# Query Processing on Dynamic Networks with Customizable Contraction Hierarchies on Neo4j

## MSc Thesis Title

by
**Marius Hahn**

at the

Universität Konstanz

Faculty of Sciences
Department of Computer and Information Science

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

Konstanz, 2023

# Abstract

# Contents

CHAPTER 1

# Introduction

intro

CHAPTER 2

# Related Work

As this is mainly a database paper, we want to divide this chapter in two main sections Algorithmic History and Contraction Hierarchies Database History. Algorithmic History that will give some basic overview what has been published regarding index structures to speed up shortest path queries for graphs. Contraction Hierarchies Database History we will try to give an overview of efforts that have been made to make [DSW16, Customizable Contraction Hierarchies] it suitable for graph databases.

## 2.1 Algorithmic History

[GSSV12, Contraction Hierarchies] or CH is heavily influenced by the idea of the [BFSS07, Transit-Node] approach and as transit node approach itself, CCH is a technique to speed up [Dij59, Dijkstras Algorithm], which is the most basic and robust algorithm to find shortest path in graphs. CH goes back to the diploma thesis of [GSSD08, Geisberger] in 2008. The [BFSS07, Transit-Node] approach tries to find vertices inside the graph that are more important than others. Important in this case means, these are vertices that reside on many shortest paths. This speeds up especially long distance queries, as one only needs to calculate the distance to the the next transit node of the source and target vertex as the shortest paths between the transit or access nodes will be known.
CH goes even further on the idea of having important vertices. It applies an importance to each vertex in the graph a so called rank. Furthermore it adds edges to the graph, so called shortcuts, that preserve the shortest path property of the graph in case a vertex that is contracted resides on a shortest path between others. When querying a shortest path CH uses a modified bidirectional-dijkstra that is restricted to only visit nodes that are of higher importance, or rank, than the its about to expand next. 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 [DSW16, Customization Contraction Hierarchies] 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, they 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. Additionally the [DSW16, Customization Contraction Hierarchies] provides an update approach that only updates, edges that are affected by a weight change.

## 2.2 Contraction Hierarchies Database History

There is one bachelor thesis by Nicolai D'Effremo [D'E19, Some text] that has implemented a version on [GSSV12, Contraction Hierarchies] for Neo4j, one of the most used graph databases of today in 2023. This implementation shows that even in 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. [Zic21] showed in his bachelor thesis that it is even possible to restricted these queries with label constraints. Although CH and CCH have little difference, sadly we could not use much of the code provided by there works. It was deeply integrated into the Neo4j-Platform and since then two major release updates happened that have breaking changes which make it nearly impossible to reuse any of this code.

Finally there is [SSV, Mobile Route Planning] by Peter Sanders, Dominik Schultes, and Christian Vetter. In this paper it is described how one can efficiently store the a CH index structure on a hard drive. It states an interesting technique to how store edge that are likely to be read sequentially spatially close on the hard drive which makes read operations that have to be done during query time fast. The motivation of [SSV, Mobile Route Planning] through was slightly different. They came up with this idea because computation power on mobile devices is limited, so they could precalculate the CH index on a server and then later 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 are many similarities.

CHAPTER 3

# Preliminary

As the target platform for this work is the graph 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 doesn't play a role.

## 3.1 Notation and Expressions

We denote a graph $G(V, A)$ in case me mean an *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 to vertices $v_a$ and $v_b$ such that $v_a \neq v_b$, so there are no loops nor multi edges. An edge additionally has a weight function $w : A \to \mathbb{R}_{<0}$ it's weight which must be a positive.
We use $A$ as the arc set and $a$ a single arc which is directed. $a \epsilon A$ can be replace with $e \epsilon E$ which refers to edges that are *undirected*.
$G$ represents the input graph. The contraction graph $G'(V', A')$ is the graph that will be used at contraction for initially building the CCH index structure. A vertex $v$ in will never be really deleted. Instead the rank property $r(v)$ is set to mark this as an already contracted. So $V \equiv V'$ but $A \subseteq A'$ there will be edges add while building the CCH index. $S = A' \setminus A$ is the shortcut set that is added throughout the contraction.
$G^*(V^*, A^*)$ is the is the search graph while doing one a shortest path query. Futhermore one query will have two search graphs. $G^*_\uparrow$ representing the upwards search graph and the $G^*_\downarrow$.
Finally there will be the edge set of edges that are written to the disk. These will $\bigcirc A$ will be separated into to sets $\bigcirc A_\downarrow$ and $\bigcirc A_\uparrow$, too.

## 3.2 Dijkstra

some text

## 3.3 Updating Prio Queue

CHAPTER 4

# Customizable Contraction Hierarchies

In this section we will present the basic idea of [DSW16, Customization Contraction Hierarchies] and also work out the main difference between CCH and [GSSV12, Contraction Hierarchies]. It is far form being complete, but there will be some easy examples to show the concept.

## 4.1 Contracting

Algorithm 1 provides our contraction algorithm. We do what is called a *metric dependent* contraction in [DSW16, Customization Contraction Hierarchies]. This is a greedy algorithm which always takes the next best vertex to contract. Some use the simple edge difference as [GSSV12, Contraction Hierarchies], but we will use a more advanced metric as desired in

### 4.1.1 Example

In Figure 1 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 $S$ that 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. While contracting a vertex we want to preserve its via connection. If a vertex that 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 between the two. Let's reconstruct the contraction of Figure 1. At first vertex $v(1)$ is removed. As $v(1)$ resides on a simple path to between $v(3)$ and $v(5)$ and there is no edge $e(v(3), v(5)) \notin E'$, there must be a shortcut added to keep the via path. 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.

