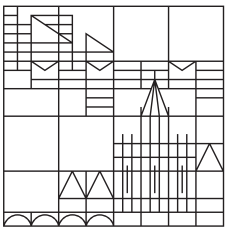
Query Processing on Dynamic Networks with Customizable Contraction Hierarchies on Neo4j

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

Neo4J is a database that uses a graph model to handle data. In contrast to relational databases, graph databases handle relationships a first-class citizen. This means a relationship that connects nodes is an entity just as the nodes it connects themselves. Nodes have a direct pointer to their relationships, and relationships have a pointer to their start end node. Whereas 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 different table using joins on matching cells, usually primary- foreign key relationships. To retrieve information that is stored in two tables 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 that have to be done to retrieve the desired information, such queries can be very expensive.

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 such connected data, which can make such queries very performant. Examples for such queries are shortest path queries on domains which resembles graph by their nature.
We will focus on these, shortest path queries and try to make these even quicker by adding an index call Customizable Contraction Hierarchies, *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.
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. Therefore, will we be able to store the index in a fashion such that we can keep the performance gain we had before. Additionally we want to explore how the indes behaves after an update on the 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 discover how big such an index graph will get, as thre 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 later on as easy as possible to port the implementation to other graph databases that might become more relevant in future.

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 in Neo4j. 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 in Neo4j 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

There have been many approaches to become quicker than dijkstra. To name some there is [HNR68, A*], which adds a heuristic to each node to help dijkstra to go in the right direct. Then there is [GH05, ALT] with stands for *A* + Landmarks + Triangle Inequality*, which is a approach to improve the A* heuristic with landmarks. Then there are the [BDD09, Arc Flag] approach that which is based on the idea that many shortest paths in a graph usually overlap, so we can store them in a compact way. Later only edges which are flagged true for the target direction will be expanded by dijkstra. Then there is the [BFSS07, Transit-Node] that uses the observation that if you want to go far, nodes of a small area usually pass all by the same node. If you can determine such nodes you store their distance in a table. If finally you are looking for the way from one to another node one only starts to local dijkstra for the source and the target side until one finds transit node. As the distance between the transit nodes are known, we simply can look up the rest of the path. [GSSV12, Contraction Hierarchies] or CH, as transit nodes, goes with the idea that there are nodes that are more important than others, but instead of selecting some important nodes we rank them all. CCH 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 [GSSD08, Geisberger] in 2008. As the transit node approach CH and CCH speed up especially long distance queries.
CH 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 Neo4j Shortest Path Queries

Neo4j contains a implementation of dijksta, bidirectional dijksta and an A* implementation out of the box. They are provided by the traversal API, such that every plugin can use it. Sadly the only provide a *one to one* dijksta, which made it useless for our purpose. Additionally our test that that prove our dijkstra implementation as stated in 3.3, uses the neo4j bidirectional dijkstra and compares the path weight. By that we also measure the time both need to find the shortest path. It shows our undirectional dijkstra is about twice as fast as the one of neo4j.
Then there is the neo4j graph data science library which provides a *one to all* and *one to one* dijkstra implementation. Additionally it contains also a A* implementation, a *Yen's Shortest Path* algorithm and a *Delta-Stepping Single-Source Shortest Path* algorithm.

2.3 Contraction Hierarchies in Neo4j

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 Updating Priority Queue

One data structure that is heavily used in this paper is a priority queue. This a very useful structure as it returns always the element with the lowest priority. In dijkstras algorithm for example, the priority is calculated on the weight, so by retrieving an vertex you will always get the next shortest path. The problem arises when it comes to updating an element of an priority queue that is already in the queue, because a priority queue can holt the same element multiple times and even with the different priorities. This explanation refers to the [aiaOPRSCU23, Java 17 reference], but priority queues are which based an a [Flo64, binary heap] as this one should have all the same properties.
One might ask why can't we simply remove the element that is already in the queue and then re-push it. That is actually possible but slow, as the operations *contains(element), and remove(element)* are running in linear time $\mathcal{O}(n)$. Better would be to use *offer(), poll(), remove() or add()* which run in logarithmic time $\mathcal{O}(\log(n))$. One could think of just keeping a reference to the element and later on change the priority as needed. But the queue will not be notified buy such a manipulation, and because queue keeps the next element to dequeue at the top one will retrieve the wrong element. So we have to cum up with something better. If the priority

