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 $w : A \rightarrow \mathbb{R}_{<0}$ 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.

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 and Searching

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.

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.

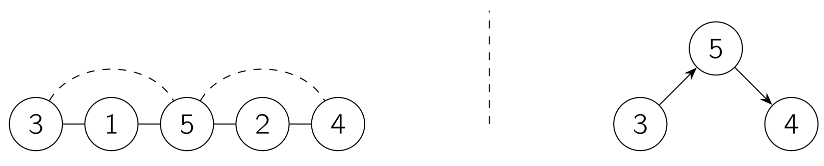


Figure 1 The numbers inside the vertices represent their contraction order

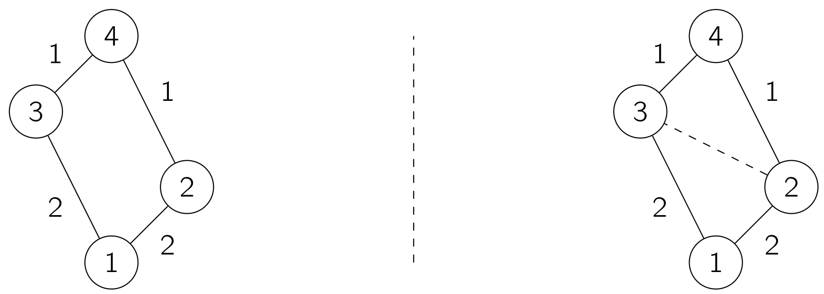


Figure 2 The left represents a CH and the right a CCH contracted graph

4.2 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.

4.3 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.

4.3.1 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

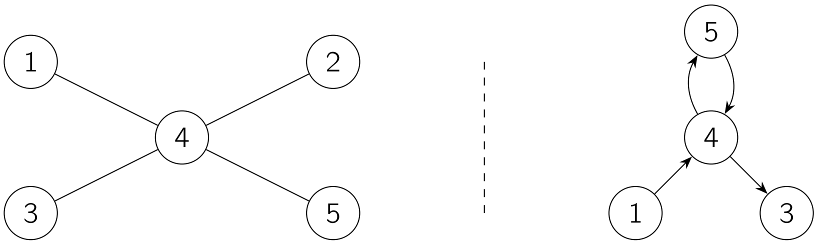


Figure 3 The numbers inside the vertices represent their contraction order

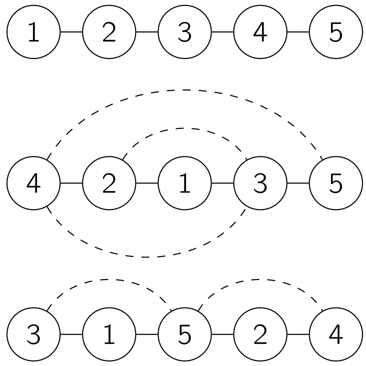


Figure 4 Linear Contraction

$ED = -1$. 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 $e(v(1), v(2))$, $e(v(1), v(3))$, $e(v(1), v(5))$, $e(v(2), v(3))$, $e(v($ 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 4.4.

4.3.2 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.
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.

4.4 Vertex importance

As shown in section 4.3.1 and 4.3.2 there are vertices that are more important that other vertices. Contracting these vertices late is key to get a efficient search later on.

4.4.1 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].

4.4.2 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.

4.5 Update CCH

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

Algorithm 1 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
```

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.

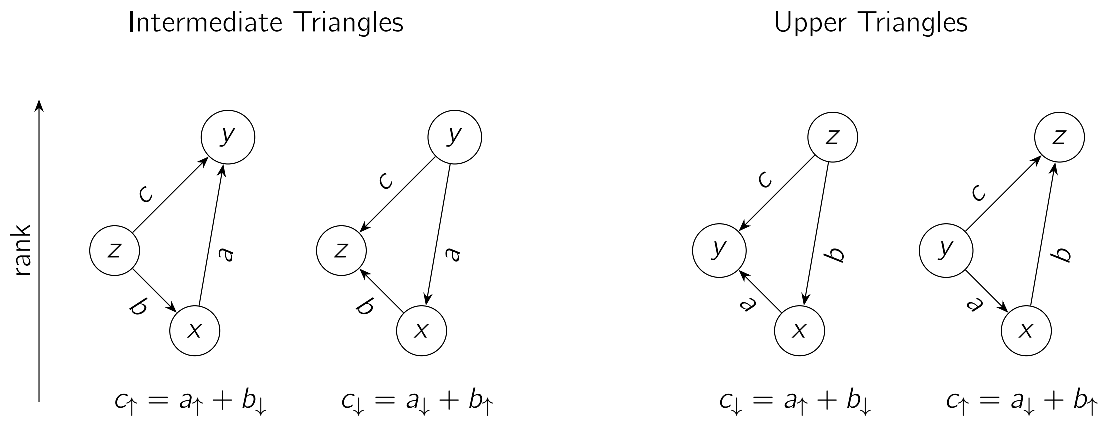


Figure 5 Update Triangles

Algorithm 2 Compute Triangles

