approximation Algorithms for Search Space Sizes in Contraction Hierarchies" [BS20].

4.5.2 Metric dependent Importance

As shown in section 6 and 7, taking only the edge difference ED into account does not necessarily lead to a proper order. We decided to take the vertex importance calculation as proposed by Customization Contraction Hierarchies [DSW16]. To every vertex we add the level property $l(v)$ which is initially set to 0. If a neighbor $w \epsilon N(v)$ is contracted, the level is set to $l(v) =$

$max\{l(v)+1, l(w)\}$. For every arc $a \epsilon A'$ we add a hop length to the arc $h(a)$. The hop length equal the number of arcs. This arc represents when fully unpacked. Additionally, we denote as $A(v)$ the set of inserted arcs after the contraction of v and $D(v)$ the set of removed arcs. We calculate the importance $i(v)$ as follows:

$$i(v) = l(v) + \frac{|A(v)|}{|D(v)|} + \frac{\sum_{a \epsilon A(v)} h(a)}{\sum_{a \epsilon D(v)} h(a)} \tag{4.1}$$

Our tests show that this importance calculation results in a small increase regarding in the amount of shortcuts added, but the maximum vertex degree is smaller. Which speeds up the contraction process towards the end. Additionally the average search time decreases as the search space decreases.

4.6 Update CCH

Algorithm 4 Update

```
1:  procedure update()(G)
2:      Q ← G'.updatedEdges(G);
3:      while Q ≠ ∅ do
4:          a ← Q.poll();
5:          oldWeight ← w(a)
6:          newWeight ← determineNewWeight(a)
7:          if oldWeight ≠ newWeight then
8:              w(a) ← newWeight
9:              checkTriangles(Q, oldWeight, upperTriangles(a))
10:             checkTriangles(Q, oldWeight, intermediateTriangles(a))
11:         end if
12:     end while
13: end procedure
14: procedure checkTriangles(Q, oldWeight, triangles)
15:     for all triangle in triangles do
16:         if triangles.c() == triangle.b() + oldWeight then
17:             Q.push(triangle.c())
18:         end if
19:     end for
20:     return triangles
21: end procedure
```

The biggest advantage of CCH over CH is that it is easy to update without the need of changing the topological structure of the index graph. This is the reason why CCH can be interesting for graph databases. If an arcs $w(a(x,y))$ weight increases or decreases this can result in a weight change on arcs which connect vertices of higher rank than x, y. We determine all arcs of the input graph G that have been changed and push them to a priority queue. The queue always returns the arc $a(x,y)$ with the lowest rank of the start vertex x. If there are multiple it returns the one with the lowest rank of y among the ones with the lowest rank of x. Afterwards we determine the new weight of the arc using the lower triangles. If there is a lower triangle which can be used as a pass through such that the arc weight in G' does not change we only update the middle vertex. If the weight of the arc has changed we assign the new weight to the arc. Then we check all upper triangles, as shown in Figure 8 of the arc $a(x,y)$. If there is an upper arc, denoted by c in Figure 8, that is influenced by this change. If it is influenced by this change we

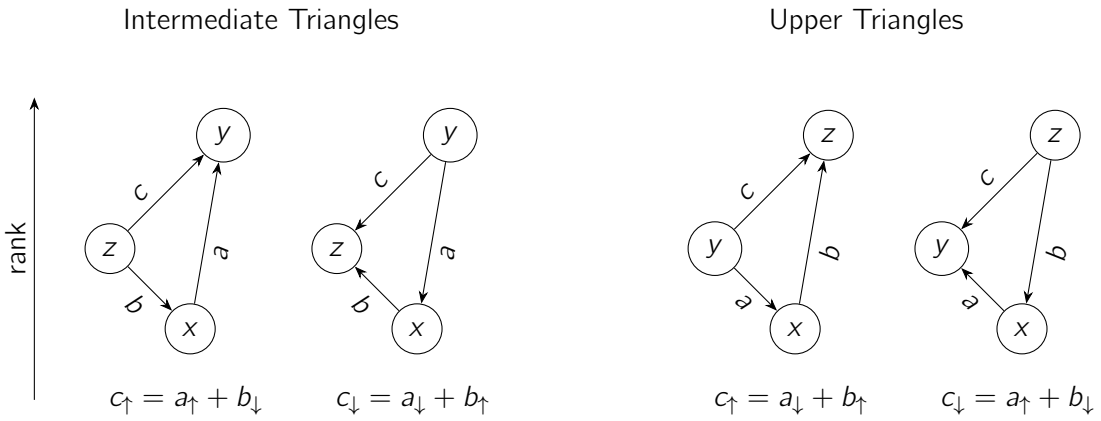


Figure 8 Update Triangles

push it to the priority queue. We do the same with all intermediate triangles.

4.6.1 Intermediate and Upper Triangles

As the explanation of intermediate and upper triangles in Customization Contraction Hierarchies [DSW16] exists, but it is difficult to understand as kept quite short, we will describe it here in detail.

We define an arc as its start and end vertex $a(x, y)$, where the rank of x is smaller as the rank y, $r(x) < r(y)$. For every a we want to construct triangles that involve a vertex z. We assign the letter b to arc between x and z and the letter c to the arc between y and z.

The motivation to construct these triangles is to find the shortest path from y to z, to set the weight of c. Therefore we have to resolve the equation to c

$$c = a + b$$

In the case of upper triangles we want to find a triangle for x and y, such that middle vertex z is of higher rank than both, $r(x) < r(y) < r(z)$. Therefore as shown in Figure 8 results for the upper triangle:

$$\begin{aligned} c_\uparrow &= a_\downarrow + b_\uparrow & \qquad c_\downarrow &= a_\uparrow + b_\downarrow \\ a(y, z) &= a(y, x) + a(x, z) & \qquad a(z, y) &= a(x, y) + a(z, x) \end{aligned}$$

In the case of intermediate triangles we want to find a triangle for x and y, such that middle vertex z is of higher rank than x but of lower rank than y, $r(x) < r(z) < r(y)$. Therefore as shown in Figure 8 results for the upper triangle:

$$\begin{aligned} c_\uparrow &= a_\uparrow + b_\downarrow & \qquad c_\downarrow &= a_\downarrow + b_\uparrow \\ a(y, z) &= a(y, x) + a(x, z) & \qquad a(z, y) &= a(x, y) + a(z, x) \end{aligned}$$

CHAPTER 5

Integration into Neo4j

In this section it is described how Customization Contraction Hierarchies [DSW16] is integrated into Neo4j. CCH augments the input graph, which means it inserts arcs, so called shortcuts, which do not belong to the original data. We want to keep the change to the input graphs as small as possible. We decided not to insert any arc into the graph which is stored inside the Neo4j database, but introduce another graph data structure, the index graph. The mapping between the index and the input graph, which resides in the database, is achieved by the rank property. The rank property is set to the input and the the index graph at contraction time. Through this we achieve a unique mapping between G and G'. This results in another two advantages. Firstly, we gain full control of the graph representation which is helpful to efficiently store and read the index graph for the disk. Secondly, it makes it easier to later on port the idea to another graph database manufacture. Additionally we should mention here that the plugin documentation Neo4j provides is incomplete. The only solution is to reverse engineer their code as it always stops where things become interesting. Presumably a support marketing strategy.