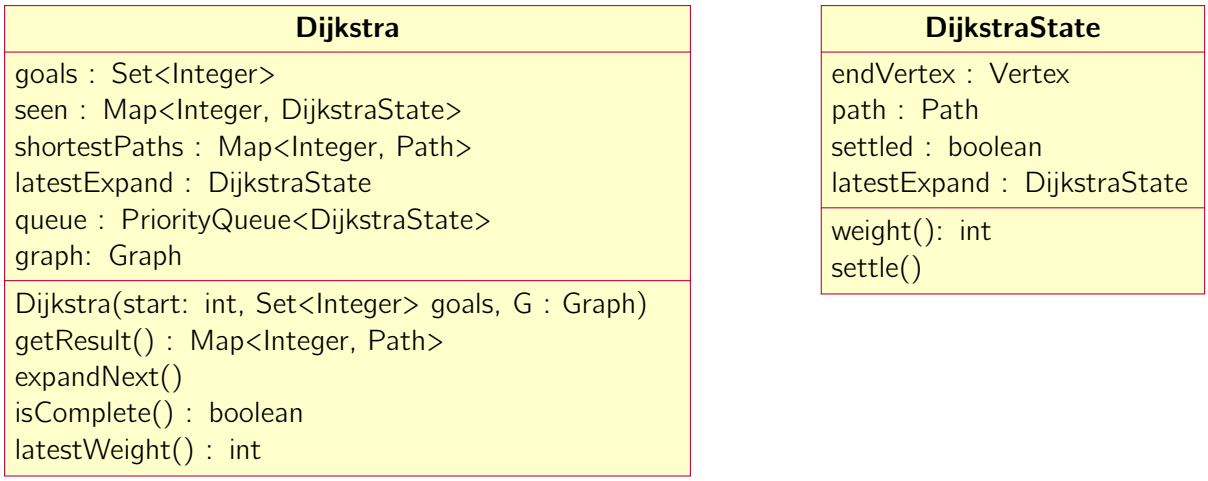


Figure 1 Dijkstra Class Diagram

of an element can only decrease, this usually doesn't cause a problem, because by retrieving an element you will definitely get the one with the lowest priority. Therefore you can simply repush elements to the queue, as you will always get the real smallest possible. To avoid processing an element twice you insert the elements into a set where you keep track of the already processed onces and if you retrieve a already seen one you simply pull again. If the priority can decrease and increase and you simply push updated the elements to again, the problem is a bit more difficult as now you can possibly dequeue an element with the old priority which is lower as the current but not valid anymore. We initialize a priority queue and a map which 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 map, 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 mal we increase the value it has in the map by one, wrap this new version number together with the element and push it queue. At pull we retrieve the element together with its current version number. If the number is not equal to the one in the control map we simply pull again and check again until we find an element that is up to date.

3.3 Dijkstra

We want to do a little explain dijkstras algorithm as it crucial for the ch/cch search. Additionally we want to decuple two thing that are usually done together. The initialization of a dijkstra query and the iteration step, that expands vertices until the shortest path to the target is found. Dijkstra finds the shortest path from a source vertex to a target vertex by always go the *best* path. Dijkstra starts with expanding the neighborhood of the source vertex and will move on to the vertex that has smallest total distance to the source node. This is done until dijkstra is about to expand the target node next. 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, queries but also *one to many* and *one to all* shortest path queries.
Looking at algorithm 1 we initialize the query with a start id, a set of target ids and the graph on which we want to do the query. At the construction process the query will take the start id, create an object of type *DijkstraState* and push it to the *queue*. This priority queue is organized my the path weight of *DijkstraState* such that it always returns the shortest path that is inside the queue.
After the initialization we can start expanding nodes using the *expandNext()* method, which we provided in algorithm 1. At first we pull the 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, mean we do a *one to all* search. If so we add the just found shortest path to the *shortestPaths*. Then we settle that state we know that we found the shortest path for the endVertex of this state. Then we iterate over all arcs that are attached to this endVertex and therefore get its neighbors. For each of this neighbors we check if we have to update it with a new state in the queue. This is the case if either we haven't seen this vertex so far or the state we pushed with this vertex before contains a longer path 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 what 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 such a dijkstra query object as you can see in figure 1. Being able to call the *expandNext()* procedure from outside is a very powerful. For example if we want to write a bidirectional dijkstra, we can create to 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($t, [s], \overleftarrow{G}(V, \overleftarrow{A})$). Now we can build our algorithm around that tell the bidirectional query when to stop or which side to expand next, but the essential thing here is we have reused the logic of the *expandNext()* procedure and not written it again.

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 2 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 technique that assigns an importance to each vertex. This importance is further described in section 4.5.2, with 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 int the index graph G'. At first we calculate the so called *contraction* for each vertex, which is the set of shortcut arcs that 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 that 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 starts at 0, on the current vertex, add this information to 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 that connects the vertices of a shortcut, we update the weight of that arc if the shortcut weight is shorter and update the middle vertex to be able to later on reconstruct the actual path in the input graph G. In comparison to [DSW16, Customization Contraction Hierarchies] we do two steps in one. The first is adding shortcuts and the sections is basic customization, through lower triangle enumeration. We do because at this point we already have the lower triangles at hand, so we do not need to iterate onces again over all arcs to set the correct weight and middle node. Then we iterate over all neighbors of the just 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. We have to to this because 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 done, we return