## 4.2 Searching

some text

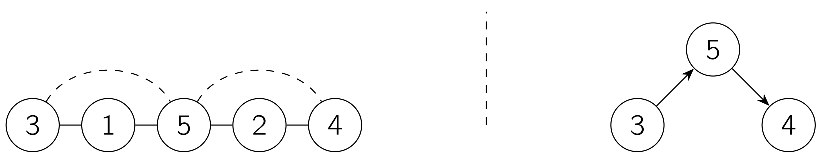



**Figure 1** The numbers inside the vertices represent their contraction order

**Algorithm 1** Insert Shortcuts Algorithm

```
function contractGraph(V)
    for v ∈ V do
        queue.offer(getContraction(v))
    end for
    while queue is not empty do
        contraction ← getContraction(queue.poll())
        v ← contraction.v
        v.rank ← rank
        updateNodeInNeo4J(vertex, rank++)
        for shortcut ∈ contraction.shortcuts do
            createOrUpdateEdge(vertexToContract, shortcut)
        end for
        for neighbor ∈ N↓(v) ∩ N↑(v) do
            queue.update(getContraction(neighbor))
        end for
        return v
    end while
end function
function getContraction(v)
    shortcuts ← []; outerCount, innerCountTimesOuter ← 0
    for inArc ∈ v.inArcs do
        if inArc.start.rank = Vertex.UNSET then
            outerCount++; inNode ← inArc.start
            for outArc ∈ v.outArcs do
                if outArc.end.rank = Vertex.UNSET then
                    innerCountTimesOuter++; outNode ← outArc.end
                    if inNode ≠ outNode then shortcuts.add(Shortcut(inArc, outArc))
                    end if
                end if
            end for
        end if
    end for
    ED ← |shortcuts| − outerCount − (outerCount = 0 ? 0 : innerCountTimesOuter / outerCount)
    return Contraction(v, ED, shortcuts)
end function
```

**Algorithm 2** Find Search Path

```
function find(start, goal)
    pickForward ← true; forwardQuery ← Query(start);backwardQuery ← Query(goal);
    while not isComplete(forwardQuery, backwardQuery, candidates.peek()) do
        query ← pickForward?forwardQuery : backwardQuery
        other ← pickForward?backwardQuery : forwardQuery
        pickForward ← ¬pickForward
        if isNotComplete(query) then query.expandNext()
        else continue
        end if
        latest ← query.latestExpand()
        if other.resultMap().containsKey(latest.rank) then
            forwardPath ← forwardQuery.getPath(latest.rank)
            backwardPath ← backwardQuery.getPath(latest.rank)
            candidates.offer(forwardPath + backwardPath)
        end if
    end while
    return candidates.poll()
end function
procedure expandNext()
    state ← queue.poll()
    latestExpand ← state
    if goals.contains(state.getEndVertex().rank) then
        shortestPaths.put(state.getEndVertex().rank, state.getPath())
    end if
    state.settle()
    for arc in state.getEndVertex().arcs do
        neighbor ← arc.otherVertex(state.getEndVertex())
        if mustUpdateNeighborState(state, neighbor, arc.weight) then
            newState ← state.getPath() + arc
            queue.update(State(neighbor, newState))
            seen.put(neighbor.rank, newState)
        end if
    end for
end procedure
```

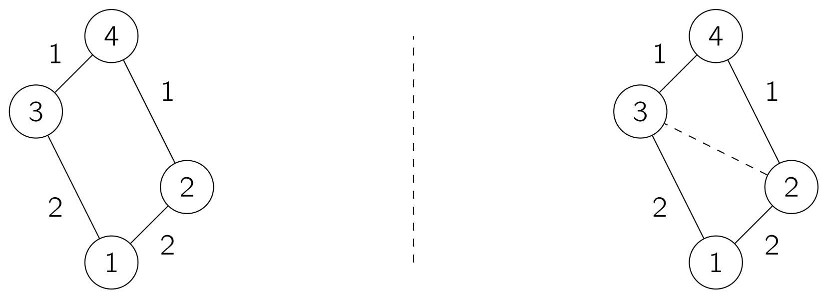



**Figure 2** The left represents a CH and the right a CCH contracted graph

### 4.2.1 Example

As we preserved all via paths during the contraction the shortest path can be retrieved by a bidirectional Dijkstra that 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 the meeting point, too. Finding at least one meeting point in the forward an 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 is it possible that there are 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.
The stopping condition for such a CH-Search is either, both forward and backward search, have reached the top vertex so there is no further vertex to expand, which happens in the example of figure 1 or, backward and forward search exceed the length that has already been found among the merged paths.

## 4.3 Difference between CH and CCH

Looking at the left graph in Figure 2 it has been contracted in the CH way, whereas the right is the CCH way. We explicitly state this here because we have found paper [OYQ$^+$20] that mix up these well known names, claiming they to Contraction Hierarchies CH while actually doing Customizable Contraction Hierarchies CCH. The main difference is, CH will only insert an 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 which is via vertex $v(4)$.
Whereas in the CCH case the edge weights do not play a role a contraction time. 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 later on we can easily update edge weights without inserting new shortcut, as all possibly needed shortcuts already exist.
Let's complete this example by updating the edge $e(v(2), v(4))$ that 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 need to update the weight and the middle vertex of the already give shortcut edge.

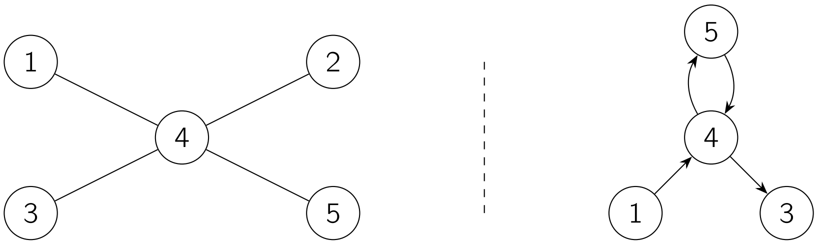



**Figure 3** The numbers inside the vertices represent their contraction order

## 4.4 Metric Dependent Vertex Order

There are two ways to get a suitable vertex order. A so called *metric independent* and a so called *metric dependent* one. The metric independent recursively uses 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 isn't trivial, 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 to expand in a search. However using only the edge differences doesn't lead to desired result. This is because during contraction there will be areas that get less dense than others. There are two problems that can arise. One is that important vertices are not contracted last. The other is the search space of the query gets linear although it could be logarithmic.

