MSc Thesis

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

1 Introduction **4**

2 Related Work **5**

2.1 Algorithmic History 5

2.2 Contraction Hierarchies Database History 5

3 Preliminary **7**

3.1 Notation and Expressions 7

3.2 Customizable Contraction Hierarchies 7

4 Integration in a Neo4j **8**

4.1 Index Graph Data Structure 8

4.2 The Contraction 8

4.2.1 Metric Dependent Order 9

4.2.1.1 Important Vertices not contracted last 9

4.2.1.2 Linear Query Search Space 9

4.2.2 Vertex importance 10

4.3 How to Store the Index Graph 10

5 Important Algorithms **11**

5.1 Updating a Priority Queue 11

5.2 Search Algorithm 11

5.3 Triangles 12

Bibliography **14**

CHAPTER 1

Introduction

intro

CHAPTER 2

Related Work

As this is manly a database paper, we want to divide this chapter in two main sections Algorithmic History and Contraction Hierarchies Database History. Algorithmic History will give some basic overview what has been published regarding index structures to speed up shortest path queries for graphs. Contraction Hierarchies Database History 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 heavly influenced by the idea of the [BFSS07, Transit-Node] approach and as transit node approach itself is a technique to speed up the [Dij59, Dijkstra 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 vertex inside the graph that are more important than others. Important in this case means, these are vertices that lie 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 because 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 add edges to the graph , so called shortcut, that preserve the shortest path property of the graph in case a vertex that is contracted lies on a shortest path between other. 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 violtes the shortest, 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 CH an index structure is also worth pursuing in a database context, as the speedup of shortest path queries paired with a reasonable preprocessing time. [Zic21] wrote a 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 as it was deeply integrated into the Neo4j-Platform and since that there have been two major release updates with 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.

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 rare cases we might refer to *edges*. There you can be sure that it doesn't matter weather it is directed or not.

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 e is an edge $a \epsilon A$. 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 $w : E \to \mathbb{R}_{<0}$ it's weight which must be a positive.
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 $\bigcirc E$ will be separated into to sets $\bigcirc E_\downarrow$ and $\bigcirc E_\uparrow$, too.

3.2 Customizable Contraction Hierarchies

CHAPTER 4

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.

4.1 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.

4.2 The Contraction

There are two way 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

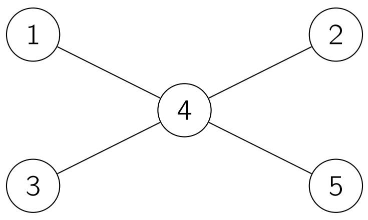

Figure 1 The numbers inside the vertices represent their contraction order

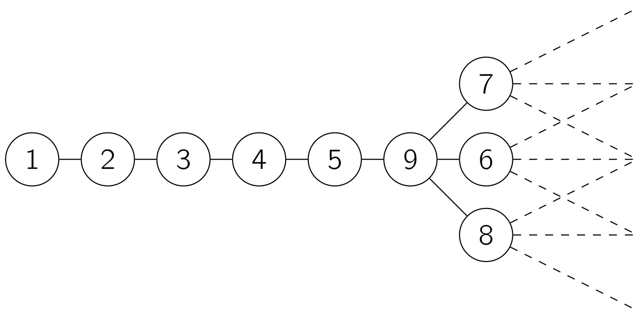

Figure 2 Linear Contraction

optimizing the contraction process.

4.2.1 Metric Dependent Order

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. 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 other. 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.

4.2.1.1 Important Vertices not contracted last

Looking at figure 1, this is a possible contraction order, if only the ED is used to contract vertices. At the beginning the nodes with rank 1, 2, 3, 5 have the same edge difference, which is $ED = -1$. One edge will be removed after contraction and the is no shortcut inserted. This happens until there are only the vertices 4 and 5 left. Now vertex 4 has an $ED = -1$, too, same as vertex 5. Therefore the algorithm contracts the vertex with rank 4 before the one with rank 5.
However this is not the desired result. There are six $(1,2), (1,3), (1,5), (2,3), (2,5), (3,5)$ shortest paths that involve vertex 4, all the other vertices do not encode any shortest path, so vertex 4 should be contracted last. This can be overcome by the method that is explained in section 4.2.2

4.2.1.2 Linear Query Search Space

Looking at figure 2 this is a possible contraction order using the only the ED as method to determine the importance of a vertex. As you can see vertex 9 is the one with the highest importance. Which is the desire result as all shortest path come from vertices $1, 2, 3, 4, 5$ and