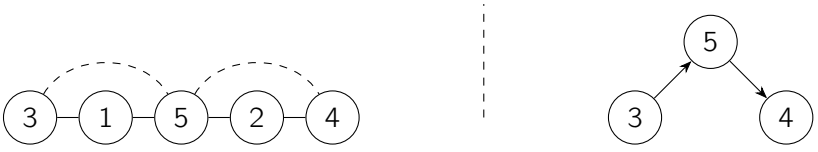


Figure 2 The numbers inside the vertices represent their contraction order

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
```

top vertex, which is the one with the highest rank. This vertex is later on needed to store the index to the disk.
To get a contraction of a vertex we have to iterate over all connected vertices of v that are not yet contracted. We initialize a collection of shortcuts to an empty list. Then we iterate over all incoming arcs of v, where the start vertex has not yet been contracted. For each of this 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 arc.
Finally we calculate the edge difference and return the vertex together with the edge difference and the shortcuts.

4.1.1 Lower Triangles

To determine an arc weight of in G' we have to have a look at its lower triangle as shown in figure 3. This is what the two for loop in *getContraction(v)* in algorithm 2 do. 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 that 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. There as shown in figure 3 it follows:

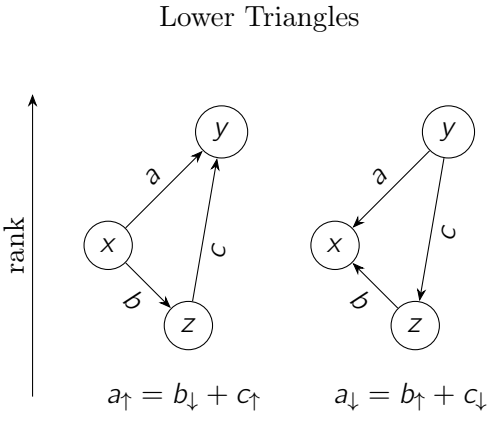


Figure 3 Lower Triangle

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

4.1.2 Example

In Figure 2 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. Let's reconstruct the contraction of Figure 2. 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.

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 shows our search algorithm. It finds the shortest path between the to vertices. As input parameter is takes two integer values, which represent the rank of the start vertex and the rank of the target vertex. At first we init a boolean variable that 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. As long as we have not found the shortest path we continue the search. We definitely have found the shortest path if either both queries have expanded the top node with the highest rank or the next vertex to expand in both queries is farer than the shortest 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 that is called *candidates*. It is organized by the path length and will return the shortest path merge that has been found on *candidates.peek()*. If the search is complete we simply peek 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. After that we flip *pickForward*, such that at the next iteration the respective other query will be expanded. If the query we are about 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 that 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 shortest path pairs found so far. As two merged shortest paths don't necessarily result in a shortest path, we still have to continue as described before.

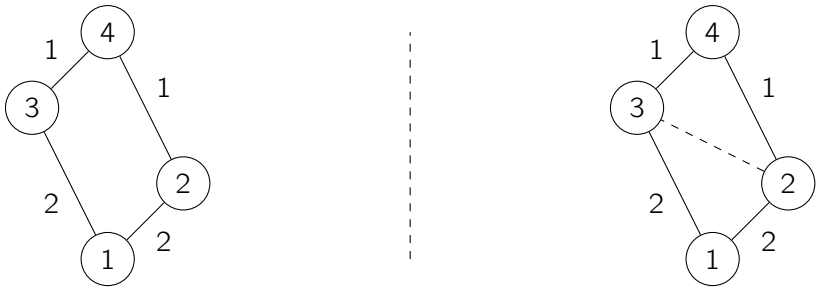


Figure 4 The left represents a CH and the right a CCH contracted graph

4.2.1 Example

Regarding Figure 2, 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.
In the example of figure 2 or, backward and forward search both reach the top vertex, so the search can stop.