5.1 Index Graph Data Structure

The index graph data structure is neither an adjacency list nor an adjacency matrix. There is a vertex object which has two hash tables. One for incoming arcs and one for outgoing arcs. The hash tables key is of type *Vertex* and the value is of type *Arc*. An arc has a reference to its start vertex and one to its end vertex as shown in Figure 9. This also means we cannot construct them all at once, but need a function which initializes the graph because we have a circular reference. If you want to initialize such a graph, for an ID pair that represents an arc or from a relationship that come from Neo4j, our solution: We iterate over all ID pairs, create all vertices which we have not seen so far an push them to a hash table. Then we create the arc and attach the vertices to it. We receive the vertices from the hash table. Finally we add the arc to its start and end vertex.

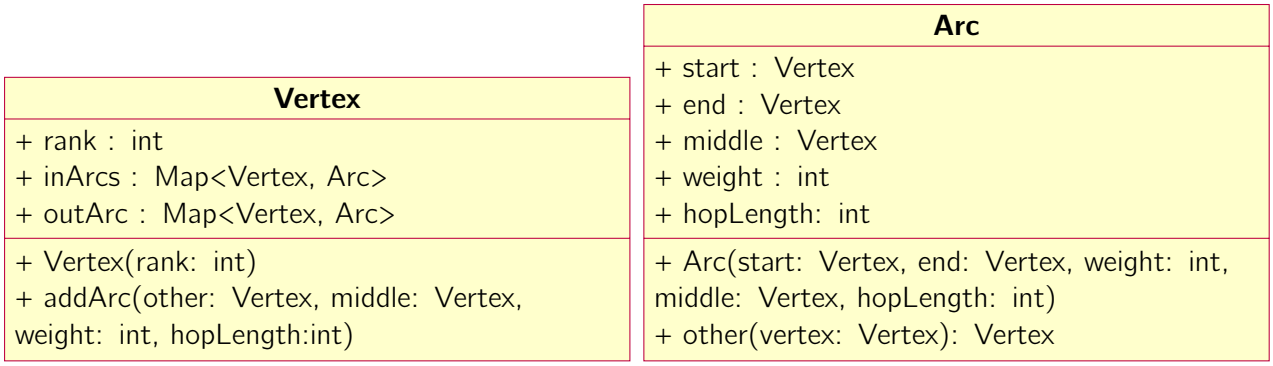
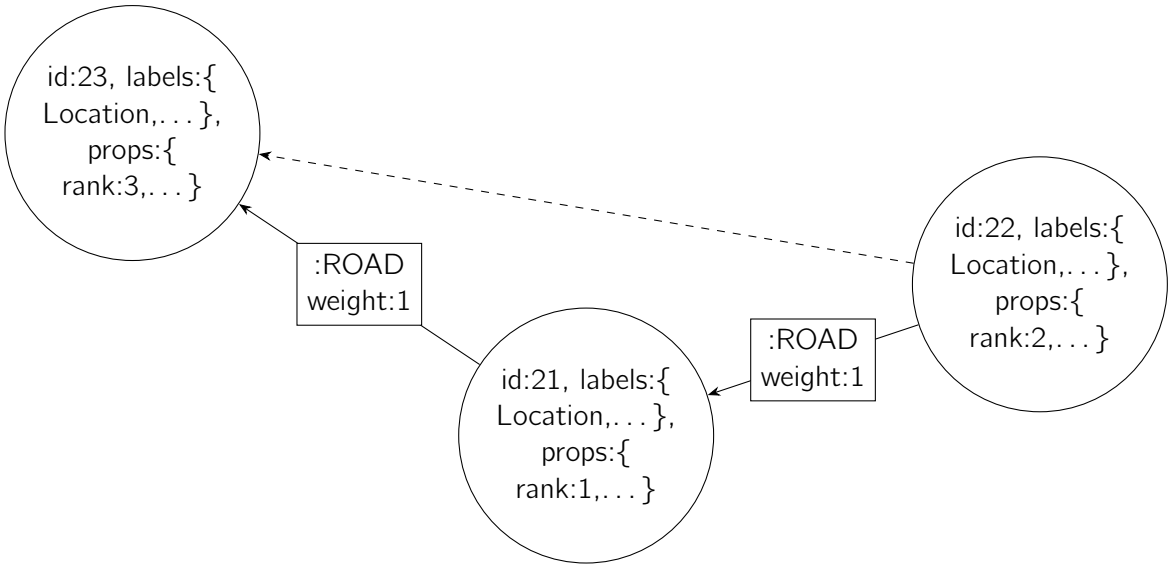


Figure 9 Index Graph



start rank	end rank	middle rank	weight
2	3	1	2
1	3	-1	1
2	1	-1	1

Figure 10 Mapping between Neo4j and the index graph

A disadvantage of this model could be that some modern hardware optimization that exist for arrays do not match with this data structure. Due to paging algorithms as described in "Modern Operating Systems" [AH15] the values of an array are stored sequentially in main memory. When one value of an array is accessed by the CPU, modern hardware reads subsequent values into the CPU-cache because they most probably reside inside the same main me page. The model of the index graph is a linked data structure, a bit like a linked list. The elements of a linked list are contained somewhere in main memory. There is no guarantee that subsequent values have any spacial proximity. However, this makes the the graph traversal easy. Additionally it makes it very efficient to explore the neighborhood of a vertex. There is no array traversal to find a vertex and only one hash table lookup for finding an arc of a vertex. On a small graph, which represents the road network of Oldenburg, shows that CCH queries can be answered in less than one millisecond, which is close to our test results with the original CCH application.

5.2 The Mapping

The in-memory data structure of Neo4j is similar to the explained index graph data structure in section 5.1. A *node* has a collection of *relationships* and a *relationship* has a reference to its *start node* and *end node*. As Neo4j is a full blown property graph, nodes and relationships contain a lot of other information. A node has a collection of *labels*, a relationship has a *type*. The class *Node* and the class *Relationship* are both derived from the class *Entity* which also has a collection of properties as well as an ID that is managed by the database system. Note that, as of version Neo4j 5.X, this ID can change over time and should not be used to make mappings to external systems. Additionally worth mentioning here is that the Neo4j system shifted its ID concept as it moved from major release 4 to 5. Until major release 4 every entity had a unique integer identifier. Since major release 5 every entity has a string identifier which is a UUID and the old *id* identifier is not guaranteed to be unique anymore. It is deprecated and marked for removal.