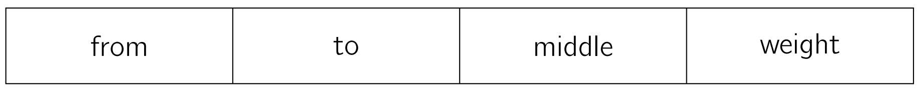

Figure 3 Single Record

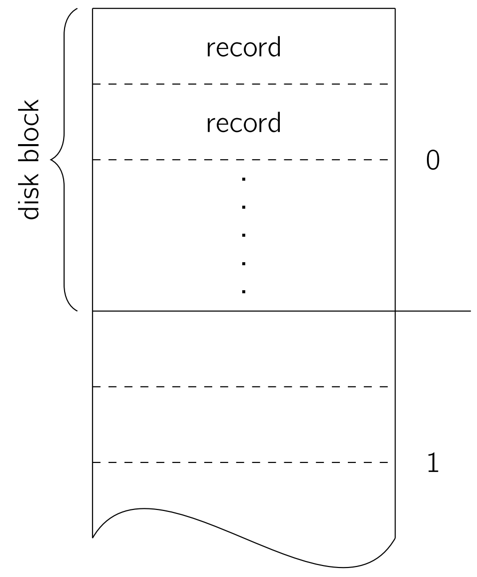

Figure 4 Disk Block

go any of the vertices $6, 7, 8$ have to pass through vertex 9. However regarding only vertices $1, 2, 3, 4, 5$, there is only one way to go when looking for the shortest path. And if we only go left vertex by vertex, there is no difference between our search or a Dijkstra search. Which means, no gain in query performance. Better would be to contract vertex 3 before vertex 2. This would insert one shortcut more which is bearable in sparse areas, but it decreases the hops that need to be done by one as you insert an arc between vertex 2 and 4, so you can surpass vertex 3.

4.2.2 Vertex importance

As shown in section 4.2.1.1 and 4.2.1.2 using only the ED as a metric to determine which vertex to contract next is not sufficient to get a suitable

4.3 How to Store the Index Graph

CHAPTER 5

Important Algorithms

5.1 Updating a Priority Queue

5.2 Search Algorithm

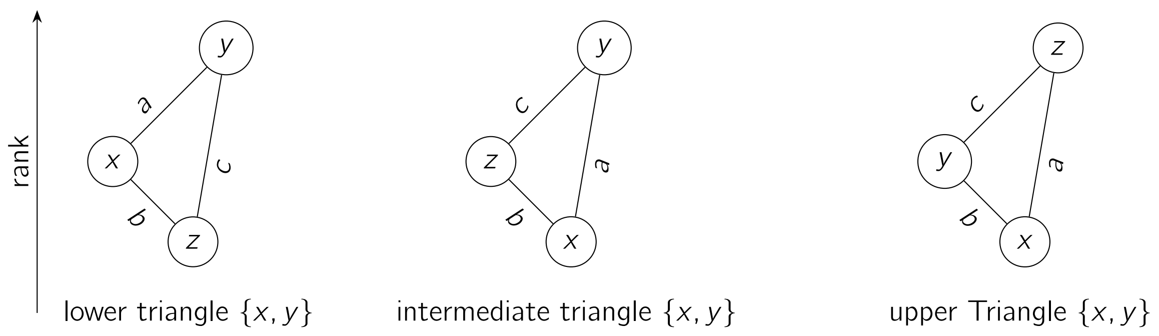
rank
y
a
x
c
b
z
lower triangle $\{x, y\}$
y
c
z
a
b
x
intermediate triangle $\{x, y\}$
z
c
y
a
b
x
upper Triangle $\{x, y\}$

Figure 5 Triangle Enumeration

5.3 Triangles

Algorithm 1 Compute Triangles

```
procedure lowerTriangles($arc$, $neighbors$)
    triangles ← {}; x ← min(arc.start, arc.end); y ← max(arc.start, arc.end);
    for all $z$ in $neighbors$ do
        if $z < x < y$ then
            if upwards($arc$) then triangles.add(Triangle($arc$, $x$.getArcTo($z$), $z$.getArcTo($y$)))
            else triangles.add(Triangle($arc$, $z$.getArcTo($x$), $y$.getArcTo($z$)))
            end if
        end if
    end for
    return triangles
end procedure
procedure intermediateTriangles($arc$, $neighbors$)
    triangles ← {}; x ← min(arc.start, arc.end); y ← max(arc.start, arc.end);
    for all $z$ in $neighbors$ do
        if $x < z < y$ then
            if upwards($arc$) then triangles.add(Triangle($arc$, $z$.getArcTo($x$), $z$.getArcTo($y$)))
            else triangles.add(Triangle($arc$, $x$.getArcTo($z$), $y$.getArcTo($z$)))
            end if
        end if
    end for
    return triangles
end procedure
procedure upperTriangles($arc$, $neighbors$)
    triangles ← {}; x ← min(arc.start, arc.end); y ← max(arc.start, arc.end);
    for all $z$ in $neighbors$ do
        if $x < y < z$ then
            if upwards($arc$) then triangles.add(Triangle($arc$, $z$.getArcTo($x$), $z$.getArcTo($y$)))
            else triangles.add(Triangle($arc$, $x$.getArcTo($z$), $y$.getArcTo($z$)))
            end if
        end if
    end for
    return triangles
end procedure
```

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.

[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.

[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.