4.3 Difference between CH and CCH

Looking at the left graph in Figure 4 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.

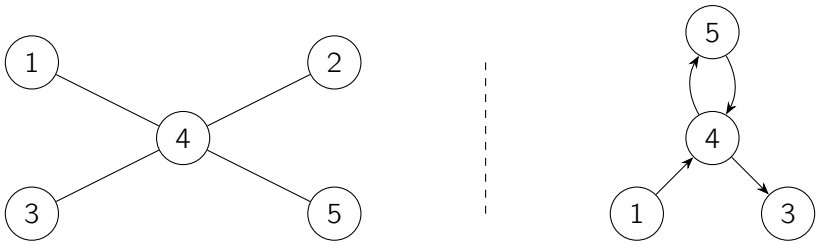


Figure 5 The numbers inside the vertices represent their contraction order

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 5, 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 5 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 2. This can be overcome by the method that is explained in section 4.5.

4.4.2 Linear Query Search Space

Regarding figure 6 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.

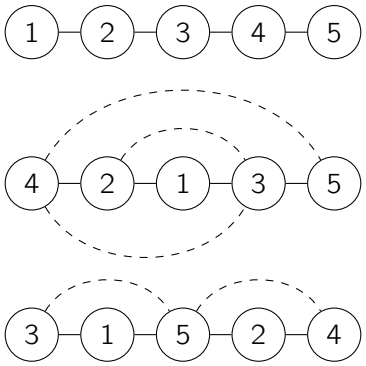


Figure 6 Linear Contraction

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:

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

Our tests show that this importance calculation 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 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 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 7, of $a(x, y)$ if there is an upper arc; denoted by c in figure 7, 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.

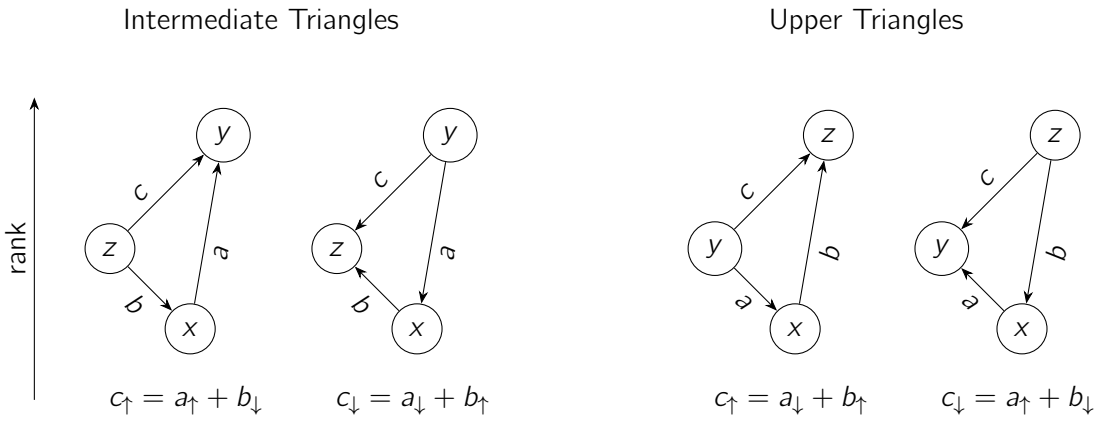


Figure 7 Update Triangles

4.6.1 Intermediate and Upper Triangles

As the explanation of intermediate and upper triangles in [DSW16, Customization Contraction Hierarchies] exist, but is hardly understandable as kept quit short, we will describe the here in every very detail.

We define an arc as its to end vertex x and 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. The arc between x and z we assign the letter b and the arc between y and z we assign the letter 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 node z is of high rank than both, $r(x) < r(y) < r(z)$.

Therefore the as shown in figure 7 follows 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 node z is of high rank than both, $r(x) < r(z) < r(y)$.

Therefore the as shown in figure 7 follows 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 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. We 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. Through that decision we get full control over the data structure so we can decide how to store it. The mapping between the index and the graph that 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 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 with this approach 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 as shown in figure 8. This also means we cannot construct them all at once, but need a function which initializes the graph, because we have a circular reference. Our solution if you want to initialize such a graph, for an id pair that represents arc or from relationships that come from neo4j is as follows: We iterate over all id pairs, create all vertices that we have not seen so far an push them to a hash table. Then create the arc and attach the vertices to it. These vertices we get from the hash table. Finally we add the arc to its start and end vertex.