### 4.4.1 Important Vertices not contracted last

Looking at figure 3, this is a possible contraction order, if only the $ED$ is used 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 an $ED = -1$, too, same as vertex 5. Therefore the algorithm contracts $v(4)$ before $v(5)$. However 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, so $v(4)$ should be contracted last. The search graph on the right of Figure 3 shows why. 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'll have to expand $v(5)$. Although as you can see in the original graph on the right, its not possible that $v(5)$ is on the shortest path. Therefore a better contraction order would be a in Figure 1. This can be overcome by the method that is explained in section 4.5.

### 4.4.2 Linear Query Search Space

Regarding figure 4 there are three possible index graphs $G'$ of one and the same base graph $G$. The numbers inside the vertices represent the contraction order.

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 though 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.

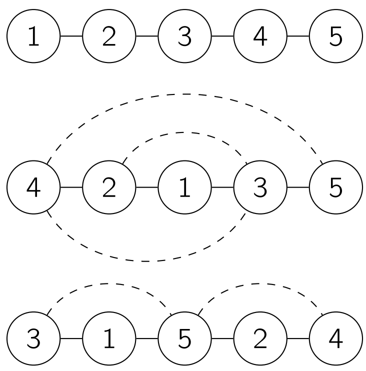



**Figure 4** Linear Contraction

The second $G'$ one contracts the middle vertices, which encodes the most shortest paths, 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 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 $e(v(4), v(5))$, so 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 most one single vertex. This example additionally shows that recursively finding a balanced separator, as proposed in [DSW16, Customization Contraction Hierarchies], is very a promising method to obtain a good contraction order.

## 4.5 Vertex importance

As shown in section 4.4.1 and 4.4.2 there are vertices that are more important that other vertices. Contracting these vertices late is key to get a efficient search later on.

### 4.5.1 Suitability of CCH

As it is important to contract important vertices last, the advantage one gets making a CCH search over a simple dijkstra run depends whether the base graph $G(V, E)$ has vertices that are more important than others. A vertex $v \epsilon E$ is important if there a many shortest paths that contain 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, have a look at [BS20, Lower Bounds and Approximation Algorithms for Search Space Sizes in Contraction Hierarchies].

### 4.5.2 Metric dependent Importance

As shown above, taking only the edge difference $ED$ into account doesn't necessarily lead to a proper order, we decided to take the vertex importance calculation that is propose by [DSW16, Customization Contraction Hierarchies]. To every vertex we add the level property $l(v)$. The level of the vertex with is initially set to 0. If a neighbor $w = N(v)$ is contracted level is set to $l(v) = max\{l(v) + 1, l(w)\}$. For every arc $a \epsilon A'$ we add a the hop length to the arc $h(a)$. The hop length is equals the number of arcs, this arc represent 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:

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

Our tests show that this importance calculation result in slightly increase 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 too.

