Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Computing Shortest Path Distances based on Cluster-Pairs

Lukas Epple

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Prof. Dr. Stefan Funke

Commenced: January 15, 2021

Completed: June 15, 2021

Abstract

In this thesis we present a new lookup based shortest path distance computation scheme. Like many other lookup based schemes, this new approach consists of two stages, a preprocessing and a query stage.
After the properties, which need to be met by the results of the preprocessing, are formally defined, we propose different preprocessing procedures as well as procedures which are able to use those results to calculate shortest path distances.
The proposed new query procedure is able to decide whether a given query can be answered when the preprocessed data is incomplete.
This makes it possible to prune the preprocessed data according to arbitrary memory constraints, while a speed up of orders of magnitude in comparison to conventional techniques can be achieved.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Various techniques and algorithms have been developed to solve shortest path distance problems, and while Dijkstra's algorithm [Dij59] can be used to solve the problem, a lot of speed up techniques have been developed to address the problem of answering a large numbers of shortest path distance queries in a short time.
Most of these techniques make use of the fact that the graph is known beforehand and can be transformed in a way to later speedup shortest path distance queries.
While the simplest lookup based technique is to precompute all shortest path distances of a graph using Dijkstra's algorithm and storing them in a lookup table, it is not applicable for large graphs because of the huge memory consumption of the lookup table. Because of this, more sophisticated lookup based techniques, such as Hub Labels [ADGW11] or Transit Nodes [BFSS07] have been developed. Those proposed schemes use more ingenious techniques to achieve fast shortest path distance queries, while only using a reasonable amount of memory.

In this thesis a new such technique based on cluster-pairs is proposed, which also consists of a preprocessing and a query stage. In the beginning, the idea was to find a scheme for grid graphs, which makes use of the idea of well separated pair decompositions, but then a more efficient technique which is also applicable to general graphs was discovered. Unlike Hub Labels, this new technique, which is based on cluster-pairs, is able to arbitrarily prune the information stored per node, while still calculating correct shortest path distances if possible. While the proposed preprocessing procedure requires a high memory consumption, the query routine can compete with even the fastest state of the art techniques. Similar to Hub Labels, the proposed preprocessing scheme stores information per node, in order to answer the shortest path distance queries. Thus for nodes s and t, only the information stored at those nodes is required. As a result, the original graph will only be needed during the preprocessing stage and therefore can be discarded once the preprocessing is done, if necessary.

In the second chapter, the theoretical fundamentals, such as `graphs`, `grid graphs` or `shortest paths` are formally defined. The chapter also introduces existing methods to calculate shortest path distances, as well as some speed up techniques.
The third chapter contains the newly developed cluster-pair based shortest path distance calculation methods. First a scheme which only works for grid graphs is described. Then a generalized technique which works for all graphs is introduced, making the restricted scheme for grid graphs obsolete.
In the fourth chapter, benchmarks of different versions of the proposed algorithms are compared to each other and to Hub Labels [DGSW14] in terms of runtime, success rate and memory usage. The last chapter summarizes the thesis and gives some ideas for further optimizations and improvements of the proposed shortest path distance calculation schemes.

2 Basics

In this chapter different concepts and algorithms are presented, which are needed to understand and compare the algorithms proposed in this work. At first graphs as well as grid graphs are formally defined, then the shortest path distance problem, as well as different algorithms to solve it, are presented.

2.1 Graphs

A **graph** G is defined as a tuple $G = (V, E)$ where V is a set of nodes and E is the set of edges.

For **directed graphs** $E \subseteq V \times V$, which means edges connecting nodes have a direction.

Edges of **undirected graphs** do not have such a direction, and therefore $E \subseteq \binom{V}{2}$
Since road networks do contain one-way roads most of them are modeled as directed graphs. It is also worth mentioning that undirected graphs can be easily represented as directed graphs by the addition of two edges (u, v) and (v, u) for an undirected edge $\{u, v\}$.

A **weighted graph** is a graph with a cost function $c : E \longrightarrow \mathbb{R}$. For road networks the cost function is often used as a metric for the distance which is associated with an edge. Because of this road networks are often modeled as directed weighted graphs. In this thesis only weighted graphs with positive edge weights and therefore only cost functions $c : E \longrightarrow \mathbb{R}^+$ are considered.

A **simple path** between two nodes $s, t \in V$ is defined as a sequence of pairwise disjoint nodes $x_1, \ldots x_i$ with $x_1 = s$ and $x_i = t$, such that $\forall 1 < k \leq i : (x_{k-1}, x_k) \in E$.
A **shortest path** between two nodes $s, t \in V$ in a weighted graph $G = (V, E)$ with a cost function $c : E \longrightarrow \mathbb{R}^+$ is defined as the simple path π between s and t where $\sum_{e \in \pi} c(e)$ is minimal.

Figure 2.1 shows a visual representation of a directed graph $G = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (2, 3), (2, 4), (3, 5), (5, 2), (5, 4)\}$. A valid simple path between node 1 and 5 would be $1, 2, 3, 5$.

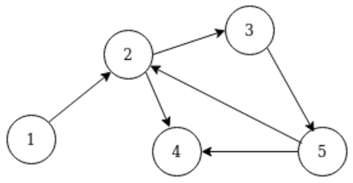

Figure 2.1: Example of a directed graph

2.2 Grid Graphs

A grid graph is a model of a $m \times n$ grid as a graph. Given a $m \times n$ grid, every cell (i, j) is represented as a node $v \in V$. Two nodes can only be connected by an edge, if the cell which they represent are direct neighbors in the grid. This means $((i_s, j_s), (i_t, j_t)) = (s, t) = e \in E \Rightarrow |i_s - i_t| + |j_s - j_t| = 1$. Grid graphs also exist for other types of grids, like for example triangular grids, but in the following only grid graphs for square grids are considered. Grid graphs are often used to model maps in computer games, such as Baludurs Gate or Dragon Age: Origins [Stu12]. Figure 2.2 shows an example of such a map used in the computer game Balurs Gate II. The white cells of the grid are represented by nodes in the graph, which are connected to the nodes representing the neighbor cells if they are also white in the grid. In such a case the white cells of a grid are referred to as *walkable*.