A disadvantage of this model could be that some modern hardware optimization that exist for

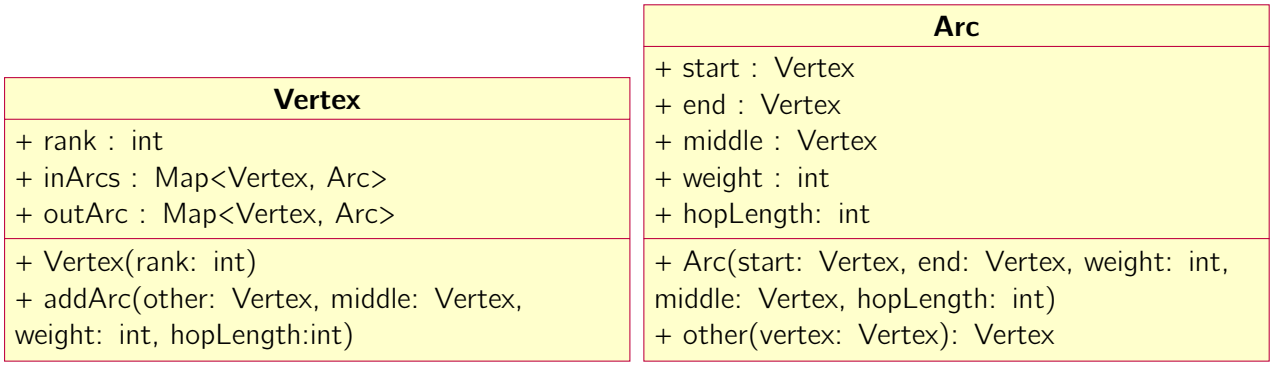
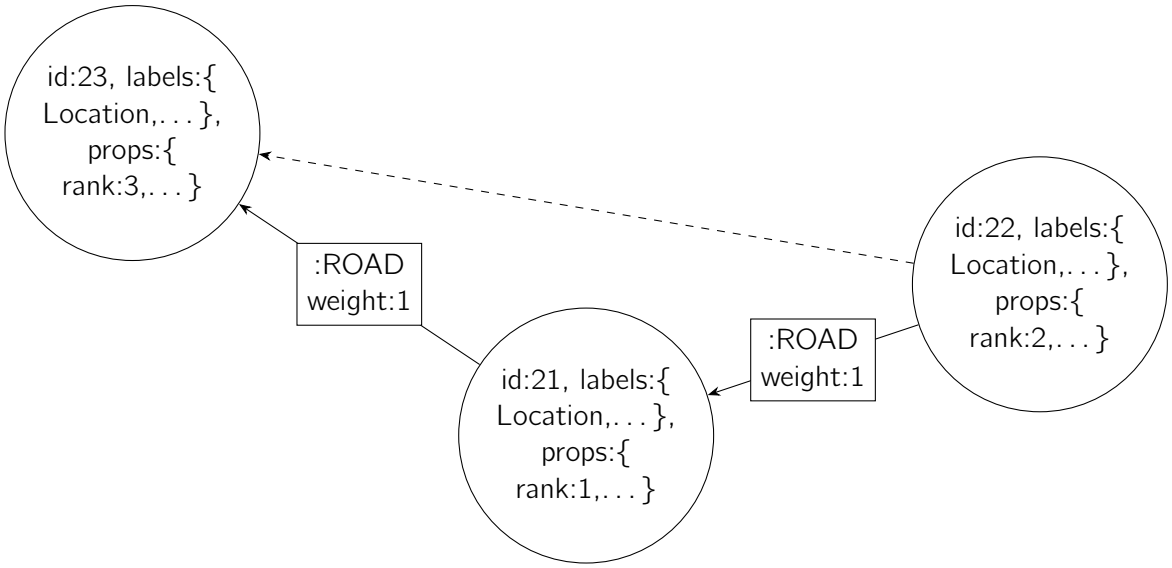


Figure 8 Index Graph



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

Figure 9 mapping

arrays do not match with this data structure. Due to paging algorithms as described in [AH15, Modern Operatins Systems] 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 apply.
However, this makes the 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. 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 *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.

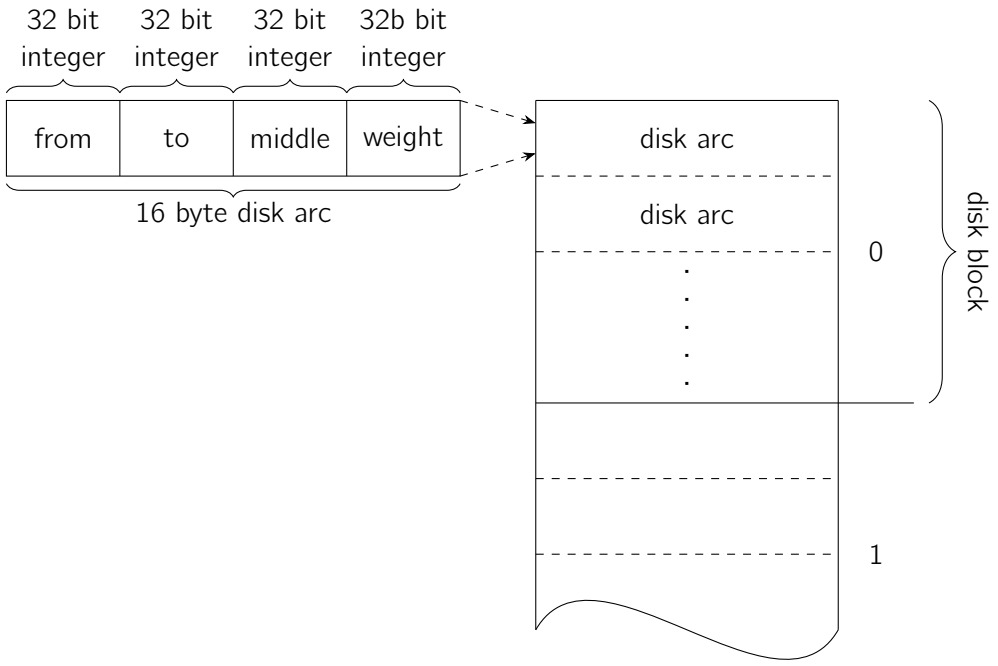


Figure 10 Disk Block

As just explained there are lot of information in this data structure. A lot of information we don't need. Looking at 9 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 index graph G', 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* as you can see in figure 10.
This 16 Byte disk arcs are collected to 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 once can gets and it would be a wast of space. The second lower bound is the maximum degree that exists in the index graph after the contraction as all arc of one vertex, as all upward arcs a vertex have to be stored int 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 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 provided the algorithm for that in algorithm 0. As you can see in figure 12, the 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 on 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 write for writing the arc and one file writer for writing the position file as shown in figure 11. Then we start the *store()* function of algorithm 0. As long as the stack is not empty we pick the top iterator of the stack, but leaf it inside. Then 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 hasn't 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 get's an iterator of this vertex neighbors sorted ascending by their rank. Finally when the algorithm stops, we write the position file to the disk and also 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.

5.4 Reading Disk Arcs

If one wants to get all upwards arcs of rank i one needs 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 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

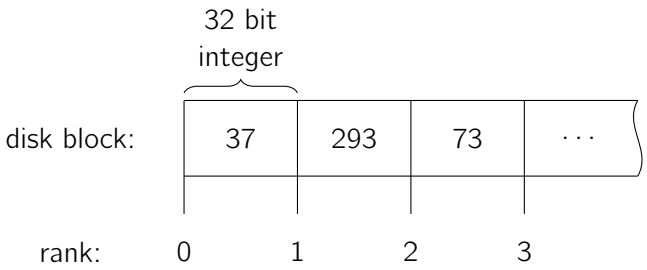


Figure 11 Position File

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 12 StoreFunction

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 upward arc set is empty $A_{\uparrow} = \emptyset$. The vertex set $V_{\downarrow}$ of the downwards graph

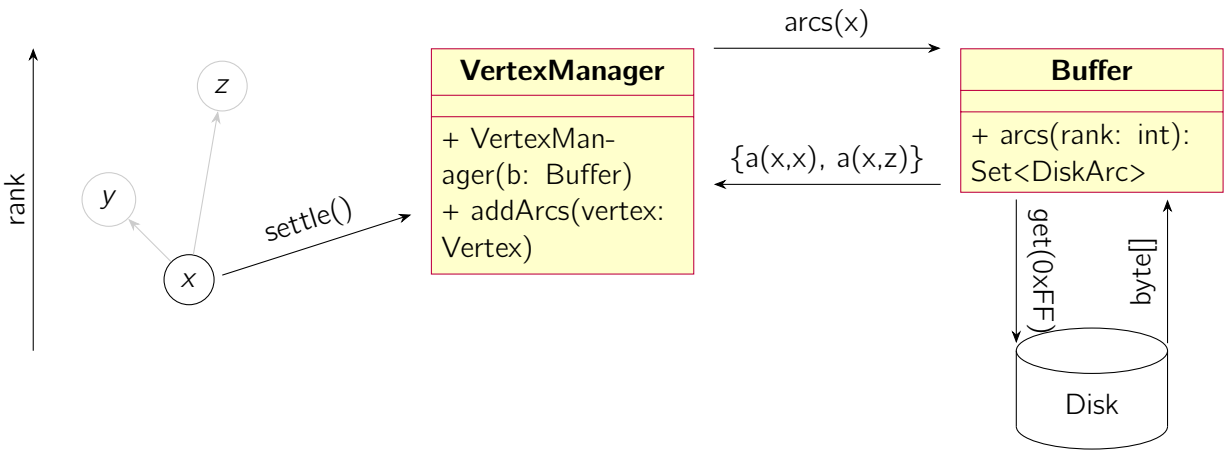


Figure 13 lazy load vertex at query time

$G'_{\downarrow}(V_{\downarrow}, A_{\downarrow})$ only contains the $v(t)$ and the arc set is empty $A_{\downarrow} = \emptyset$. Looking at graph in figure 13 visualize the search upwards search size after initialization. The ars an vertices in grey are not loaded yet.