## 4.6 Update CCH

**Algorithm 3** Update

```
procedure update()(G)
    Q ← G'.updatedEdges(G);
    while Q ≠ ∅ do
        a ← Q.poll();
        oldWeight ← w(a)
        newWeight ← determineNewWeight(a)
        if oldWeight! = newWeight then
            w(a) ← newWeight
            checkTriangles(Q, oldWeight, upperTriangles(a))
            checkTriangles(Q, oldWeight, intermediateTriangles(a))
        end if
    end while
end procedure
procedure checkTriangles(Q, oldWeight, triangles)
    for all triangle in triangles do
        if triangles.c() == triangle.b() + oldWeight then
            Q.push(triangle.c())
        end if
    end for
    return triangles
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 that 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 pops the the arc $a(x, y)$ with the lowest rank of the start vertex $x$. If there are multiple it pops the one with the lowest rank of $y$ among the ones with the lowest rank of $x$. Then we determine the new weight of the arc using the lower triangles. If there is a lower triangle that can be used as a pass through such that the arc weight in $G'$ does not change we do nothing. If the weight of the arc has changed we assign the new weight to the arc. Then we check all upper triangles, as drawn in figure 5, of $a(x, y)$ if there is an upper arc; denoted by $c$ in figure 5, that is influenced by this very change. If it is influenced by this change we push it to the priority queue. We do the same with all intermediate triangles.

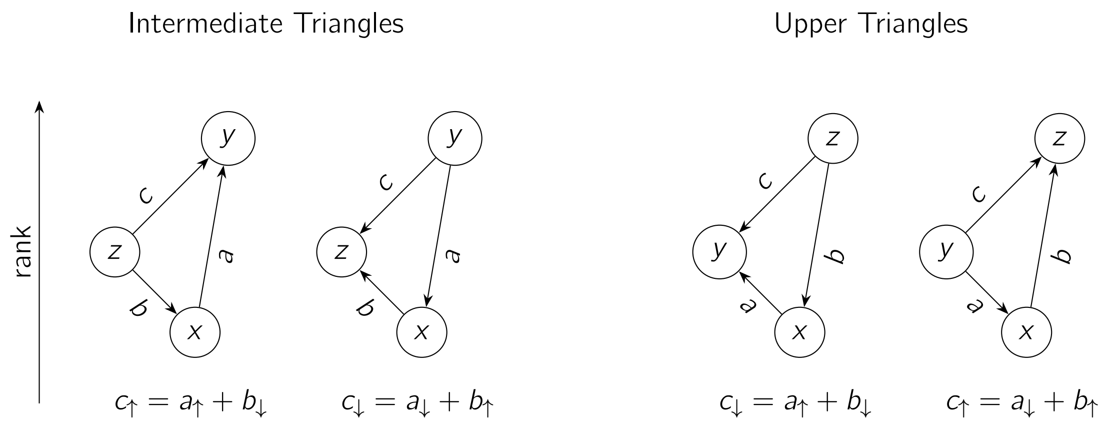



**Figure 5** Update Triangles

CHAPTER 5

# Integration in a Neo4j

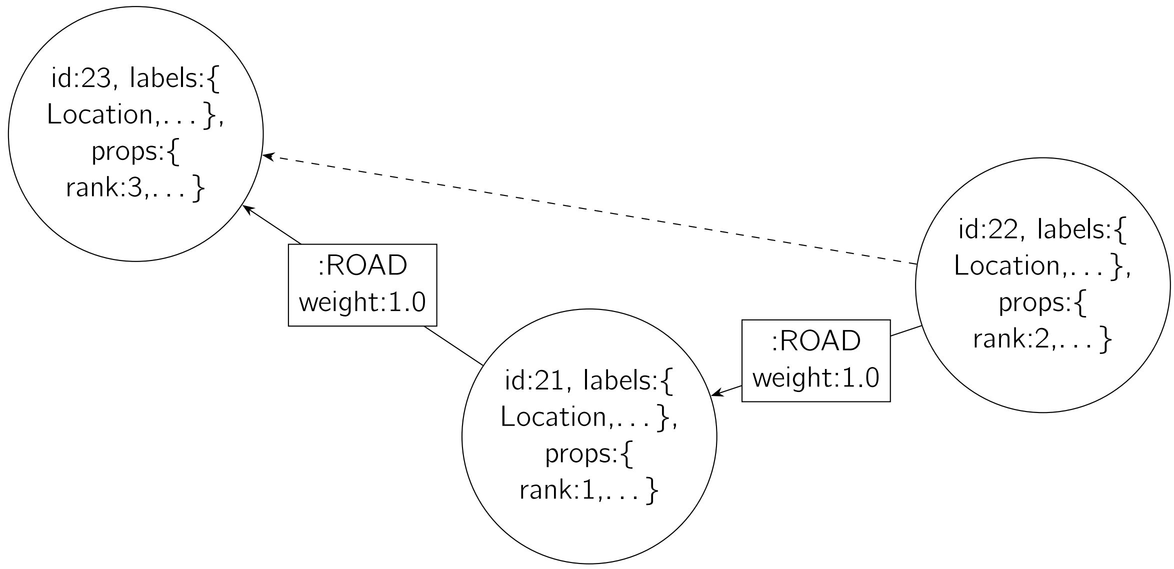
In this section it is described how "Customizable Contraction Hierarchies" CCH is integrated into Neo4j. CCH arguments the input graph, which means it inserts arcs, so called shortcuts, that do not belong to the original data. To keep the change to the input graphs as little as possible we decided to not insert any arc into the graph that is stored inside the neo4j database, but introduce another graph data structure, the index graph. This index graph has an mapping to the input graph that is held by the database, by inserting two properties into the node of the input graph. The *rank* this vertex has in the index graph and the *indexing weight* it had during the last customization process. This gives yet another two advantages. One is that we get full control about the graph representation which is helpful to efficiently store and read the index graph for the disk. Another is that the with this approach it makes it easier to later on port the idea to another graph database manufactures.

## 5.1 Index Graph Data Structure

The index graph data structure is neither a adjacency list nor adjacency matrix. There is a vertex object that has two hash tables. One for incoming arc and one for outgoing arcs. The hash tables keys are of type vertex and the value is the arc. An arc has a reference to its start vertex and one to its end vertex.
A disadvantage of this model could be that some modern hardware optimization that exist for arrays do not match with this data structure. When using an array, the values this 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 it is likely that they are accessed right after it. The model of the index graph is a linked data structure, a bit like a linked list. The elements of an linked list are contained somewhere in main memory. There is no guarantee that subsequent values have any spacial proximity. Therefore the just explained hardware optimization will not give any advantage.