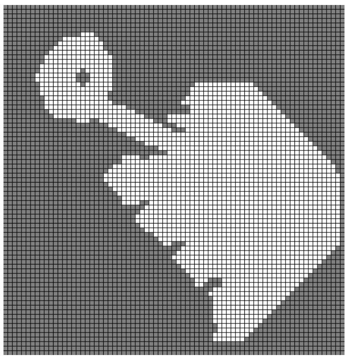

Figure 2.2: Example grid graph used in Baldurs Gate II [Stu12]

2.3 Shortest Path Distances

The shortest path distance between two nodes s and t in a graph $G = (V, E)$ with cost function $c : E \longrightarrow \mathbb{R}^+$ is defined as $\sum_{e \in \pi_{opt}} c(e)$ where π_{opt} is a shortest path between s and t in graph G and cost function c. To calculate the shortest path distance between two given nodes s and t various approaches have been developed.

Exploration based shortest path distance calculation

An algorithm which works for graphs without negative edge weights is Dijkstra's algorithm [Dij59]. The algorithm starts from the source and explores the graph, always following the most promising candidate nodes until the target node itself is the most promising candidate and the cost of the shortest path from the source to the target node can be reported.

```
Algorithm 1: Dijkstra
Data: Graph $G(V, E), s, t \in V$, edge cost function $c$
Result: shortest path distance between $s$ and $t$
for $v \in V$ do
    $cost[v] = \infty$
// Q is a min-heap where the top node has always the smallest value in the $cost$ map;
$minHeap\ Q$;
$Q \leftarrow s$;
$cost[s] = 0$;
while $\neg Q.empty$ do
    $current \leftarrow Q.pop$;
    if $current = t$ then
        return $cost[current]$
    for $v \in neig(current)$ do
        if $cost[current] + c((current, v)) < cost[v]$ then
            $cost[v] = cost[current] + c((current, v))$;
            if $v \notin Q$ then
                $Q \leftarrow v$;
    $Q.reheap()$;
return $\infty$
```

Algorithm 2.1: Dijkstra's Algorithm

To further speedup the Dijkstra's algorithm Contraction Hierarchies [GSSD08] can be used. In order to gain a speedup, the shortest path distance computation is split into two stages:

1. **Preprocessing:** In this stage, the new shortcut edges E' are added to the graph $G = (V, E)$ and a level function $l : V \longrightarrow \mathbb{N}$ is introduced, such that for every two nodes s and t, there exists a shortest path $x_0, \ldots, x_i$ from s to t, such that $l(x_0) < \ldots < l(x_m) > \ldots > l(x_i)$ where $m = \text{argmax}_{0 \leq k \leq i}\, l(x_k)$ and therefore x_m is the node with the highest level on the shortest path from s to t.
2. **Query:** For a given source and target, two modified versions of Dijkstra's algorithm are performed, one starting from the source another starting from the target. The one which starts from the target is performed on a graph where all edges $(u, v) \in E$ are reversed. Those modified versions do only consider edges $(u, v) \in E$ where $l(u) < l(v)$.

Since the modified version of Dijkstra's algorithm only considers edges $(u, v) \in E$ where $l(u) < l(v)$, the search space is much smaller for road networks, which is why Contraction Hierarchies gain a speedup in comparison to Dijkstra's algorithm. The idea of splitting the shortest path distance computation into a preprocessing and query stage is used by multiple other speedup techniques including the one proposed in this thesis.

Lookup based shortest path distance calculation

Another approach to calculate shortest path distances are lookup based methods. The simplest lookup based method would be to calculate shortest path distances for all $(s, t) \in V \times V$ to store them in a lookup table. For this purpose, Dijkstra's algorithm can be used. While this would result in $O(1)$ queries, the preprocessing time as well as the memory overhead would be unreasonable for big graphs. Different other lookup based techniques were developed which do need much less memory and preprocessing time, like Transit node routing [BFSS07] or Hub Labels [DGSW14].

The basic idea of hub labels is to calculate labels $L(v)$ for every node $v \in V$ such that for two given nodes s and t only the labels $L(s)$ and $L(t)$ need to be inspected. Once such labels are computed for every node, the graph is not needed anymore to answer shortest path distance queries. To achieve this the labels $L(v)$ of a node $v \in V$ have the form:

$$L(v) = \{(h, d(v, h)) : h \in H(v)\}$$

where:

$H(v)$ is a set of important nodes for v called **hubs**.

$d(v, h)$ is the optimal shortest path distance between nodes v and hub h.

Those labels need to be computed such that for any pair $(s, t) \in V \times V$ there exists $(h_s, d_{s,h}) \in L(s)$ and $(h_t, d_{t,h}) \in L(t)$ such that $h_t = h_s$ and $d_{s,h} + d_{t,h} = d(s, t)$.

Because of the fact that the calculation of the shortest path distance between two nodes s and t can be achieved by finding a label $l \in L(s) \cap L(t)$, which minimizes the distance between s and t, only $L(s)$ and $L(t)$ are needed. If such labels are given from a preprocessing stage, the labels for each node are stored sorted lexicographically. For labels stored in such a way Algorithm 2.2 can be used to answer shortest path distance queries for nodes s and t. Since $|L(s) \cap L(t)|$ could be greater than 1, all the common labels need to be looked at and the one which minimizes the distance between s and t can be used to report shortest path distance. This means that a shortest path distance for s and t can be answered in $O(|L(s)| + |L(t)|)$

Algorithm 2: HubLabelQuery

```
Data: Source $s$, Target $t$
Result: $d_{opt}(s,t)$ or UNREACHABLE
$best$ = UNREACHABLE;
$idx_s = 0$;
$idx_t = 0$;
while $idx_s < |L(s)| \wedge idx_t < |L(t)|$ do
    $(h_s, d_{s,h})s = L(s)[idx_s]$;
    $(h_t, d_{t,h})s = L(t)[idx_t]$;
    if $h_s = h_t \wedge best > d_{s,h} + d_{t,h}$ then
        $best = d_{s,h} + d_{t,h}$;
        $idx_s = idx_s + 1$;
        $idx_t = idx_t + 1$;
    else if $h_s < h_t$ then
        $idx_s = idx_s + 1$;
    else
        $idx_t = idx_t + 1$;
return $best$
```

Algorithm 2.2: Hub Label Query

3 Cluster-Pair based Distance Calculation

The basic idea of cluster-pair based distance calculation is to separate a given graph $G = (V, E)$ into a set of cluster pairs $\mathbb{S} = \{(A_i, B_i) \mid i \in \mathbb{N}\}$, such that for each pair $(u, v) \in V \times V$ with $u \neq v$ there exists a pair $(A, B) \in \mathbb{S}$ with $a \in A \wedge b \in B$.
We say a pair $(A, B) \in \mathbb{S}$ answers a pair of nodes $(u, v) \in V \times V$ iff $u \in A$ and $v \in B$.
Furthermore, given a pair $(A, B) \in \mathbb{S}$, it should be easy to report the optimal distances for paths from nodes $u \in A$ to $v \in B$. We call such a pair $(A, B) \in \mathbb{S}$ a separation and the sets A and B node patches.