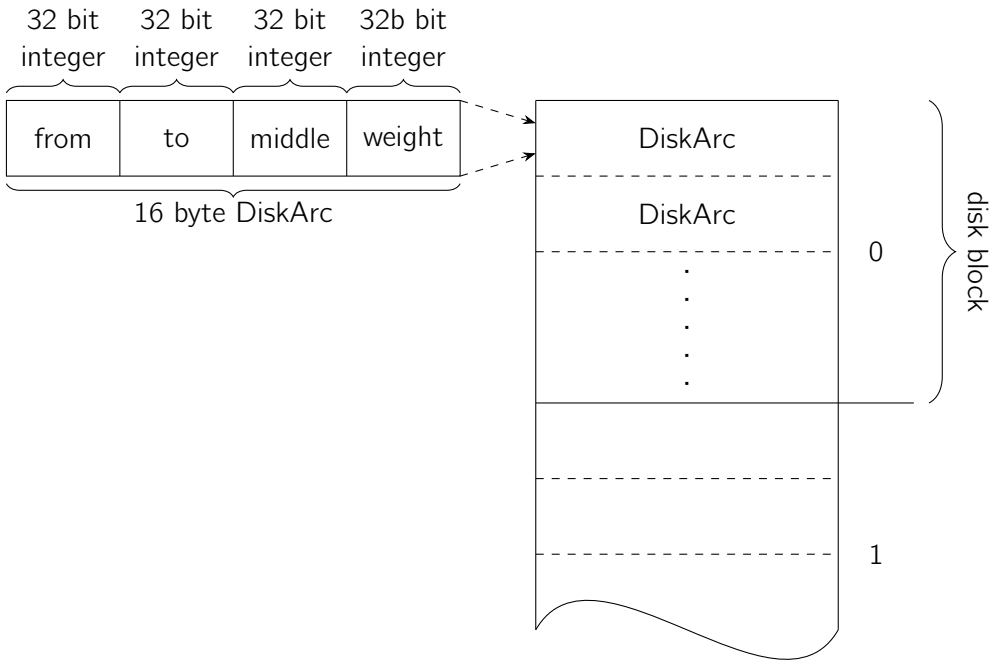


Figure 11 Disk Block

As just explained there is lot of information in this data structure. A lot of information we do not need. Looking at Figure 10 we only want to keep track of the information which is needed for the CCH index. Additionally as disks are divided into blocks and sectors we want to flatten the graph, which in memory more looks like a tree, to a structure that looks like a table. Therefore we decided that the disk data structure only consists of arcs $\bigcirc A$. A DiskArc $a \epsilon \bigcirc A$ consists of four values, the *start rank*, the *end rank*, the *start rank* and the *weight*. The middle node is set -1 in case that this arc is an arc of the input graph. We will get two arc sets $\bigcirc A_{\downarrow}$ for the downwards graph and $\bigcirc A_{\uparrow}$ for the upwards graph. $\bigcirc A_{\downarrow}$ contains all downward arc which are needed for the backward search and $\bigcirc A_{\uparrow}$ contains all upward arcs that are needed for the forward search.
During the contraction every node gets assigned a rank. This rank is the only change that is made to the Neo4j data structure and it is the mapping identifier between the input graph G and the index graph G'. G' will then be used to generate $\bigcirc A_{\downarrow}$ and $\bigcirc A_{\uparrow}$.

5.3 How to Store the Index Graph

After generating the index graph G', we now want to store it as efficiently as possible to the disk. In order to refine the definition of a disk arc. It consist of four $4Byte$ signed integer values *start rank*, the *end rank*, the *middle rank* and the *weight* as you can see in Figure 11.
These 16 Byte disk arcs are collected into disk blocks. The size of a block can be set as parameter but has two lower bounds. At first a disk block should not be smaller as the block size of the file system beneath as this is the smallest unit one can get and it would be a waste of space. Therefore it should also be a multiple of the system disk block size. The second lower bound is the maximum upwards or downarrow degree that exists in the index graph after the contraction, as all upward arcs a vertex have to be stored in the same disk block. This applies for the downward arcs, too.

$$max(d_{\uparrow max}(v), d_{\downarrow max}(v)) \leqslant \frac{diskBlockSize}{16}$$

5.3.1 Persistance Order

Algorithm 5 DFS in StoreFunction

```
procedure store()
    while stack ≠ ∅ do
        if stack.peekFirst().hasNext() then
            vertex ← stack.peekFirst().next()
            if ¬ positionWriter.alreadyWritten(vertex) then
                position ← arcWriter.write(vertex)
                positionWriter.write(vertex, position)
                stack.addFirst(neighbors(vertex))
            end if
        else
            stack.pollFirst()
        end if
    end while
end procedure
```

As the disk arcs are used for the CCH search, we want to sequentially write them in a way which provides a high spatial proximity of vertices that are likely to be requested together. Here we will adopt the idea of Mobile Route Planning [SSV]. In the transformation form G' to its disk arcs $\bigcirc A_{\uparrow}$ and $\bigcirc A_{\downarrow}$ we conduct a simple depth first search on all ingoing arcs on the target rank to determine the order for $\bigcirc A_{\uparrow}$. We provided the algorithm for that in Algorithm 5. As you can see in Figure 13, the *StoreFunction* receives the vertex with the highest rank, whether it shell store the upward or the downwards graph and the path, where to store the arcs onto the disk. For the upwards graph the mode is set to upwards. At initialization we push an iterator over all incoming vertices that reach the top vertex to stack. Also we open a file writer for writing the arc and one file writer for writing the position file as shown in Figure 12. Then we start the *store()* function of Algorithm 5. As long as the stack is not empty we pick the top iterator of the stack, but leave it inside. Afterwards we check if that iterator still holds a vertex. If not we remove the iterator and continue. If it holds one we retrieve it. If that vertex has not yet been written to the disk, we tell the arc writer to store it. The arc writer will return the disk block number at which the arcs of that vertex are stored in the arc file. This position writer will keep track of this information. Then we call the *neighbors()* function which returns an iterator of this vertex's neighbor sorted ascending by their rank. Finally when the algorithm stops, we write the position file onto the disk and flush the rest of the arc file buffer.
We do the same for all the downwards graph using all outgoing instead of the incoming arcs to determine the neighborhood. The arc writer is a write buffer that is as big as the defined disk block size. If during an iteration step there have been more arcs pushed to this arc writer than would fit in the current block, the arc writer flushes its cache to the disk, filling the remaining disk arc slots with four times -1.

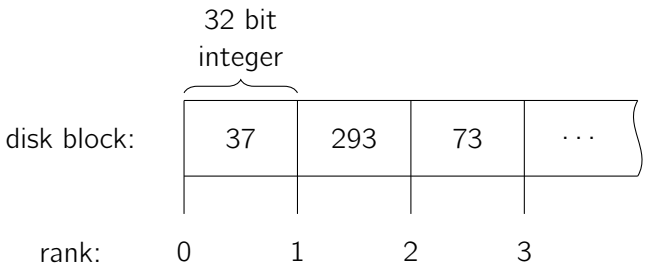


Figure 12 Position File

StoreFunction
- stack : Deque<Iterator<Vertex>> - mode : Mode{UP or DOWN} - positionWriter : PositionWriter - arcWriter : ArcWriter
+ StoreFunction(top: Vertex, mode: Mode, path : Path) + store(vertex: Vertex) - neighbors(vertex: Vertex): Iterator<Vertex>

Figure 13 Class Diagram of the StoreFunction

5.4 Reading Disk Arcs

If one wants to get all upward arcs of rank i one needs to take the upwards position file, retrieve the integer j which is stored at index i, and read the complete block j in the upwards arc file. One will get an array, containing the requested arcs but also some others. These arcs are likely to be requested next. Therefore we want to keep them in main memory and to do so we use buffers. We implemented two buffers a *circular buffer* and a *least recently used buffer* LRU.