However, this makes the makes the graph traversal easy. Additional 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. Additionally these hash tables only contain few elements. This makes this data structure efficient anyway. Test on small graphs [Oldenburg] show that cch queries can be answered in less than one millisecond, which is close to what we tested with the original cch application.

## 5.2 The Mapping

The in memory data structure of neo4j is similar to the just 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 relationship contain a lot of other information. A node has a collection of *labels*, relationship has a *type*. The class




| start rank | end rank | middle rank | weight |
|---|---|---|---|
| 2 | 3 | 1 | 2.0 |
| 1 | 3 | -1 | 1.0 |
| 2 | 1 | -1 | 1.0 |

**Figure 6** mapping

*Node* and the class *Relationship* are both derived from the class *Entity* which also has a collection of properties as well as and 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 to 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 isn't guaranteed to be unique anymore. It is deprecated and marked for removal.
As just explained there are lot of information in this data structure. A lot of information we don't need. Looking at 6 we only want to keep track of the information that is needed for the CCH index. Additionally as disks are divided into blocks and sectors we want to flatten the graph which is 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 edges $\bigcirc A$. A disk edge $a \epsilon \bigcirc A$ 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 edge sets $\bigcirc A_{\downarrow}$ for the downwards graph and $\bigcirc A_{\uparrow}$ upwards graph. $\bigcirc A_{\downarrow}$ contains all downward edge that which are needed for the backward search and $\bigcirc A_{\uparrow}$ contains all upwards arcs that are needed for the forward search.
During the the contraction every node gets a rank assigned. This rank is the only change that is made to the Neo4j data structure and its 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 disk arc sets $\bigcirc A_{\downarrow}$ and $\bigcirc A_{\uparrow}$, we now want to store them as efficiently as possible to the disk. To refine the definition of a disk arc. It consist of four values *start rank*, the *end rank*, the *start rank* and the *weight*. The first three are 32 bit signed integer, which gives a maximum indexable amount of vertices for $2^{16}$. The last one, the arc *weight* is a 32 bit

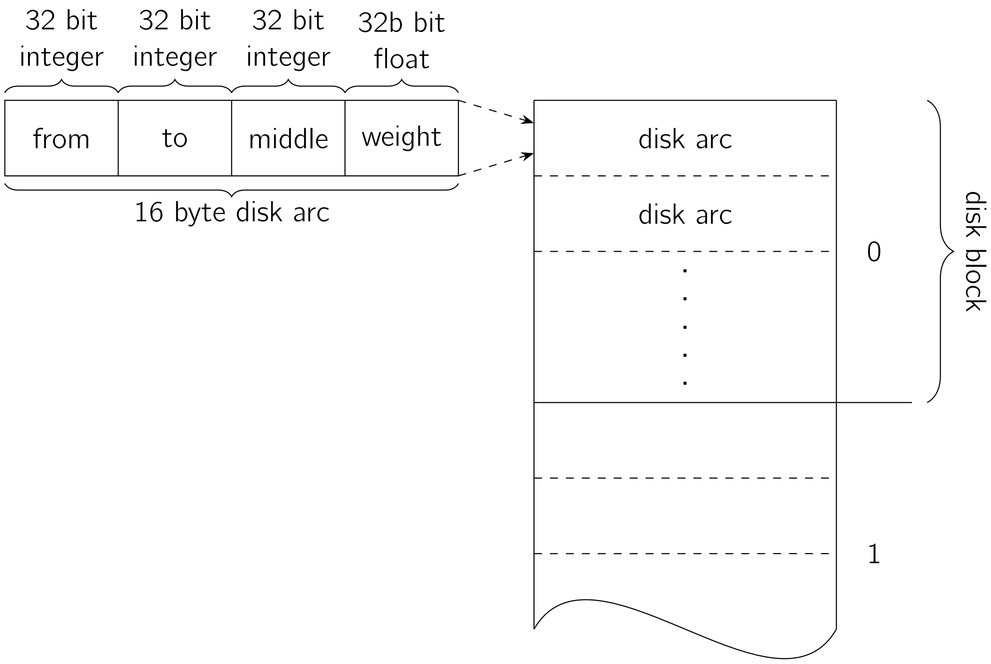



**Figure 7** Disk Block

floating point number. One could argue that 32 bit are not very precise, though our experiments we never had any imprecision problem. Furthermore, little imprecision would not cause a big problem, as the index graph is only needed to find the shortest path. The exact weight can later on be retrieved after the shortest path is resolved in $G$.
This 16 Byte disk collected to disk blocks, that have at least the size of one disk block on the systems hard drive, as usually the system will always read a complete disk block even if you request only 16 Byte of information. But it is possible to make blocks bigger than one disk block. This can be wanted if you have a big read cache, or even need it the highest outgoing or in ingoing arc degree exceed the $\frac{diskBlockSize}{16}$ as you will see in 5.3.1.