When given such a set $\mathbb{S}$, calculating the shortest path distance between two given nodes s and t, boils down to finding a separation $(A, B) \in \mathbb{S}$ answering (s, t) and reporting the distance of an optimal path from s to t.

This means the challenge of finding the optimal distances for a grid graph fast breaks down into two sub-challenges:

- **Preprocessing:** calculating set $\mathbb{S}$ with the given properties and a way of reporting the distances of the optimal ways between nodes of A and B.
- **Query:** for a given pair $u, v \in V$, finding a set $a \in A \wedge b \in B$ or $b \in A \wedge a \in B$ must be fast.

In the following different approaches for the preprocessing and query phase are presented.

3.1 Cluster Pairs for Grid Graphs

3.1.1 Well Separated Pair Decomposition

The idea of node separations comes from the well-separated pair decomposition, which organizes a given point set P such that the $\Theta(n^2)$ pairwise distances are encoded into a structure of only size $O(n)$ [CK95].

In order to achieve this, a quadtree is built for the point set P. This quadtree can then be used with algorithm 3.1 to calculate the well separated pair decomposition. To achieve this algorithm 3.1 needs to be initially called with the root of the quadtree as input.

```
Algorithm 3: SeparateWSPD
Data: QuadTree Nodes $X, Y$
Result: Cluster Set $\mathbb{S}$
if $X = Y$ then
    return $\emptyset$
if $size(X) < size(Y)$ then
    return $SeparateWSPD(Y, X)$
if $areWellSeparated(X, Y)$ then
    return $\{(X, Y)\}$
return $\bigcup_{C \in child(X)} SeparateWSPD(C, Y)$
```

Algorithm 3.1: Calculation of a Well Separated Pair Decomposition

Notice how the `areWellSeparated(X, Y)` function can be used to find out whether two clusters X and Y are well-separated.

3.1.2 Well Separated Pair Decompositions of Grid Graphs

For a given grid graph $G = (V, E)$ a well separated pair decomposition can be calculated by first calculating a quadtree for the nodes V of grid graph and then using the approach presented in section 3.1.1. In order to use the well separated pair decomposition of the grid graph for answering queries regarding the optimal shortest path distance between two nodes s and t, a suitable condition which specifies if a cluster pair (A, B) is well separated is required. A condition which can be used for this purpose is the following:

$$\texttt{areWellSeparated(A, B)} = \begin{cases} 1, & \text{if} \quad \exists p_a \in A : \exists p_b \in B : \forall s \in A : \forall t \in B : p_a, p_b \ \in \ \pi_{opt}(s,t) \\ 0, & \text{else} \end{cases}$$

where:

$\pi_{opt}(s,t)$ is the optimal path from s to t

This means there exists a node $p_a \in A$ and a node $p_b \in B$ which are both part of all shortest paths from A to B. Figure 3.1 shows an example of such a separation with p_a and p_b colored in red.

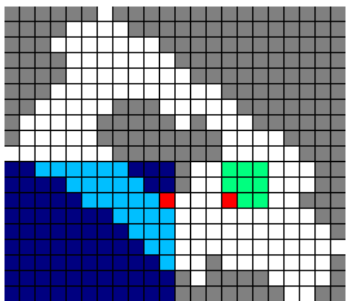

Figure 3.1: Example of a valid separation of a grid graph [Stu12]

Calculating Candidates for p_a and p_b

p_a and p_b both need to be part of all shortest paths from A to B. Therefore the distance $d_{opt}(p_a, p_b)$ between (p_a, p_b) needs to be minimal, otherwise the shortest path $\pi_{opt}(p_a, p_b)$ would be a sub-path of another shorter path, which is impossible.
Therefore candidates for p_a and p_b for given node sets A and B can be calculated with:

$$(p_a, p_b) = \underset{(x,y)\in A\times B}{\operatorname{argmin}} \; d_{opt}(x, y)$$

Algorithm 3.2 shows how to check, if two node sets A and B are well-separated. With this `areWellSeparated` function and the algorithm 3.1 a well-separated pair decomposition for grid graphs can be computed. Since the grid graph is assumed to be undirected, the decomposition satisfies the condition that $(u, v) \in V \times V$ with $u \neq v$, there exists a pair $(A, B) \in \mathbb{S}$ with $u \in A \wedge v \in B$ or $v \in A \wedge u \in B$.

```
Algorithm 4: areWellSeparated
Data: QuadTree Nodes $A, B$
Result: Boolean
$(p_a, p_b) = \operatorname{argmin}_{(x,y)\in A\times B} d_{opt}(x, y)$
for $x \in A$ do
    for $y \in B$ do
        if $d_{opt}(x, y) \neq d_{opt}(x, p_a) + d_{opt}(p_a, p_b) + d_{opt}(y, p_b)$ then
            return FALSE
return TRUE
```

Algorithm 3.2: Well Separation Check for Grid Graphs

Answering Shortest Path Distance Queries

While there exists a technique to retrieve cluster pairs for query points in $O(1)$ [FMS03], it is not applicable to the cluster pairs built for the grid graph. This is the case, because the clusters built for the grid graph do not use the original *areWellSeparated* of well separated pair decompositions with euclidean distance, but a modified version which was designed to work for grid graphs. Because of this, the required cell wide w cannot be computed from the euclidean distance between two points, which is required to apply the constant time query scheme proposed in [FMS03].