```
procedure updateTriangles(arc, neighbors, predicate)
    triangles ← {}; x ← min(arc.start, arc.end); y ← max(arc.start, arc.end);
    for all z in neighbors do
        if predicate then
            if upwards(arc) then triangles.add(Triangle(a(x, y), a(z, x), a(z, y)))
            else triangles.add(Triangle(a(x, y), a(x, z), a(y, z)))
            end if
        end if
    end for
    return triangles
end procedure
procedure intermediateTriangles(arc)
    neighbors ← N↑(arc.start) ∩ N↓(arc.end)
    return updateTriangles(arc, neighbors, x < z < y )
end procedure
procedure upperTriangles(arc, neighbors)
    neighbors ← N↑(arc.start) ∩ N↓(arc.end)
    return updateTriangles(arc, neighbors, x < y < z )
end procedure
```

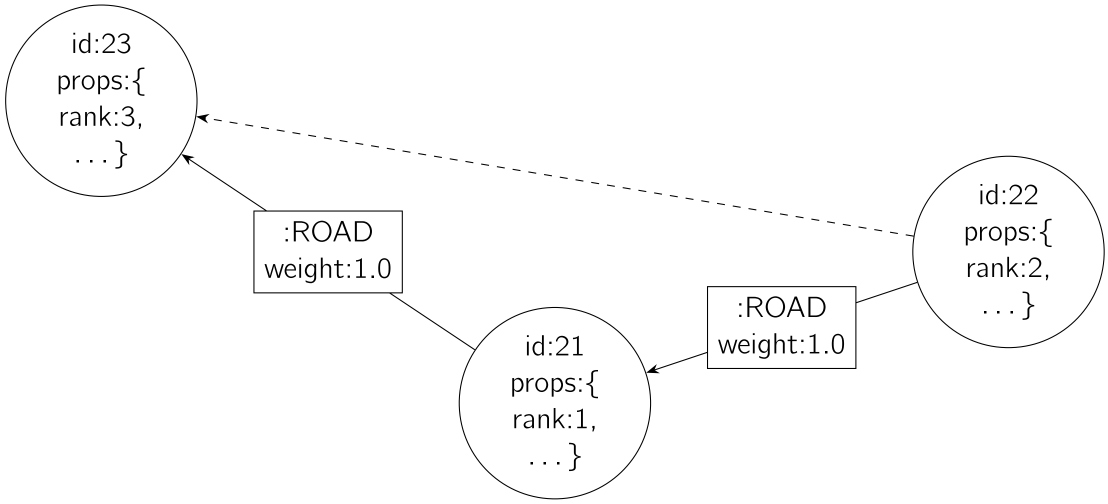
CHAPTER 5

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.



from	to	middle	weight
2	3	1	2.0
1	3	-1	1.0
2	1	-1	1.0

Figure 6 mapping

5.2 The mapping

5.3 The Contraction

5.4 How to Store the Index Graph

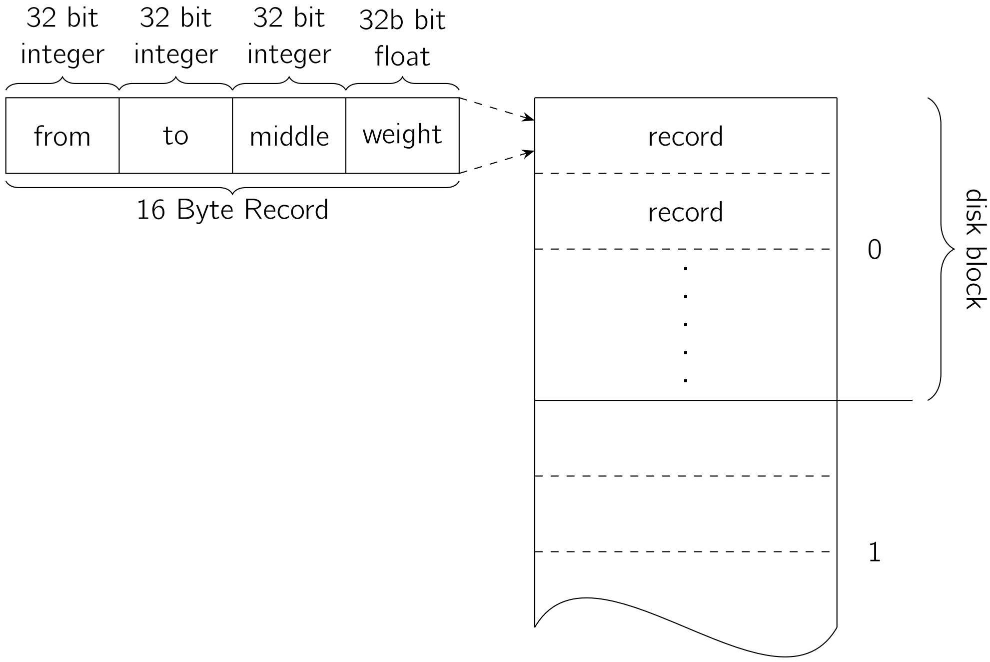


Figure 7 Disk Block

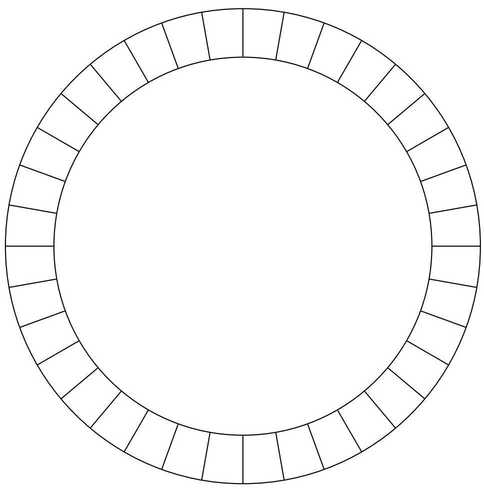


Figure 8 Circular Buffer

CHAPTER 6

Experiments

some experiments

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.

[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.