### 5.3.1 Persistance Order

As the disk arc are later on is used for the CCH search, we want to sequentially write them in a way that provides a high spatial proximity of vertices that are likely to get requested together. Here we will adopt the idea of [SSV, Mobile Route Planning]. In the transformation form $G'$ to its disk arcs $\bigcirc A_{\uparrow}$ and $\bigcirc A_{\downarrow}$ we do a simple depth first search on all ingoing arcs on the target rank to determine the order for $\bigcirc A_{\uparrow}$. We do the same for all outgoing arcs on the source rank to determine the order for $\bigcirc A_{\downarrow}$. There is one file for storing all upward arc and on file storing all downward arcs. Each of this files has a position file that belongs to it, as shown in figure 8. After every vertex that is expanded, all its arcs are pushed to a buffer of size disk block. If the buffer is full or the number of arcs doesn't fit anymore, the buffer is flushed to the current position in the arc file and the file position is incremented by 1. The store position is saved in the position file of the arc file, such that we can find the arcs back later on. If the buffer wasn't complete full all remaining arc slots are filled with dummy arcs that contain only $-1$ at every property.

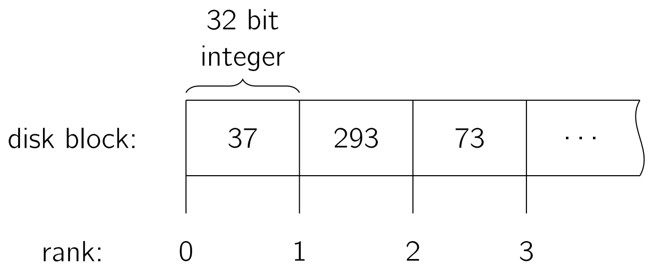



**Figure 8** Position File

## 5.4 Reading Disk Arcs

If one wants to get all upwards arcs of rank $i$ one need take the upwards position file, retrieve the integer $j$ that is stored at index $i$, and then read the complete block $j$ in the upwards arc file. There There one will get an array, contain the requested arcs but also some other. These arcs are likely to be request next. Therefore we want to keep them in memory. We implemented two buffer a *circular buffer* and a *least recently used buffer* LRU

### 5.4.1 Circular Buffer

For the circular buffer we simply used an array of disk arcs. If we reach the end of the buffer we restart overwriting the values from the array start. To get the all arcs of a rank one request that rank number. There is a position hash table which tells the start position of that rank inside the buffer. If it is missing, the containing disk block is read to the buffer. It continues to read sequentially until the request rank and the read arc doesn't belong together anymore. This buffer has the advantage that we can exactly determine the amount of arcs we buffering. Also as it is just a simple array, it will be easy for the operation system to cache it. The disadvantage is that it is possible that we request arc sets very often as it is possible they get evicted just before request again.

### 5.4.2 Least Recently Used Buffer

Cache is a *java.util.LinkedHashMap*. This class provides the possibility to evicted the entry that has been requested longest time ago. 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 have always the same size. Higher rank vertex usually have bigger sets as they are of higher degree. The advantage is, it is very easy to implement and therefore very resilient to programming errors.

## 5.5 The Search

The search brings all things explained in this chapter together.

At the beginning there are to index graphs initialized 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 $A_\uparrow$ only contains the upward edges to $v(s)$. The vertex set $V_\downarrow$ of the downwards graph $G'_\downarrow(V_\downarrow, A_\downarrow)$ only contains the $v(t)$ and the edge set $A_\downarrow$ only contain the arcs to $v(t)$.
Now both $G'_\uparrow$ and $G'_\uparrow$ are alternatingly expanded with CH-Dijkstra. When the CH-Dijkstra has