In order to use a decomposition of a grid graph as presented above to answer shortest path distance queries, it needs to be stored in a specific way. Given a decomposition of a grid graph $\mathbb{S} = \{(A_i, B_i) \mid i \in \mathbb{N}\}$ every node $n \in V$ needs to store the node sets $\mathbb{X}_n$ with

$$\mathbb{X}_n = \{X \subseteq V | (X, Y) \in \mathbb{S} \wedge n \in Y \vee (Y, X) \in \mathbb{S} \wedge n \in Y\}$$

This means that every node needs to store the node sets, which are well separated with the node sets it is a member of. Any node $n \in V$ also needs to store the p_a, p_b for all (A, B) where $n \in A$ or $n \in B$ and the distance $d_{opt}(d_a, d_b)$.
With this information stored per node algorithm 3.3 can be used to answer shortest path distance queries for nodes $s, t \in V$.

```
Algorithm 5: WSPDDistanceQuery
Data: Nodes $s, t \in V$
Result: $d_{opt}(u, v)$
$B = X \in \mathbb{X}_s \quad with \quad t \in X$
$(p_a, p_b) = lookupPs(s, t, B)$
if $s = p_a \wedge t = p_b$ then
    return $d_{opt}(p_a, p_b)$
if $s = p_a$ then
    return $d_{opt}(p_a, p_b) + WSPDDistanceQuery(p_b, v)$
if $t = p_b$ then
    return $WSPDDistanceQuery(s, p_a) + d_{opt}(p_a, p_b)$
return $WSPDDistanceQuery(s, p_a) + d_{opt}(p_a, p_b) + WSPDDistanceQuery(p_b, v)$
```

Algorithm 3.3: Distance Query for Decomposed Grid Graph

3.1.3 Optimization

Avoiding Neighbors

Since the distance from a node to its neighbor is easily calculatable in a grid graph, all separations (A, B) with $|X| = |Y| = 1$ and the nodes of A are neighbors of the nodes of B, can be omitted. This changes the query routine only slightly, such that for two given nodes u, v first it needs to be checked if u and v are neighbors. If they are, $d_{opt}(u, v)$ can be immediately reported as 1.

Separation Weight Optimization

Algorithm 3.1, which is used to calculate the well separated pair decomposition for a given quadtree, currently only checks if two quadtree nodes are well-separated if they have a level difference of at most one in the quadtree. This can be used to further optimize a well separated pair decomposition, which was calculated by Algorithm 3.1. Given two quadtree nodes A, B which are well-separated according to Algorithm 3.2, it is possible that there exists a parent node of A in the quadtree, which is well-separated with B. The same is true for the parent nodes of B and the quadtree node A. This is the case because increasing the size of a node set A or B also adds new candidates for nodes p_a and p_b and because Algorithm 3.1 does not check all combinations quadtree nodes.

```
Algorithm 6: WSPDOptimization
Data: Well Separated Pairs $(A, B) \in \mathbb{S}$
Result: Optimized Well Separated Pairs $\mathbb{S}'$
$\mathbb{S}' = \emptyset$;
for $(A, B) \in \mathbb{S}$ do
    if $\exists (C, D) \in \mathbb{S}' : A \times B \subseteq C \times D$ then
        continue;
    $best\_weight = |A| \cdot |B|$;
    $(A', B') = (A, B)$;
    for $P \in parents(A)$ do
        if $areWellSeparated(X, Y) \wedge |P| \cdot |B| > best\_weight$ then
            $best\_weight = |P| \cdot |B|$;
            $(A', B') = (P, B)$;
    for $P \in parents(B)$ do
        if $areWellSeparated(X, Y) \wedge |A| \cdot |P| > best\_weight$ then
            $best\_weight = |A| \cdot |P|$;
            $(A', B') = (A, P)$;
    $\mathbb{S}' = \mathbb{S}' \cup \{(A', B')\}$;
return $\mathbb{S}'$
```

Algorithm 3.4: Optimize WSPD of a Grid Graph

These properties can be used to apply Algorithm 3.4 to a well separated pair decomposition of a grid graph, to search for node separations (A, B), optimizing $|A| \cdot |B|$, while node separations (C, D) are omitted if another node separation can be used to answer all node pairs $(u, v) \in C \times D$. The set of node separations $\mathbb{S}'$ is therefore greater or equal in size in comparison to the original set $\mathbb{S}$. This means every node u needs to store less node separations, which results in faster query times and less memory overhead compared to the unoptimized node separations.

Trivial Separation

The `areWellSeparated` function can be extended to also check for other types of separation. Another separation condition which can be used in addition is the following:

$$\texttt{areWellSeparated(A, B)} = \begin{cases} 1, & \text{if} \quad \forall s \in A : \forall t \in B : d_{opt}(s, t) = |s_1 - t_1| + |s_2 - s_2| \\ 0, & \text{else} \end{cases}$$

Figure 3.2 shows such a separation for a grid graph. Given two nodes $s \in A$ and $t \in B$ in such a separation, $d_{opt}(s, t)$ can be easily calculated by $|s_1 - t_1| + |s_2 - s_2|$. Because of this property, such a separation is called a *trivial* separation.

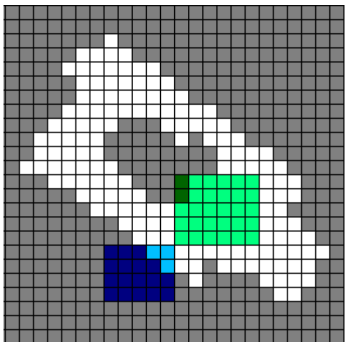

Figure 3.2: Example of a trivial separation of a grid graph [Stu12]

3.2 Generalized Cluster Pairs

The method presented in Section 3.1.2 only works for grid graphs, since a quadtree built on top of the graph is needed. Therefore, to also build node separations for a directed graph $G = (V, E)$, the node sets A, B of a separation (A, B) cannot be based on quadtree nodes.
Instead, node sets are grown from two initial, distinct nodes $s, t \in V$. As a condition, to decide whether two node sets A, B can be used as a separation (A, B) the following term is used:

$$\texttt{areSeparation(A, B)} = \begin{cases} 1, & \text{if} \quad \exists p_{AB} \in V : \forall s \in A : \forall t \in B : p_{AB} \in \pi_{opt}(s,t) \\ 0, & \text{else} \end{cases} \tag{3.1}$$

The node p_{AB} is referred to as the *portal node* of separation (A, B). Since these new separations (A, B) are not based on well separated pair decompositions, they are referred to as node selections instead of node separations.

To grow such node sets A, B, initial nodes $s, t \in V$ are selected randomly, such that $s \neq t$. From an optimal shortest path $\pi_{opt}(s, t)$ a node $p_{AB} \in \pi_{opt}(s, t)$ is selected.
$(\{s\}, \{t\})$ with node p_{AB} is now a valid selection for the definition above. For a given valid selection (A, B) Algorithm 3.5 and 3.6 can be used to enlarge the selection while remaining valid.