As defined earlier in section **??**, the vertices of the dijkstra query are warped in a object called *DijkstraState* which is define in figure 1. We now extend this class by one parameter, the *VertexLoader* which you can see in figure 13. The *VertexLoader* has only one method *addArcs(vertex: Vertex)*. If you pass a vertex to this method the VertexLoader will take that vertex and attached 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* get is *DiskArc* from a *Buffer*. The buffer can be of any buffer type it only has to implement *arcs(rank: int):Set<DiskArc>* method. The *VertexManager* is in responsibility create the arcs and vertices form the *DiskArc*'s it gets. A *Buffer* contains some type of collection data structure cache *DiskArc*'s. When the vertex manger requests some arcs the *Buffer* checks whether they are already 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 13.

After starting the CH-Dijksta as described in algorithm 3 both $G'_{\uparrow}$ and $G'_{\downarrow}$ are alternatingly expanded and the vertices need will be attached when needed. 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.

CHAPTER 6

Experiments

In this chapter we will experimentally check if our idea and implementation of a persisted version of CCH works out. We will give an answer to 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 that work better than others?
- Do different buffer sizes significantly change the performance?
- Is the input graph size the only factor that affect the performance?
- How long do updates take if we change all arc weights?
- Do updates effect performance of the search?

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. The fact that our test environment uses a SDD hard drive isn't ideal. Databases usually use HDD drives which are slower in reading data but faster in writing data. However, a quick lookup on the internet shows that, as of today, in November 2023, HDD drives start at a price of around 15€/TB and have 512MB of cache. Our biggest index has 430MB in total, depending on the caching algorithm of the disk, it could be that the HDD would pre cache the whole index anyway, such that there are fewer actual reads from the durning 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 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, California+Nevada and also larger networks that represent the great lakes on the north american continent as well as the USA east cost. 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. The arc size differs from the DIMACS Challenge as we have filtered out duplicate edges.

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 6 the contraction can easily find vertices as separators.

6.3.1 Limits