5.4.1 Circular Buffer

We implemented a circular buffer to read and cache the DiskArcs at query time. One might also call it FiFo-buffer, which is correct too. The key properties of this buffer are:

- The element that resides inside the longest will be overwritten next if a new element is fetched from the disk. So *first in first out* or FiFo.
- If the maximum number of elements the buffer is reached it starts to overwrite the elements from the beginning. Therefore *circular* or *ring* buffer.
- A element is never *take out*. This means, elements that have been requested and returned will stay inside the buffer until they will be eventually overwritten.

Figure 14 is simple visualization, for a circular buffer of upward arcs. There is the DiskArc array that has already been filled with some arcs, a position hash table that contains the last index at which one will find an arc for the requested rank and a write pointer, which points at the position we have to continue writing when the next cache miss occurs.

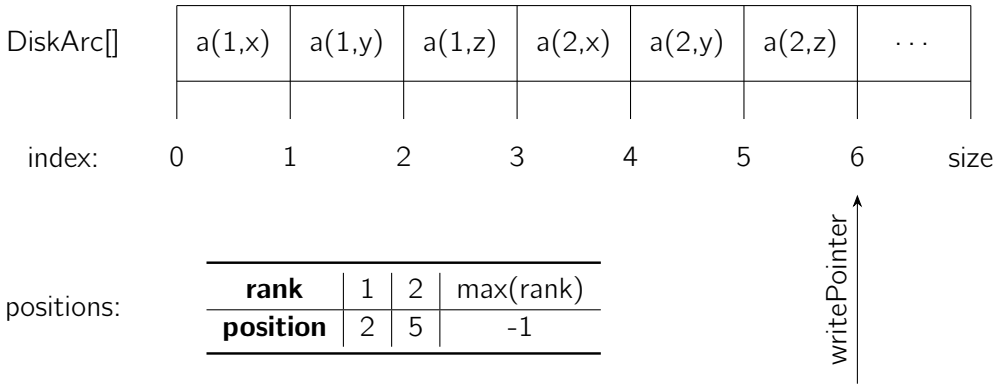


Figure 14 Circular Buffer for upward arcs

If a rank is requested which is already in the buffer, we check its position in the positions hash table and start iterating with decreasing index from the retrieved position until the start vertex of the arc we read has a different rank than the one requested. If we reach the start of the DiskArc query the iteration index is reset to the last index of the DiskArc array, such that we continue reading at the end of the DiskArc array.
If the rank we requested is not in the hash table we request the containing block from the disk. We receive an array of DiskArcs which contains all arcs of the requested rank and maybe some more, which are also likely to be requested soon. At first we insert -1 for the request rank into the hash table. This is because one will not find a DiskArc for each rank. For instance the vertex with the highest rank will not have any outgoing arcs. To prevent having such vertices requested from the disk every time the search touches the top of the graph we have a -1 in the position table and if that rank is requested we simply return an empty collection. We iterate the load DiskArc array. For each DiskArc we insert the start arc rank into the position table together with the current position. The DiskArc inserted into the array at the current position. After each insert we increment the positionWriter by 1. If we reach the end of the buffer array it is set back to 0.
As it is possible that we only partly overwrite some arc sets which are already in the buffer, we receive the start rank of the arc at the *writePointer* position and remove this rank from the positions table after each disk read invocation.
Determining whether a arc set is inside the buffer is not trivial. If it is not in position table, it is not in the buffer. If it is in the positions table it is possible that it has been overwritten. Thus we check whether at position we retrieve from the table there is actually a arc that starts with the requested rank. If yes, the buffer contains the arc set, if no it does not and we remove that rank from the position table.

5.4.2 Least Recently Used Buffer

With the class *java.util.LinkedHashMap* it is very easy to create a LRU-Buffer. It provides the possibility to evicted the entry that has been requested the furthest in the past, if a certain key size has been reached. In our case it maps ranks to sets of disk arcs. We can only determine how many disk arc sets we have in memory and disk arc sets do not always have the same size. Higher rank vertices usually have more arcs as they are of higher degree. The advantage is, it is very easy to implement and therefore very resilient to programming errors.