Algorithm 7: canBeAddedToA

```
Data: Valid Selection $(A, B), p_{AB}, v \in V$
Result: $Boolean$
for $b \in B$ do
    if $d_{opt}(v, p_{AB}) + d_{opt}(p_{AB}, b) \neq d_{opt}(v, b)$ then
        return FALSE
return TRUE
```

Algorithm 3.5: Check if $(\{v\} \cup A, B)$ is a valid selection for a valid selection (A, B)

```
Algorithm 8: canBeAddedToB
Data: Valid Selection $(A, B), p_{AB}, v \in V$
Result: $Boolean$
for $a \in A$ do
    if $d_{opt}(a, p_{AB}) + d_{opt}(p_{AB}, v) \neq d_{opt}(a, v)$ then
        return FALSE
return TRUE
```

Algorithm 3.6: Check if $(A, \{v\} \cup B)$ is a valid selection for a valid selection (A, B)

In order to find a set $\mathbb{S} = \{(A_i, B_i) | i \in \mathbb{N}\}$ such that for every pair s, t a selection $(A, B) \in \mathbb{S}$ with $s \in A$ and $t \in B$ can be found, Algorithm 3.7 can be used.

```
Algorithm 9: calculate-$\mathbb{S}$-1
Data: Graph $G(V, E)$
Result: $\mathbb{S}$
$\mathbb{S} = \emptyset$;
$\mathbb{T} = V \times V$;
while $\mathbb{T} \neq \emptyset$ do
    $(s, t)$ = randomly selected from $\mathbb{T}$;
    $p_{AB}$ = node from $\pi_{opt}(s, t)$;
    $A = \{s\}$;
    $B = \{t\}$;
    for $v \in V$ do
        if canBeAddedToA($A, B, p_{AB}, v$) then
            $A = A \cup \{v\}$;
        if canBeAddedToB($A, B, p_{AB}, v$) then
            $B = B \cup \{v\}$;
    $\mathbb{T} = \mathbb{T} \setminus A \times B$;
    $\mathbb{S} = \mathbb{S} \cup \{(A, B)\}$;
return $\mathbb{S}$
```

Algorithm 3.7: Calculate $\mathbb{S}$ for a given Graph $G = (V, E)$

A problem which occurs for Algorithm 3.7, is that for a valid selection (A, B), a new node $v \in V$, will be added to either A or B, even if the addition does not further reduce the size of $\mathbb{T}$ and therefore is not needed to improve the capacity of $\mathbb{S}$ to answer more s, t shortest path distance queries. While this would not harm the ability to answer queries, it does drastically increase the memory amount needed. Selection sets $\mathbb{S}$ calculated by Algorithm 3.8, which mitigates this issue, were up to 99.999582% smaller than selection sets calculated by Algorithm 3.7.

```
Algorithm 10: calculate-$\mathbb{S}$-2
Data: Graph $G(V, E)$
Result: $\mathbb{S}$
$\mathbb{S} = \emptyset$;
$\mathbb{T} = V \times V$;
while $\mathbb{T} \neq \emptyset$ do
    $(s, t)$ = randomly selected from $\mathbb{T}$;
    $p_{AB}$ = node from $\pi_{opt}(s, t)$;
    $A = \{s\}$;
    $B = \{t\}$;
    for $v \in V$ do
        if canBeAddedToA$(A, B, p_{AB}, v)$ $\wedge \mathbb{T} \cap \{v\} \times B \neq \emptyset$ then
            $A = A \cup \{v\}$;
        if canBeAddedToB$(A, B, p_{AB}, v)$ $\wedge \mathbb{T} \cap A \times \{v\} \neq \emptyset$ then
            $B = B \cup \{v\}$;
    $\mathbb{T} = \mathbb{T} \setminus A \times B$;
    $\mathbb{S} = \mathbb{S} \cup \{(A, B)\}$;
return $\mathbb{S}$
```

Algorithm 3.8: Calculate smaller $\mathbb{S}$ for a given Graph $G = (V, E)$

Figure 3.3 shows an example of a valid selection which was grown using algorithm 3.8 on a grid graph, whereas Figure 3.4 shows an example of a valid selection of a directed graph model of the road network of Andorra. The nodes of the different sets A and B are colored in the different colors green and blue, whereas the *portal node* is colored in red.

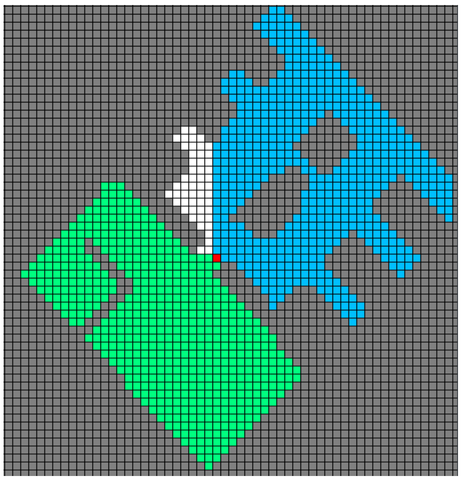

Figure 3.3: Example of a valid selection of a grid graph [Stu12]

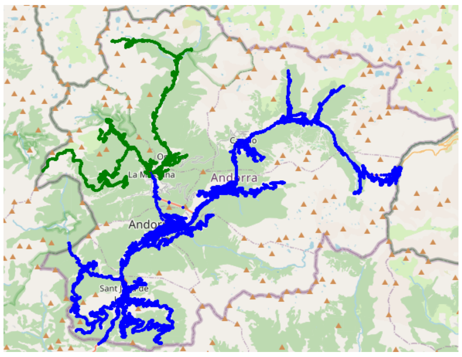

Figure 3.4: Example of a valid selection of a road network of Andorra

Answering Shortest Path Distance Queries

The previous section explains how to calculate a selection set for a given graph. How this selection set $\mathbb{S}$ can be used to answer shortest path distance queries for node pairs (s, t) will be explained in this section.

At first, all selections $(A, B) \in \mathbb{S}$ need to be sorted in descending order according to $|A| \cdot |B|$. This step could be omitted, but ensures that shortest path distances can be found as fast as possible. Every selection in this sorted list then can be uniquely identified by the index i at which it is stored in the sorted list. Now every node $n \in V$ needs to store two lists of tuples S and T.

$$S_n = \{(i, d(v, p_{AB_i}))|(A, B) \text{ at index } i \wedge v \in A\}$$
$$T_n = \{(i, d(p_{AB_i}), v)|(A, B) \text{ at index } i \wedge v \in B\}$$

where:

p_{AB_i} is the *portal node* of selection (A, B) at index i

Those lists S and T contain tuples with

- an index i, which identifies a selection
- the optimal distance between the node and the *portal node* of selection at index i