We decided to set the time limit a contraction should not exceed to about one day. If, within this time the contraction did finish, we decided to abort the process. This happend for the graphs *Western USA*, therefore we didn't try larger ones. 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 graph that *Great Lake* turned complete with 1078 nodes left in the queue. at this time it toke about 110 second to contract a single vertex, and here comes why. The algorithm for the contraction 2 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 $(|Q|)^3$ calls to the *getContraction()* method in algorithm 2, to 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.
Our test show that as if the number of neighbors that are not contracted yet and there need to be updated rise above 150, the time to 500ms to contract a single vertex.

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.
We do 10000 point to point shortest path queries, where the start and the end vertex of the previous are always different to the one under test. This is because if you always start from the same vertex or query the same target, at least one of the buffers has most probably already the right edges in cache. This is desirable but doesn't 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

	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	23.29h
$max(d(v))$	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	1.8MB	1.3MB	7.6 MB	11.1 MB	14.4MB
$\|\bigcirc A_{\downarrow}\|$	23.1 MB	21.9 MB	56.9 MB	107 MB	201 MB	215 MB
pos-file$_{\downarrow}$	1.1 MB	1.8MB	1.3MB	7.6 MB	11.1 MB	14.4MB
$t_{dijkstra}$	0.816 s	0.549 s	2.630 s	4.858 s	5.425 s	5.387 s
t_{cch}^{40kB}	0.140 s	0.122 s	0.147 s	0.289 s	0.732 s	0.727 s
I/O (40kB)	574	437	500	899	1671	1572
t_{update}	90 s	51 s	142 s	444 s	1827s	1557s
$t_{cch-updated}^{40kB}$	0.147 s	0.129 s	0.150 s	0.302 s	0.783 s	0.855 s
$I/O_{cch-upd.}^{40kB}$	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

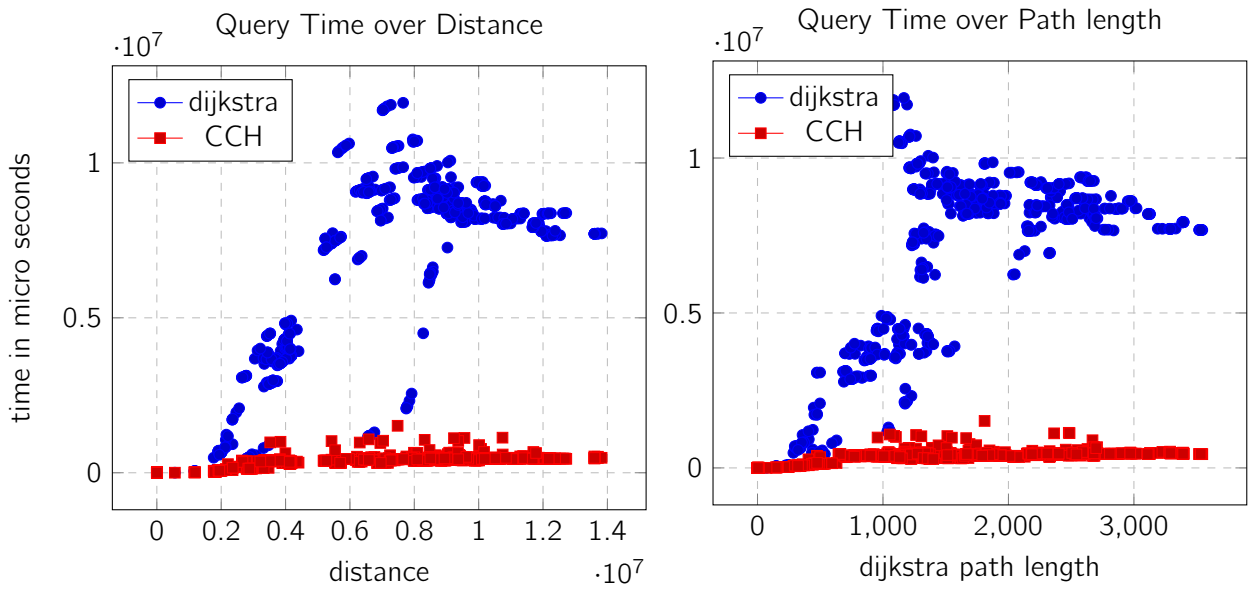


Figure 14 Comparison of CCH query performance on the California+Nevada graph, before updating. Buffer size 40kB. Random sample of 1000 vertex to vertex queries

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.

One of the big questions can we even beat dijkstra's performance. As you see 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 that represents Florida the query performance improvement was tremendous. An average query time could be reduces from 2.6 seconds down to 0.039 second 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 did sometimes lead to better results, as for California+Nevada. Here we could decrease the query time by half. For some other graphs it didn't bring any improvement as for Florida. The others had some improvement, in the region of about 20%. A 20% caching strategy though could be something that is useful if you have an application that often queries the same vertex. With t_{cch}^{40kB} and $t_{cch-updated}^{40kB}$ we also tested the performance after updates have happened. There was very little change in that. All queries got slightly worse but didn't loose much.

6.4.1 Short and Long Distance Queries

Regarding Figure 14 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 14 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. This is underling our theory as there are still some longer queries that perform well but now there are closer together.