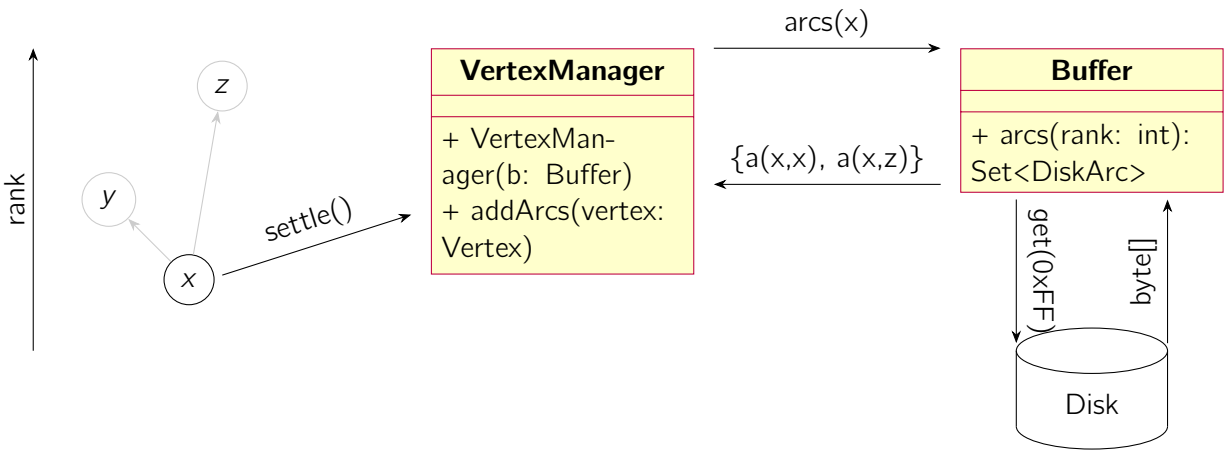


Figure 15 Lazy load vertex at query time

5.5 The Search

The search combines everything explained in this chapter. At the beginning we create two index graphs, the upwards graph $G'_{\uparrow}(V_{\uparrow}, A_{\uparrow})$ and the downwards graph $G'_{\downarrow}(V_{\downarrow}, A_{\downarrow})$. We are looking for the shortest path from the source vertex $v(s)$ to the target vertex $v(t)$. The vertex set $V_{\uparrow}$ of the upwards graph $G'_{\uparrow}(V, A_{\uparrow})$ only contains one vertex $v(s)$ and the upward arc set is empty $A_{\uparrow} = \emptyset$. The vertex set $V_{\downarrow}$ of the downwards graph $G'_{\downarrow}(V_{\downarrow}, A_{\downarrow})$ only contains the $v(t)$ and the arc set is empty $A_{\downarrow} = \emptyset$. Looking at the graph in Figure 15 it visualizes the upwards search size after initialization. The arcs an vertices in grey have not yet been loaded.
As defined in section 3.3, the vertices of the dijkstra query are wrapped in an object called *DijkstraState* which is defined in Figure 1. We now extend this class by one parameter, the *VertexLoader* which you can see in Figure 15. The *VertexLoader* has only one method *addArcs(vertex: Vertex)*. If you pass a vertex to this method the VertexLoader will take that vertex and attach all its arcs and their end vertices to it. This happens when the *expandNext()* method in Algorithm 1 calls *state.settle()*. This follows the observation that the arcs of a vertex in dijkstra are only of interest if the query is about to expand this vertex.
The *VertexManager* gets *DiskArc*'s from a *Buffer*. The buffer can be of any buffer type it only has to implement the *arcs(rank: int):Set<DiskArc>* method. The *VertexManager* is responsible create the arcs and vertices form the *DiskArc*'s it gets. A *Buffer* contains some type of collection data structure caches *DiskArc*'s. When the vertex manger requests some arcs the *Buffer* checks whether they have already been cached. If yes it returns them. If not it will request the arcs from the hard drive. This whole process is also visualized in Figure 15.
After starting the CH-Dijksta as described in Algorithm 3 both $G'_{\uparrow}$ and $G'_{\downarrow}$ are alternatingly expanded and the vertices will be attached when needed. This can be compared to Command & Conquer or other real-time strategy video games where you start at a map that is almost completely grey and the map only loads to your current position plus some buffer.

CHAPTER 6

Experiments

In this chapter we will experimentally test if our idea and implementation of a persisted version of CCH is working. We answer answer the following questions:

- What is the largest graph we are able to index?
- Are we able to beat dijkstra's performance? If yes, by how much?
- Is there a category of queries which works better than others?
- Do different buffer sizes change the performance significantly?
- Is the input graph size the only factor that affects the performance?
- How long do updates take if we change all arc weights?
- Do updates affect the performance of the search?

6.1 The Test Environment

We implemented this CCH in *Java 17* and for *Neo4j 5.1.0*. The only addition Java library we use is *lombok version 1.18.24* for static code generation like getter and setters.
The code is deployed on a virtual machine which runs *Linux Mint 20.3 Una*. This VM has two AMD EPYC 7351 16-Core Processors with L1d cache=1 MiB L1i cache=2 MB, L2 cache=16 MB and L3 cache=128 MB. It has 512 GB of RAM and the system hard drive is an *Intel SSDPEKNW020T8* SSD with 2 TB. The fact that our test environment uses a SDD hard drive is not ideal. Databases usually use HDD drives which are slower in reading data but faster in writing data. However, our biggest index has 430MB in total, depending on the caching size and algorithm the disk uses, it would be possible that the HDD would pre cache the whole index, such that there are few actual reads to the disks in the HDD and the read performance gets closer to what a SSD can achieve.

6.2 The Test Data

The test graphs we evaluate for the implementation are provided by the 9th DIMACS Implementation Challenge - Shortest Paths [CD06]; we focus on the road networks of New York, Colorado, Florida, California+Nevada and also larger networks that represent the Great Lakes on the North American continent as well as the USA's east cost. We use the distance graphs, only in the case of New York we tried the distance and the time travel graph. As the results were similar and the contraction strategy does not depend on the arc weight we omitted the further tests with the time travel graphs.

6.3 Test Results

The Table 1 depicts our basic test results. The arc size differs from the DIMACS Challenge as we have filtered out duplicate edges.

	New York	**Colorado**	**Florida**	**California + Nevada**	**Great Lakes**	**Eastern USA**
$\|V\|$	264,346	435,666	1,070,376	1,890,815	2,758,119	3,598,623
$\|A\|$	730,100	1,042,400	2,687,902	4,630,444	6,794,808	8,708,058
$\|S\|$	2,153,002	1,680,290	4,397,804	8,598,552	17,833,050	17,712,722
$\frac{\|S\|}{\|A\|}$	2.93	1.59	1.62	1.85	2.62	2.03
$t_{contraction}$	545 s	233 s	579 s	4,384 s	25.29 h	23.29 h
$max(d(v))$	1,150	629	785	1,252	2,433	2,391
$\|\bigcirc A_{\uparrow}\|$	23.1 MB	21.9 MB	56.9 MB	107 MB	201 MB	215 MB
pos-file$_{\uparrow}$	1.1 MB	1.8 MB	1.3MB	7.6 MB	11.1 MB	14.4 MB
$\|\bigcirc A_{\downarrow}\|$	23.1 MB	21.9 MB	56.9 MB	107 MB	201 MB	215 MB
pos-file$_{\downarrow}$	1.1 MB	1.8 MB	1.3 MB	7.6 MB	11.1 MB	14.4 MB
$t_{dijkstra}$	0.816 s	0.549 s	2.630 s	4.858 s	5.425 s	5.387 s
t_{cch}^{640kB}	0.140 s	0.122 s	0.147 s	0.289 s	0.732 s	0.727 s
I/O^{640kB}	574	437	500	899	1671	1572
t_{update}	90 s	51 s	142 s	444 s	1827s	1557s
$t_{cch-updated}^{640kB}$	0.147 s	0.129 s	0.150 s	0.302 s	0.783 s	0.855 s
$I/O_{cch-upd.}^{640kB}$	569	457	516	924	2779	2716
$t_{cch-updated}^{20\%}$	0.136 s	0.130 s	0.092 s	0.183 s	0.660 s	0.680 s
$I/O_{cch-upd.}^{20\%}$	315	307	226	283	804	680
$t_{cch-updated}^{100\%}$	0.062 s	0.038 s	0.039 s	0.099 s	0.438 s	0.479 s
$I/O_{cch-upd.}^{100\%}$	0	0	0	0	1	0

Table 1 Graph overview table. $[t_{method}^{bufferSize}]$: average time in seconds

We do 10,000 point to point shortest path queries. We never start from the same vertex the and never query the same target vertex the previous search did. This is because starting from the same vertex or querying the same target results in the probably that at least on of the buffers already has the desired arcs in cache. This is desirable but does not reflect real world scenario. In case you always start from the same query, dijkstra will always be a good choice, as a *one-to-all* dijkstra can calculate all shortest paths from a start vertex to all reachable nodes in about 10 seconds on the Florida graph. This cannot be beaten by CCH as it can only do one to one queries. Though there is a *one-to-all* search algorithm relying on Contraction Hierarchies, called PHAST [DGNW13].