To ensure fast shortest path distance queries, those lists of tuples need to be sorted by the index of the selection they represent. Furthermore, neither selection set $\mathbb{S}$ nor the graph $G = (V, E)$ are required anymore to answer shortest path distance queries. To find the shortest path distance for given nodes s and t, the S_s list of s and the T_t list of t need to be searched for a common selection index. Once a common selection index i is found, the optimal path distance from s to t can be reported as the distance $d(s, p_{AB_i}) + d(p_{AB_i}, t)$, which is the sum of the second elements of the found tuple entries.

```
Algorithm 11: distanceQuery
Data: Source $s$, Target $t$
Result: $d_{opt}(s,t)$ or UNREACHABLE
$idx_s = 0$;
$idx_t = 0$;
while $idx_s < size(S_s) \wedge idx_t < size(T_t)$ do
    $i_s = S_s[idx_s].index$;
    $i_t = T_t[idx_t].index$;
    if $i_s = i_t$ then
        $d_s = S_s[idx_s].distance$;
        $d_t = T_t[idx_t].distance$;
        return $d_s + d_t$
    else if $i_s < i_t$ then
        $idx_s = idx_s + 1$;
    else
        $idx_t = idx_t + 1$;
return UNREACHABLE
```

Algorithm 3.9: Distance Query using Lists S_s and T_t

In contrast to the query scheme used by Hub Labels [DGSW14], this one does not need to find all the common indices of selections, but is able to report the shortest path distance when one common selection has been found. Since this query scheme does not contain any recursion and only performs a linear scan over the sorted lists, it performs much faster than the query scheme proposed in Section 3.1.2.

Another interesting property of this query scheme occurs when list S or T or even both of them are incomplete. Since the algorithm can report every common selection index in S and T, a missing index does not result in a wrong distance, but in UNREACHABLE.
Combined with another shortest path distance backup routine, this makes it possible to arbitrarily limit the sizes of the lists S and T while still reporting correct distances for all queries. To do so, first algorithm 3.9 is used to calculate the shortest path distance. If this query results in UNREACHABLE, the backup routine will be used instead.

3.2.1 Optimization

This section explains different approaches to minimize the total number of node selections, which results in less node selections stored per node and therefore faster query times, as well as a smaller memory overhead.

Optimizing T_n and S_n stored per Node

Currently the node selections stored by one node v can overlap which means there could be a subset of the selections stored by v which would be sufficient to answer all queries. This means that the lists S and T of every node can be optimized by using an algorithm which optimizes the set cover problem. For this purpose, a simple greedy approach can be used, which in every step adds the selection $(A, B) \in T_n$, which adds most nodes which cannot be reached from node n by the currently selected selections $T_{curr} \subset T_n$. In the same way S_n is optimized, by the addition of the node selection $(A, B) \in S_n$, which adds the most nodes which currently cannot reach n as a target when the currently selected selections $S_{curr} \subset S_n$ are used.

Such an optimization of lists S and T has the following problem:
If a node selection (A, B) can be omitted because other selections $(C_1, D_1) \ldots (C_i, D_i)$ can be used to answer all queries $(u, v) \in A \times B$, all the nodes v are not allowed to omit one of the node selections $(C_1, D_1) \ldots (C_i, D_i)$, otherwise there could exist pairs $(s, t) \in V \times V$ where s and t do not store any node selection (X, Y) which could answer the distance query (s, t).
To avoid this problem two sets S_{keep} and T_{keep} are created, where all the selections which cannot be omitted are kept. Selections which are in S_{keep} are not allowed to be omitted for any optimization of any list S. The same needs to be ensured for T_{keep} and for any T. After every optimization of any lists S and T, with the results S' and T', S_{keep} and T_{keep} need to be updated to $S_{keep} = S_{keep} \cup T'$ and $T_{keep} = T_{keep} \cup S'$.

Limiting T_n and S_n stored per Node

As mentioned in Section 3.2, the proposed query routine is able to report correct distances, even if the lists S and T are incomplete. This can be utilized to limit the size of the two lists. When lists S and T are optimized as described in Section 3.2.1, a size limit can be ensured for the lists by stopping the greedy algorithm when the number of added selections exceeds the given size limit. This can be used to calculate node selections, while it can be ensured that the result does not exceed a given memory constraint.

3.2.2 Shortest Path Extraction

While node selections can be used to calculate the shortest path distances between two nodes, they currently cannot answer shortest path queries. In this section two different approaches are presented, showing how node selections can be used to answer shortest path queries.

Explicit Node Selection Tree Views

A node selection (A, B) containing all nodes of shortest paths from node from A to B can be seen as two trees A and B with the center of the selection as root. Such a selection (A, B) has the following property:

$$\forall s \in A : \forall t \in B : \forall x \in \pi_{opt}(s, t) : x \in A \lor x \in B \tag{3.2}$$

where:

$\pi_{opt}(s, t)$ is an optimal path from s to t

Figure 3.5 shows such a selection (A, B). The nodes in blue are the set A, the nodes in green are the set B. The node in red is the portal node of the node selection. In this representation all edges of the nodes which are not part of a shortest path from a node $s \in A$ to a node $t \in B$ are omitted. Such a structure of a node selection will be called a *tree view* of the selection in the following.

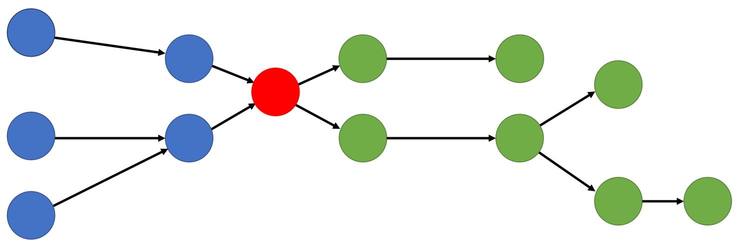

Figure 3.5: Example of a complete tree view

For such a selection shortest paths queries can be answered when a tree view of them is explicitly stored in memory. In order to do this, nodes need to store a pointer to itself in the tree view of the selection in addition to the selection index and the distance to p_{AB}. To answer shortest path queries these pointers can be used to locate s and t in the tree view of the selection which answers the shortest path distance query. Once the position of s and t in such a tree view has been found, the shortest path can be constructed by chasing the outgoing edges starting from node s in the tree view until a node with multiple outgoing edges is found. The same needs to be done for node t in the tree view, but with ingoing edges. The shortest path between nodes s and t can be constructed by concatenation of the two node sequences found by the forward search from s and the backward search from t in the tree view.

This cannot be done for all node selections, because they do not satisfy Equation 3.2. Since most of the tree views of node selection do not satisfy this property their tree views have holes and therefore cannot be used directly for constructing shortest paths. Figure 3.6 shows a tree view of such a node selection. The nodes colored in gray are not an element of $A \cup B$.