decided which vertex to expand next, the buffer will be requested to load its neighborhood. This can be compared to Command & Conquer or a lot of other strategy real-time strategy video games where you start at a map that is almost completely grey and the map is only load to the position you are plus some padding. The rest is just a plain CH-Search where we stop after we determined the correct shortest path.

CHAPTER 6

# Experiments

In this chapter we will experimentally check if our idea and implementation of a persisted version of CCH works out.

## 6.1 The Test Environment

We implemented this CCH in *Java 17* and fo *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 runs on a virtual machine that is running *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 a *Intel SSDPEKNW020T8* SSD with 2 TB.

## 6.2 The Test Data

The test graphs we evaluate the implementation are provided by the [CD06, 9th DIMACS Implementation Challenge - Shortest Paths]. There we focus on the road networks of New York, Colorado, Florida and California+Nevada. We use the distance graphs, only in case of New York we tried the distance and the time travel graph. As the results were similar and the contraction strategy is not depending on the arc weight we omitted the further test wit the time travel graphs.

## 6.3 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, though it doesn't. 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 is probably happens because the New York graph is a lot denser than the other graphs under test like Florida. In New York, regardless if you take the 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.*
Where as 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 4 the contraction can easily find vertices as separators.

| | New York | Colorado | Florida | California + Nevada |
|---|---|---|---|---|
| Verticies | 264,346 | 435,666 | 1,070,376 | 1,890,815 |
| Edges | 733,846 | 1,057,066 | 2,712,798 | 4,657,742 |
| Shortcuts | 2,153,002 | 1,680,290 | 4,397,804 | 8,598,552 |
| Shortcuts added Relative | 2.93 | 1.59 | 1.62 | 1.85 |
| Contraction Time | 545 s | 233 s | 579 s | 4,384 s |
| Max In Degree | 1,150 | 629 | 785 | 1,252 |
| Out Arcs Disk Size | 23.1 MB | 21.9 MB | 56.9 MB | 106.5 MB |
| Out Position File Size | 1.1 MB | 1.8MB | 1.3MB | 7.6 MB |
| In Arcs Disk Size | 23.1 MB | 21.9 MB | 56.9 MB | 106.5 MB |
| In Position File Size | 1.1 MB | 1.8MB | 1.3MB | 7.6 MB |
| Block filling | 0.99 | 0.95 | -122.0 | -122.0 |
| AVG Dijkstra Search Time | 0.816 s | 0.549 s | 2.630 s | 4.858 s |
| AVG Search Time (40kB) | 0.140 s | 0.122 s | 0.147 s | 0.289 s |
| AVG Disk Access (40kB) | 574 | 437 | 500 | 899 |
| Update Time | 90 s | 51 s | 142 s | 444 s |
| AVG Search Time (40kB) after multi update | 0.147 s | 0.129 s | 0.150 s | 0.302 |
| AVG Disk Access (40kB) after multi update | 569 | 457 | 516 | 924 |

**Table 1** Network overview table

### 6.3.1 Limits

We decided to set the time limit a contraction should not exceed to one day. If, within this time the contraction did finish, we decided to abort the process. This happend for the graphs *Western USA, ...*. If one would want to go this size or bigger we suggest, to achieve the vertex ordering by recursive finding balanced separators as described in [DSW16, Customization Contraction Hierarchies].