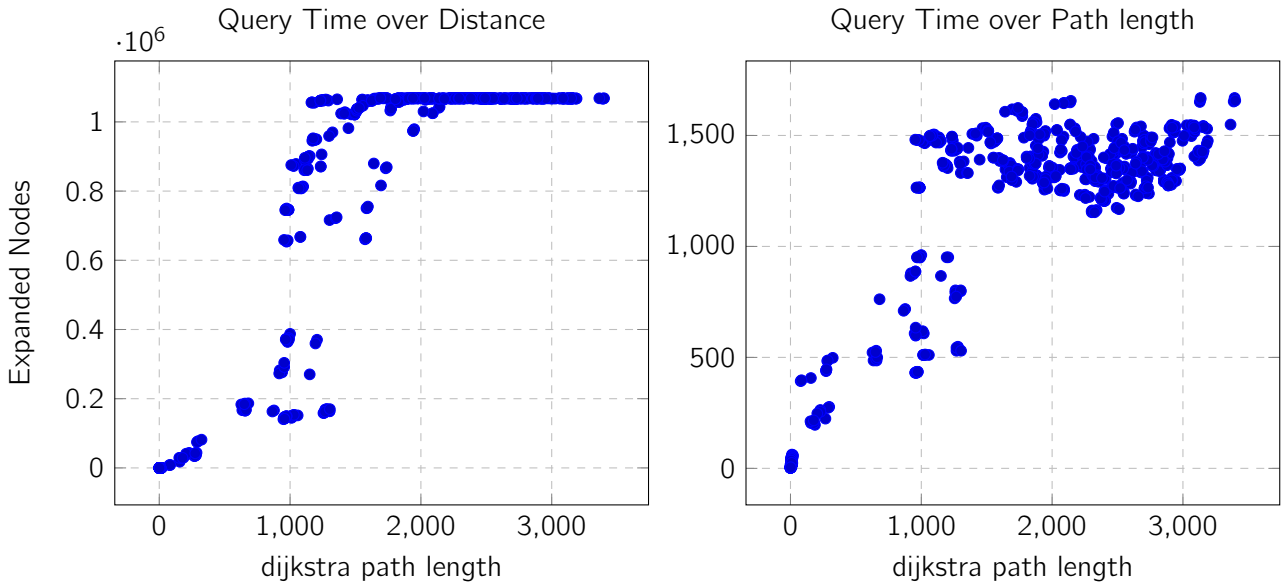


Figure 15 Comparison of CCH query performance on the California+Nevada graph, before updating. Buffer size 40kB. Random sample of 1000 vertex to vertex queries

6.4.2 Query Analyses

Having a look a figure 15 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 15 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.

6.5 IO's at Query Time

As we did our implementation to show that CCH could also fit for a graph database it is essential to have a look 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 that can happen is that there is one I/O per expanded vertex. As you can see in figure 16 we can do better. With a cache size of only 640 kB which gives the possibility to hold 40960 acs in cache we already achieve around 1.4 expanded vertices per I/O. This means in a bit less than every second we accedently already 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.
For bigger network as the *Great Lakes* and *Eastern USA* this cache size was to small. In most scenarios we had about as many I/O's as expanded nodes. So we will have to go bigger.

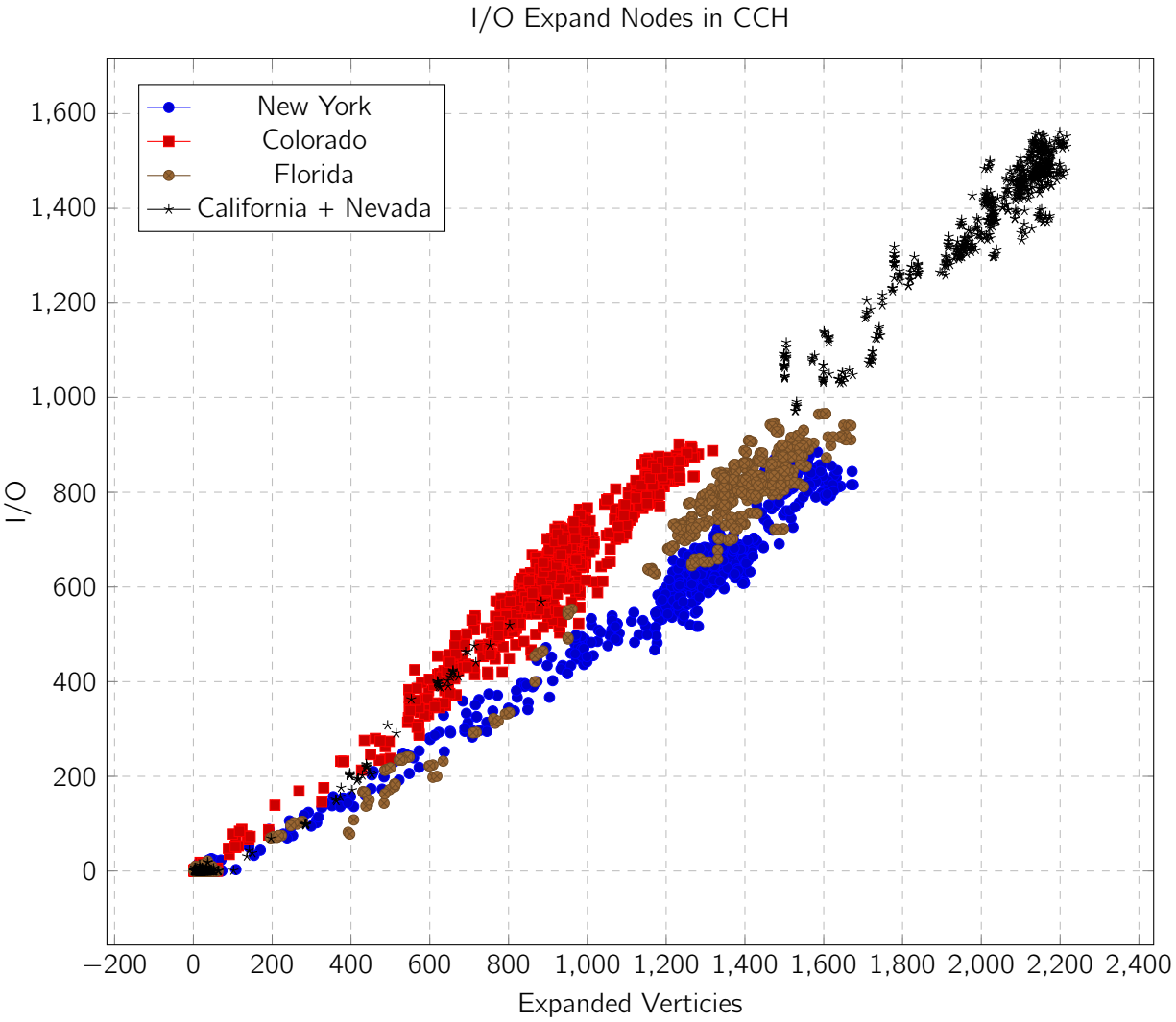


Figure 16 I/O's over Expanded Vertices for 40kB cache size per search side.

CHAPTER 7

Future Work

With this work we barely scratched the surface of what can be done with CCH in neo4j. There are a lot of special topics one could dive deeper to improve CCH for neo4j.

7.1 Contraction Algorithms

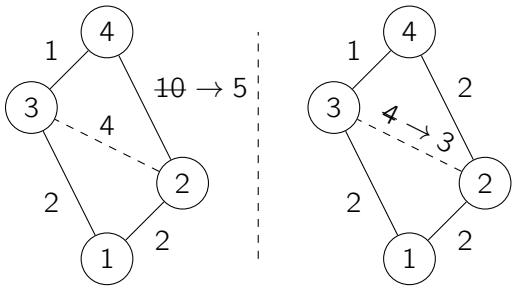
As the contraction algorithm 2 we use definitely has its limitation as we have seen in the experiment section 6.3.1, it is worth digging deeper there. One very promising method determine the vertex by recursively look for minimum balanced separators until they tend to get big and then continue with our algorithm. Another question we haven't even touched is, what if we add vertices and arcs. How can we deal with such updates in the input graph which are not unusual for databases.

7.2 The Test Dataset

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

7.3 Perfect Customization and Transition to CH

We didn't implement the perfect customization, nor the transition from CCH to CH. A CH index is usually faster at query time as the amount of arcs 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 long such a transition takes.



7.4 Using relational Databases

Furthermore it is questionable if graph databases like neo4J solve any problem. Due to [paper] 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

to be written

Bibliography

[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.

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

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

[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 Confer-*

ence on Artificial Intelligence, pages 236–247. Springer, 2013.

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