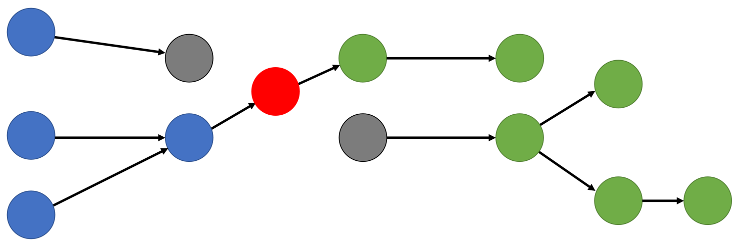
Figure 3.6: Example of tree view with a hole

In such a case the tree view needs to also include nodes which are not elements in A or B of the node selection (A, B). The tree view only needs to store the first nodes which are not in the node selection, but are on a shortest path from any node $s \in A$ to p_{AB} or on the shortest path from p_{AB} to any node $t \in B$. The construction of a path from s to t is the same as for node selections satisfying Equation 3.2, but once two nodes $x \notin A$ and $y \notin B$ are being found while the searches, have been started from s and t in the tree view, a new shortest path query for nodes x and y has to be performed, which can be answered recursively by either another node selection or any backup routine which is able to answer shortest path queries.

Implicit Node Selection Tree Views

While the previous proposed solution is able to answer shortest path queries, the explicitly stored tree views of the node selections can be omitted. Every node does not need to store a pointer into the tree view of a node selection, but to the node which would be its successor in the tree view. This encodes the whole tree view implicitly into the storage of the lists S and T of the nodes of a graph. If a tree view had a hole, as described in Figure 3.6 the stored pointer would not be able to point to a next element, but needs to encode the node which would come next in the shortest paths. As described before, this storage structure can answer shortest path queries recursively by the usage of the implicitly stored node selection tree views, while it is only required to store one pointer per node per selection more, in comparison to only shortest path distance calculations.

The approach to answer shortest path distance queries presented in this section can be used for arbitrary graphs. This makes it possible to also use it for grid graphs. It turns out that in comparison to the well separation pair decomposition based approach, presented in Section 3.1, less information needs to be stored per node, as well as the average runtime per query is much better.
Another advantage over the grid graph specific approach is the ability to prune the stored information per node while remaining the ability to answer shortest path distance queries correct. This is not possible with the query scheme proposed in Section 3.1.2, because of the recursive structure

of Algorithm 3.3. In order to correctly answer a shortest path distance query for nodes s and t, all recursive calls of Algorithm 3.3 also need to be answered correctly, which is unlikely if the information stored per node is pruned. This makes the later proposed approach superior to the grid graph specific approach, which is why in the following only the approach presented in this section is considered.

4 Benchmarks

In this chapter, different versions of the algorithm presented in Section 3.2 are compared to each other and to Hub Labels [DGSW14]. For the benchmarks three different graphs where used from the Open Street Map graph data[1]. Table 4.1 shows the different graphs used to compare the different approaches. It was not possible to preprocess bigger graphs, because algorithm 3.8 needs to store all pairs $(s, t) \in V \times V$, which caused a to high memory consumption for bigger graphs. Algorithm 3.8 needs to compute the distance between two nodes while growing selection sets. For this, a lookup table of the distances was used, but any distance oracle could have been used there. The used hardware was a AMD Ryzen Threadripper 1950X 16-Core Processor with 256 GB RAM.

Graph	**|V|**	**|E|**
Andorra	26516	49968
Malta	77950	153505
Bremen	130276	248309

Table 4.1: Graphs used for the benchmarks

For the benchmarks the distances for all pairs $(a, b) \in V \times V$ were computed and then grouped by their dijkstra rank.

Dijkstra Rank

The dijkstra rank of a node pair $(s, t) \in V \times V$ is defined as the number of nodes pulled out of the heap by dijkstras algorithm presented in algorithm 2.1 when started with initial node s as source and t as target.

4.1 Preprocessing

The preprocessing times, as well as memory consumption are quite high. Table 4.2 shows the different preprocessing times as well as the time used for the optimization presented in Section 3.2.1 and the RAM usage during the preprocessing phase.

[1]https://download.geofabrik.de

Graph	Preprocessing Time	Optimization Time	Preprocessing Memory Consumption
Andorra	1.16 min	7.23 min	9 GBs
Malta	31.7 min	52.1 min	42 GBs
Bremen	74.6 min	144.1 min	173 GBs

Table 4.2: Preprocessing Times and Memory Consumption

4.2 Memory

Table 4.3 compares the number of stored items per node between unrestricted selection sets and Hub Labels. In both cases these are two numbers per item. For the Hub Labels these are the ids of the nodes and the distances to the nodes, For the selection sets these are the distances to centers of the node selections, as well as the id which identifies the node selection. It turns out that Hub Labels need to store fewer items per node, which results in a smaller memory footprint. To avoid high memory requirements for node selections, the number of items stored per node can be restricted, as proposed in Section 3.2.1.

Graph	Hub Labels	unrestrained Node Selection
Andorra	59.92	61.81
Malta	92.70	207.63
Bremen	80.90	196.66

Table 4.3: Average number of items stored per node

4.3 Query

In this section the query routine proposed in Section 3.2 is compared to Hub Labels. Furthermore different size limits for the information stored per node, as proposed in Section 3.2.1, are compared to each other. The benchmarks were created by grouping all queries $(s, t) \in V \times V$ by their dijkstra rank and by calculating the shortest path distances of all those queries together. Then the runtimes of all those queries were divided by the number of queries which were run together. Because there are not many queries with the highest dijkstra rank, the runtime of the queries with the highest dijkstrank of a graph were not stable accross different evaluations.

4.3.1 Runtime

Figure 4.1 shows the different algorithms in comparison with the dijkstra algorithm used for Contraction Hierarchies [GSSD08] As it can be seen, this algorithm is overall much slower when compared to Hub Labels and the node selection based algorithms. Because of that, the following figures do not include this algorithm, but only Hub Labels and the different versions of the node selection based algorithms.