Contraction methods that rely on measures like edge difference suffer from very bad performance, if the graph gets dense. At the same time, the remaining graph will get denser towards the end of the contraction process. It is possible that the last few nodes form a complete graph. The algorithm as proposed in this paper always will update the importance of it's neighbors after each contracted vertex and re-push it to the queue $Q$ of remaining vertices. Update the neighbor importance means to simulate the contraction of this neighbor. So we check 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 $|Q|$ times. 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 $(|Q|)^3$ checks whether to insert a shortcut or not. In case your graph already get's complete or close to it on the last 100 this is a doable exercise. In case there are 3000 remaining, it will starve.

## 6.4 Query Performance

In this section we will have a look at the query performance. The query performance will depends mainly in quality of the contraction and the buffer size. As the circular buffer we implemented performs better we will focus on it and bring only a small comparison at the end to the LRU

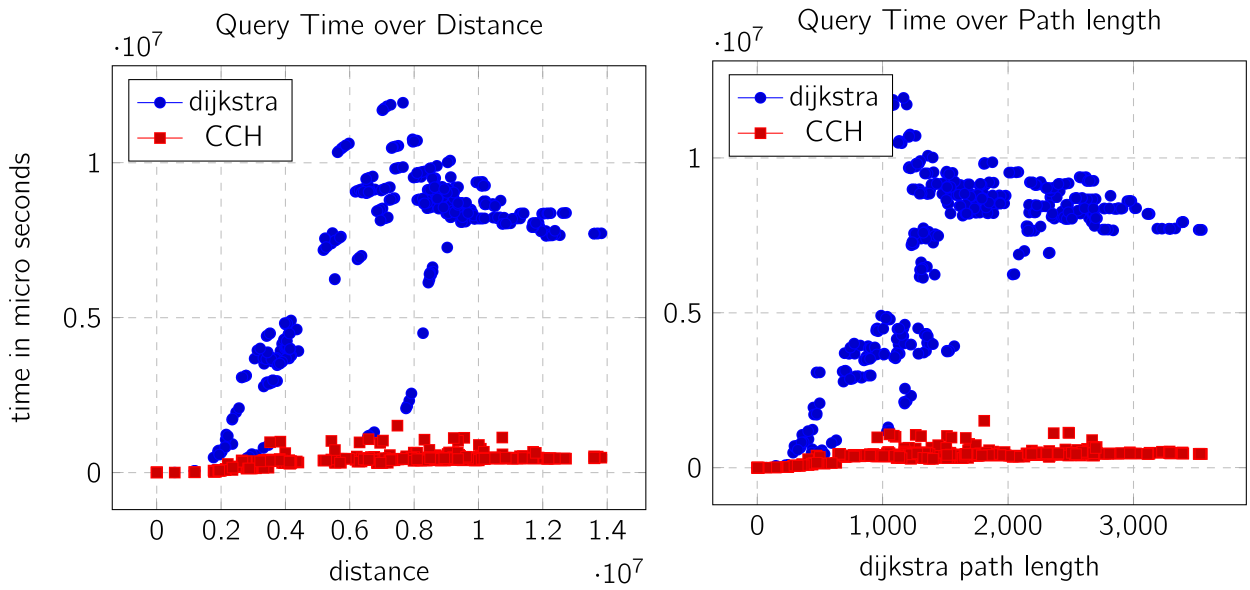



**Figure 9** Comparison of CCH query performance on the California+Nevada graph, before updating. Buffer size 40kB. Random sample of 1000 vertex to vertex queries

buffer.

### 6.4.1 Comparison with Dijkstra

Regarding Figure 9 shows speed difference of Dijkstra and CCH-shortest path query. As you can see, the greater distance, the greater the advantage of CCH over dijkstra. Small queries where the shortest path involve only a few hundred vertices have a very little speed up, whereas long distance queries are a lot faster. Figure 9 is a random sample of queries in California and Nevada. As you can see in the left chart, there are some long distance queries for which dijkstra performs very good, though you cannot see them in the right chart, that has the path length on the x-axis. We assume the good performing long distance queries to be in Nevada as its road network is sparser and than 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.

Having a look a figure 10 we compare the amount of vertices the search query has to expand to find the shortest path. As expected dijkstra 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 10 the CCH expands at most around 1600 vertices. Therefore, we assume, CCH needs at most expand 800 vertices per search side the find the node with the highest rank. So no matter which source or target one chooses, the query will be bound to these 1600 vertex expansions. This is the reason CHH performs so good especially for long distance queries.

So far we only had a look long distance queries, let's have a look at the short ones, but queries where source and target are more that 300 vertices away already perform as good or better than dijkstra. The search sides of these queries are even shorter than 800 expanded vertices. They can figure out the shortest path within a few hundred node expansions.

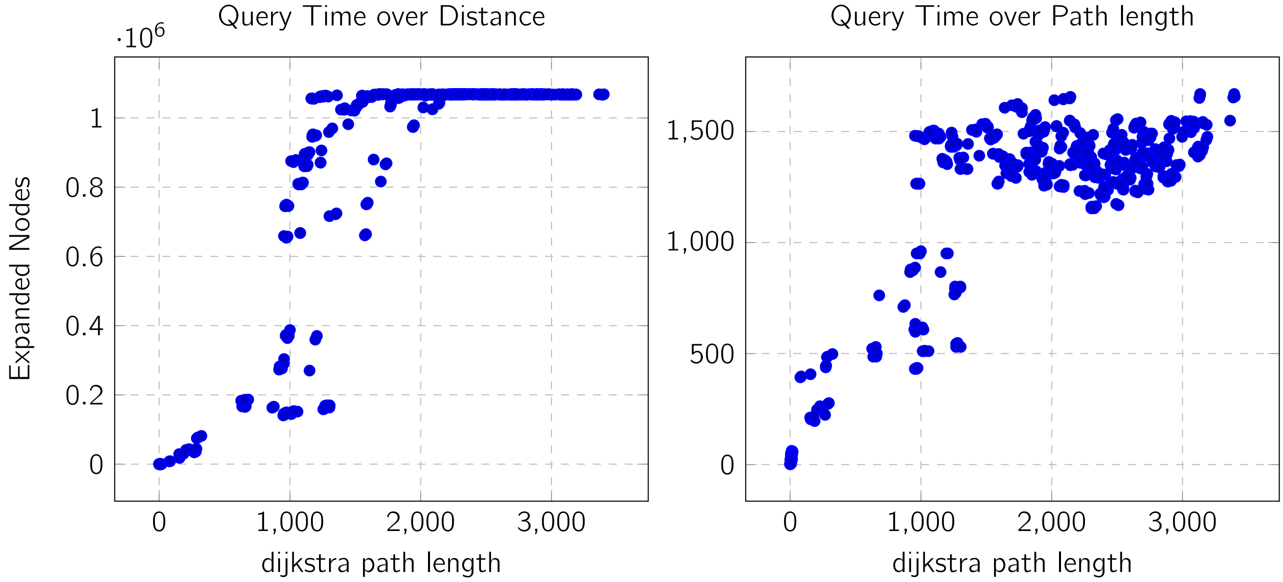



**Figure 10** Comparison of CCH query performance on the California+Nevada graph, before updating. Buffer size 40kB. Random sample of 1000 vertex to vertex queries

### 6.4.2 Disk Access at Query Time

some text

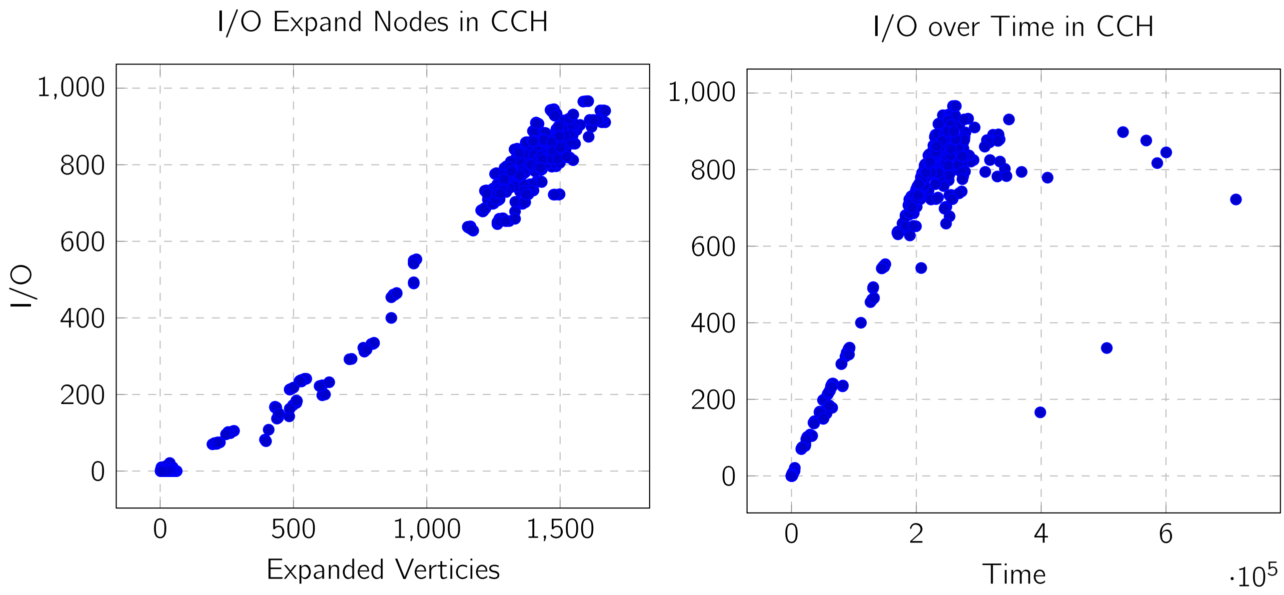



**Figure 11** Some Caption

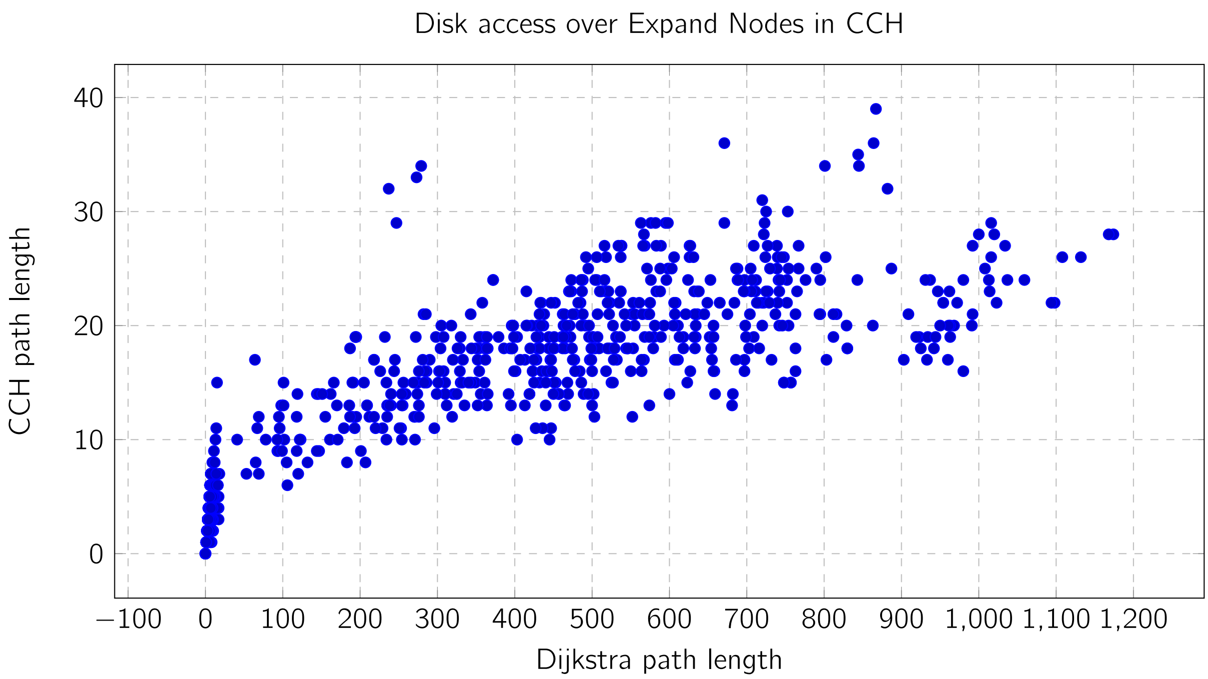



**Figure 12** Some Caption

# Bibliography

[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.

[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.

[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.

[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.

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