6.4 The Contraction

In Table 1 you can see the basic results of the networks we tested. One would think that the contraction time goes along with the size of the network, but it does not. The New York graph has has about the same contraction time as Florida which is about three times as big. Additionally the amount of shortcuts inserted relative to the already existing arcs is almost twice as big. This probably happens because the New York graph is much denser than the other graphs under test like, eg. Florida. In New York, regardless of whether you take the whole state or only the

city itself, there are four natural separators: *Manhattan and Brooklyn*, *Manhattan and Queens*, *Manhattan and Bronx*, *Bronx and Queens*, *Staten Island and Brooklyn* as well as *Staten Island and Manhattan to the mainland.*
In contrast to New York, the population of Florida is more sparse and located on a line at the cost of both side, as well as their streets. Therefore as shown in Figure 7 the contraction can easily find vertices as separators.

6.4.1 Limits

We decided to set the time limit a contraction should not exceed to approximately one day. If, within this time the contraction did not finish, we decided to abort the process. This happened for the graph *Western USA*, therefore we did not try larger ones. If one would want to calculate this size or bigger we suggest, to achieve the vertex ordering by recursive finding balanced separators as described in Customization Contraction Hierarchies [DSW16].
Contraction methods which rely on measures like edge difference suffer from very bad performance, if the graph becomes dense. At the same time, the remaining graph will become more dense towards the end of the contraction process. It is possible that the last few nodes form a complete graph. The graph that *Great Lake* was complete with 1078 nodes left in the queue. At this time it toke about 110 second to contract a single vertex. The reason is, the algorithm 2 for the contraction as proposed in this paper will always update the importance of its neighbors after each contracted vertex and re-push it to the queue Q of the remaining vertices. Updating the neighbor importance means to simulate the contraction of this neighbor. Thus we test for all pairs of incoming and outgoing neighbors $N_{\downarrow}(v) \times N_{\uparrow}(v) \setminus N_{\downarrow}(v) = N_{\uparrow}(v)$ whether we have to insert a shortcut. This you have to do for each element in the queue $|Q|$. In case of a complete graph the in- and the outgoing neighbor set will have size $|Q|$. Which lead to the this many neighbor checks $(|Q| * |Q| - |Q|) * |Q|$ which is almost $(|Q|)^3$ calls to the *getContraction()* method in Algorithm 2, to checks whether to insert a shortcut or not. In case the graph reaches completion on the last 100 this is a doable exercise. However, in case there are 3000 remaining, it will starve.
Our test show that as if the number of neighbors that are not contracted yet rise above 150, it takes 500+ milliseconds to contract a single vertex.

6.5 Query Performance

In this section we look into the query performance. The query performance depends mainly on the quality of the contraction and the buffer size. As the circular buffer we implemented performs better we will focus on this one.
One of the big questions can we even beat dijkstra's performance. As visible in Table 1, our CCH algorithm was faster on average. For all graphs under test the query times improved significantly in comparison to dijkstra. Especially for the graph which represents Florida the query performance improvement was tremendous. It was possible to reduce an average query time from 2.6 seconds to 0.039 seconds when having the whole graph in memory. But even if we only have 640kB in cache, the query performance did improve by factor five to sixteen. Increasing the cache to 20% of the edge size the graph has sometimes lead to better results, for example in California+Nevada. Here we could decrease the query time by half. For some other graphs it did not entail any improvement as for Colorado. The other graphs had some improvement, in the region of around 20%. A 20% caching strategy could be useful if you have an application which often queries the same vertex. With t_{cch}^{40kB} and $t_{cch-updated}^{40kB}$ we also tested the performance after updates have happened. It showed very little change in that. All queries got slightly worse but

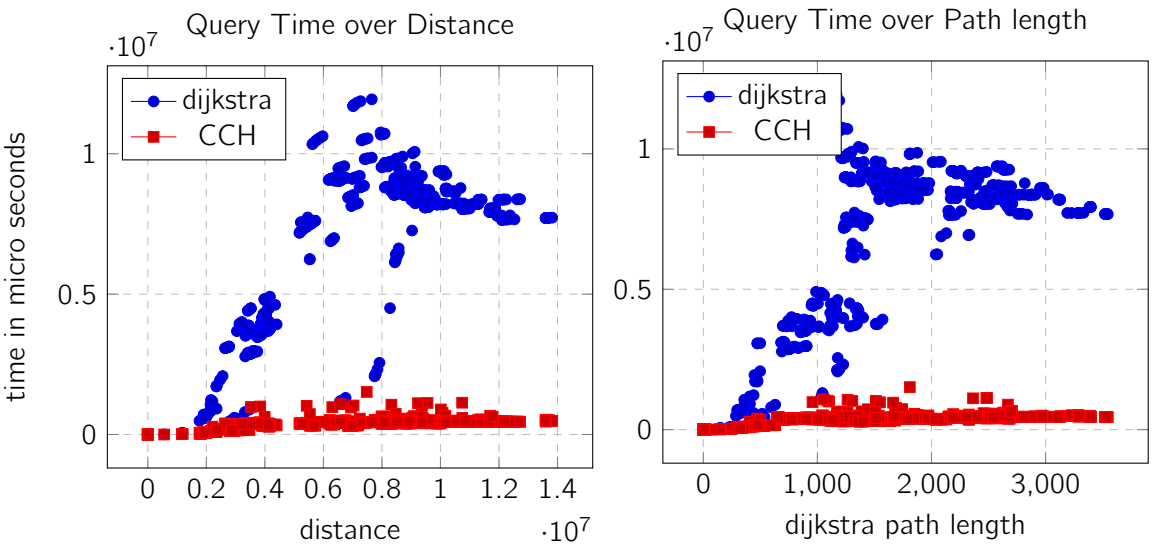


Figure 16 Comparison of CCH query performance on the California+Nevada graph, before updating. Buffer size 40kB. Random sample of 1000 vertex to vertex queries

did not loose much.

6.5.1 Short and Long Distance Queries

Figure 16 shows the speed difference of Dijkstra and CCH-shortest path query. As one can see, the bigger distance, the greater the advantage of CCH over dijkstra. Small queries where the shortest path involve only a few hundred vertices show little improvement in speed, whereas long distance queries are a lot faster. Figure 16 is a random sample of queries in California and Nevada. As visible in the left chart, there are some long distance queries for which dijkstra performs very good, although you cannot see them in the right chart, which has the path length on its x-axis. We assume the good performing long distance queries to be in Nevada as its road network is sparser compared to those of California. Therefore we added the right chart. It shows the advantage of CCH over dijkstra depends on the path length of the shortest path. This is underlining our theory as there are still some longer well performing queries but now they are closer together.

6.5.2 Query Analyses

Figure 17 we compare the amount of vertices the search query has to expand to find the shortest path. As expected Dijkstra's algorithm expands roughly quadratic many vertices to find the shortest path between vertices for shortest paths that involve up to 1000 vertices. After that the search touches the network borders and starts to expand the last leaves which happens almost linear.
The CCH search only expands vertices of higher rank. As you can see in Figure 17 the CCH expands at most around 1600 vertices. Therefore, we assume, CCH needs at most expand 800 vertices per search side to find the node with the highest rank. Thus no matter which source or target one chooses, the query will be bound to these 1600 vertex expansions. This is the reason CCH performs so well especially for long distance queries.
So far we have only looked at long distance queries. Now, we will have a look at the short ones, but queries where source and target are more that 300 vertices apart already perform as

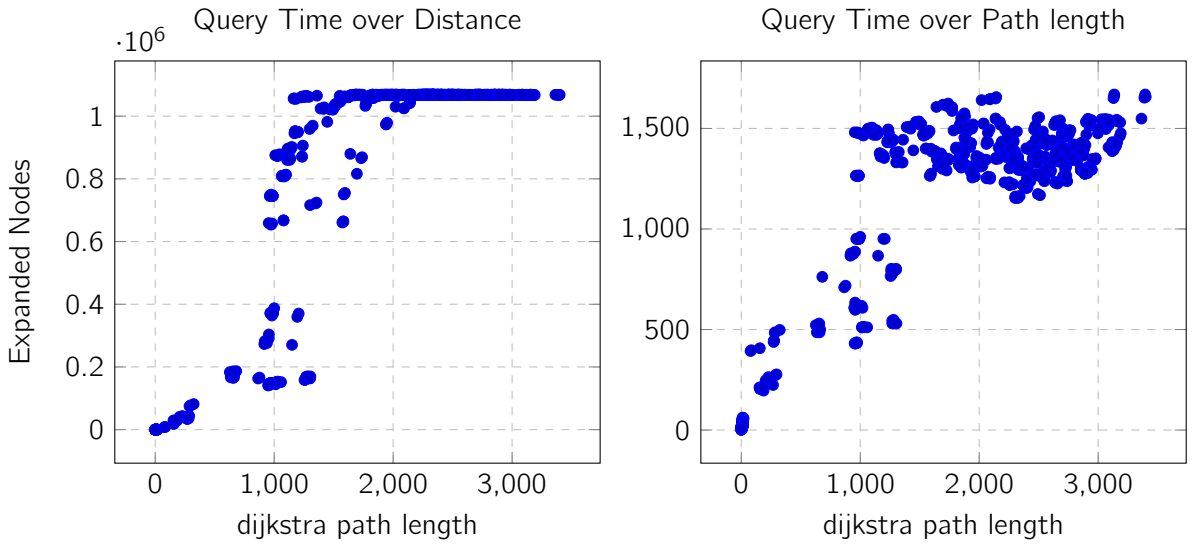


Figure 17 Comparison of CCH query performance on the California+Nevada graph, before updating. Buffer size 40kB. Random sample of 1000 vertex to vertex queries

good as or better compared to Dijkstra's algorithm. The search sides of these queries are even shorter than 800 expanded vertices. They can find the shortest path within a few hundred node expansions.

6.6 IO's at Query Time

we did our implementation to show that CCH could also fit for a graph database it is essential to have a look into how many I/O's are cased by our search. As stated in section 5.3, the arcs of one vertex are always stored in a single disk block. Therefore the worst case scenario is that there is one I/O per expanded vertex. In Figure 18 we can do better. With a cache size of only 640 kB which gives the possibility to hold 40960 arcs in cache we already achieve around 1.4 expanded vertices per I/O. This means in a bit less than every second we accidentally had the right set of arc in memory, when a new vertex was requested. This is pretty impressive as 81920 arcs are about 0.6% of the arcs the road network of California and Nevada contains after the contraction.

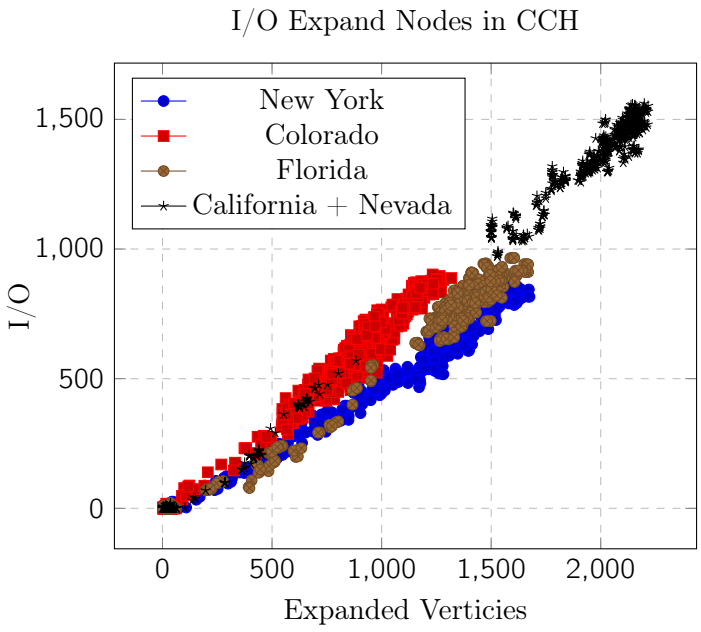


Figure 18 I/O's over Expanded Vertices for 40kB cache size per search side.

For bigger networks like the *Great Lakes* and *Eastern USA* this cache size was too small. In most scenarios we had about as many I/O's as expanded nodes. Thus we will have to increase in size.

6.7 Circular Buffer

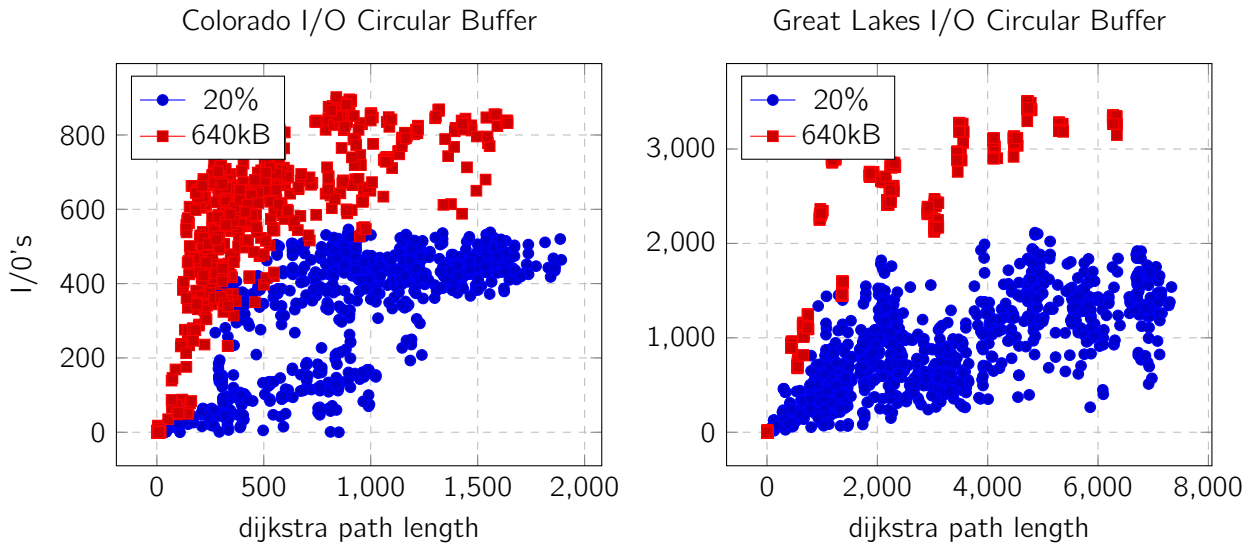


Figure 19 Comparison of CCH query I/O's. One query was done with 640kB buffer size, the other with 20% of all arcs

Here we want to compare the I/O's a query needs to find the shortest path. From the Table 1 it is visible that on average I/O's decreases heavily. In Figure 18 you can see the I/O's over the dijkstra path length. It becomes clear that the increase of the buffer size lead to a reduction of I/O's for almost all queries. Interesting to see is also the increased buffer size also lead to lower the upper bound of I/O's a query needs to finish. This is a very useful observation, it is easily possible to figure out how many I/O's you can afford and choose the buffer size respectively.

6.8 Updates and I/O's

In Table 1 I/O^{640kB} and $I/O^{40kB}_{cch-upd.}$ present how many I/O's a query needed on average, before and after. As you can see the number of I/O's roughly doubled for all of them. However after multiple updates it did not get worse, than that.

CHAPTER 7

Future Work

With this work in hand we did a basic but not fully feature complete implementation of CCH in Neo4j.

There are a lot of specific topics one investigate and implement to improve CCH for Neo4j.

7.1 Contraction Algorithms

Since the contraction Algorithm 2 we used definitely has its limitations as we have seen in the experiment section 6.4.1, it is worth to further explore this topic. One very promising method determine the vertex by recursively looking for minimum balanced separators until they tend to get big and then continue with our algorithm. Another question we have not even touched is, what happens if we add vertices and arcs. How can we deal with such updates which are not unusual for databases.

7.2 The Test Dataset

Changing the test domain can also be interesting, too. Papers like CCH [DSW16], CH [GSSV12] and many more focus on road networks. It sense for this paper to test if we end up to similar results. However, one could also try grids as Storandt [Sto13] or look for other real world domains which can be modeled such that shortest path queries are of major interest.

7.3 Perfect Customization and Transition to CH

Neither did we implement the perfect customization, nor the transition from CCH to CH. A CH index is usually faster at query time because the amount of arcs to expand is less. Therefore it can be useful to have a CCH and always calculate a CH from it after each update. In that scenario it is also useful to measure how the time frame of such.

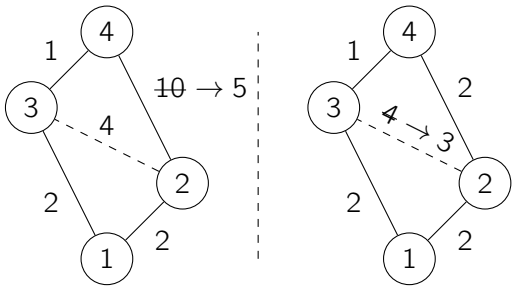


Figure 20 Perfect Customization example

7.4 Using relational Databases

Furthermore it is questionable if
graph databases like Neo4j solve problems. Due to "The Case Against Specialized Graph Analytics Engines" [FRP15] there is no significant difference in processing a graph in a graph database compared to a relational database. Therefore it would be interesting to see how an implementation of dijkstra and (C)CH perform in SQL written for relational databases.

CHAPTER 8

Conclusion

The research presented in the thesis at hand demonstrates the successful integration of Customizable Contraction Hierarchies (CCH) into the Neo4j graph database, aimed at enhancing the efficiency of shortest path queries. The adoption of CCH into Neo4j led to significant improvements in query processing speeds, particularly in road networks where certain vertices play a more crucial role in the multitude of shortest paths. The study also explores the impact of updating edge weights on the performance of the CCH index. It was found that despite multiple updates, the CCH index maintained its efficiency, proving its robustness in dynamic environments where data changes frequently. This aspect is particularly important in real-world applications where data is not static but continuously evolving. The results showed that the index graph, could be efficiently managed, thus ensuring quick access and processing. An important aspect of this research was to keep the integration of CCH into Neo4j loosely coupled, thereby allowing for easier portability to other graph databases in the future. This approach not only enhances the adaptability of the research findings but also paves the way for future developments in this field. In conclusion, this thesis makes a contribution to the field of graph databases by improving the efficiency and performance of shortest path queries in Neo4j through the integration of CCH. It opens up new avenues for further research, particularly in the area of dynamic data handling and cross-platform applicability of such advanced querying techniques.

Bibliography

[ADF+12] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Hldb: location-based services in databases. November 2012.

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. pages 230–241, 2011.

[AH15] S Tanenbaum Andrew and Bos Herbert. *Modern operating systems*. Pearson Education, 2015. page 194 - 222.

[aiaOPRSCU23] Oracle and/or its affiliates 500 Oracle Parkway Redwood Shores CA 94065 USA. Class priorityqueue<e>. Internet, 2023.

[BDD09] Emanuele Berrettini, Gianlorenzo D'Angelo, and Daniel Delling. Arc-flags in dynamic graphs. In *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, apr 2007.

[BS20] Johannes Blum and Sabine Storandt. Lower bounds and approximation algorithms for search space sizes in contraction hierarchies. 2020.

[CD06] David Johnson Camil Demetrescu, Andrew Goldberg. 9th dimacs implementation challenge. https://www.diag.uniroma1.it/challenge9/download.shtml, 2006.

[D'E19] Nicolai D'Effremo. An external memory implementation of contractionhierarchies using independent sets, 2019.

[DGNW13] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, dec 1959.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21:1–49, apr 2016.

[EEP15] Alexandros Efentakis, Christodoulos Efstathiades, and Dieter Pfoser. Cold. revisiting hub labels on the database for large-scale graphs. In *International Symposium on Spatial and Temporal Databases*, pages 22–39. Springer, 2015.

[Flo64] Robert W Floyd. Algorithm 245: treesort. *Communications of the ACM*, 7(12):701, 1964.

[FRP15] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[GH05] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156–165, 2005.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. pages 319–333, 2008.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, aug 2012.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[OYQ+20] Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, and Xuemin Lin. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *Proceedings of the VLDB Endowment*, 13(5):602–615, jan 2020.

[SSV] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile route planning. pages 732–743.

[Sto13] Sabine Storandt. Contraction hierarchies on grid graphs. In *Annual Conference on Artificial Intelligence*, pages 236–247. Springer, 2013.

[Zic21] Anton Zickenberg. A contraction hierarchies-based index forregular path queries on graph databases, 2021.