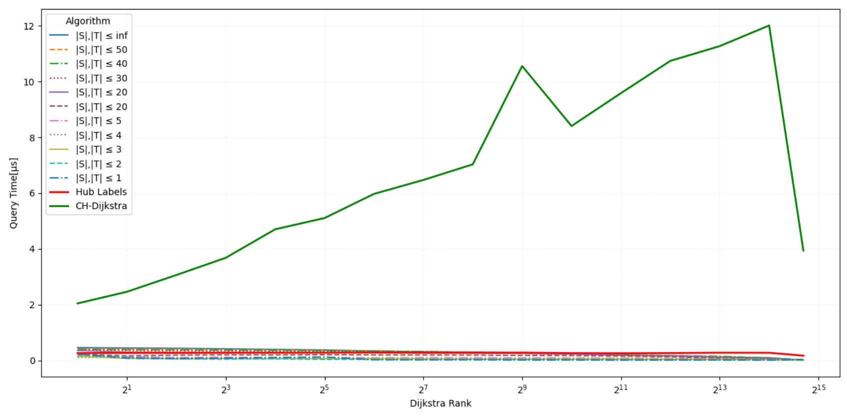

Figure 4.1: Runtime in microseconds per query for the Andorra graph including CH-Dijkstra

In the following the different runtimes of Hub Labels and different node selection queries, as described in Algorithm 3.9, are compared. The difference between the node selection based approaches are how big the lists S and T were allowed as described in Section 3.2.1. Since limiting the sizes of lists S and T can lead to the query algorithm reporting `UNREACHABLE`, even if there is a shortest path, in which case a real world application would need to call a backup routine to calculate the optimal shortest path distance, the queries which were benchmarked were split into two different sets. The first set contains only queries for which the benchmarked algorithm was able to find an optimal shortest path distance, the second one only contained the queries where `UNREACHABLE` was reported.

Runtime of queries answered with an optimal distance

Figures 4.2, 4.3 and 4.4 show the average time of a successful query in microseconds per dijkstra rank. When lists S and T are unlimited, queries with a lower dijkstra rank take longer than the ones with a high rank. This is the case, because queries with a high dijkstra rank can be answered more often by node selections with big node sets A and B, which, according to Section 3.2, have smaller indices and therefore can be found faster when scanning through sorted lists S and T. The same can be observed even for size limited lists S and T, but not as noticeably as without limits. In comparison to Hub Labels, Figures 4.2, 4.3 and 4.4 show that for high dijkstra ranks, queries can be answered faster with node selection based queries. Even for small dijkstra ranks the queries are faster compared to Hub Labels when the sizes of lists S and T were limited by an appropriate size. As it can be seen in Figures 4.3 and 4.4, even a size limit of 50 elements per list S and T is enough to achieve a performance comparable to Hub Labels for all queries independent of their dijkstra rank. Only for the Andorra graph, have Hub Labels been faster for queries with a small dijkstra rank even for size limits as small as 20 elements per list.

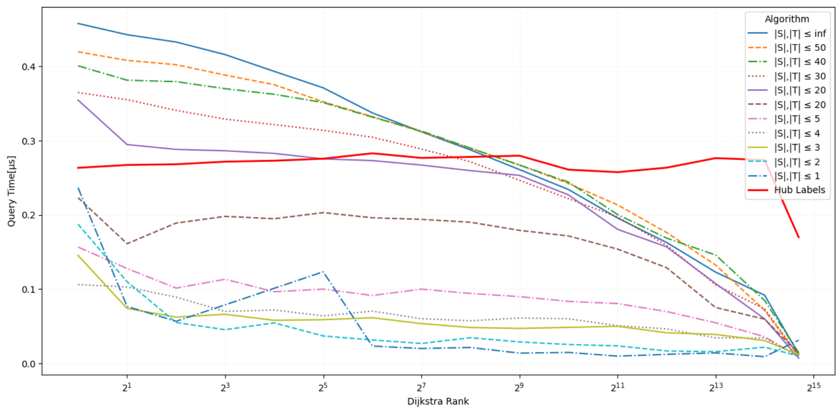

Figure 4.2: Runtime in microseconds per query for the Andorra graph

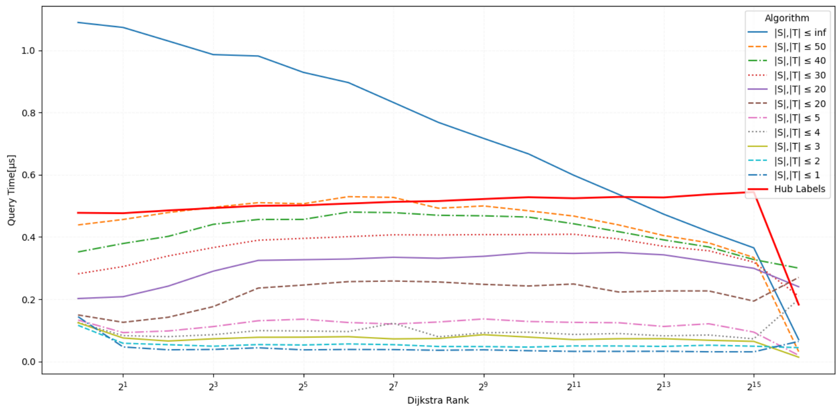

Figure 4.3: Runtime in microseconds per query for the Malta graph

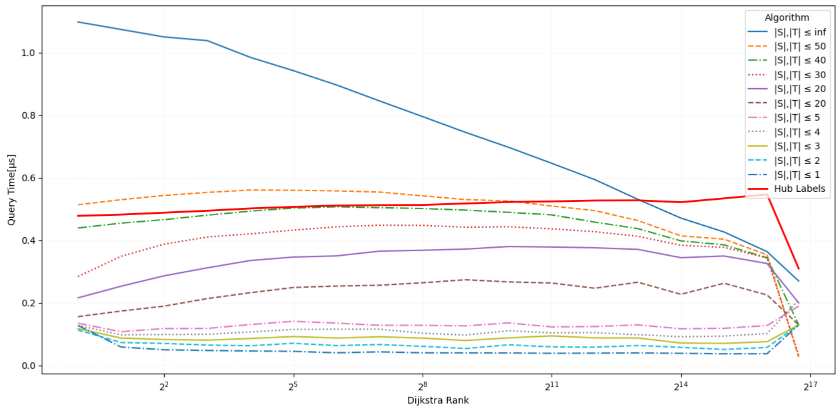

Figure 4.4: Runtime in microseconds per query for the Bremen graph

Runtime of queries answered with `UNREACHABLE`

Figures 4.5, 4.6 and 4.7 show the average time of a query returning `UNREACHABLE` in microseconds per dijkstra rank. As expected, the average runtime of queries which are not successful is independent of their dijkstra rank. This is due to the fact that lists S and T need to be scanned through and the query routine cannot return early because $S \cap T = \emptyset$ and therefore no common index can be found. The smaller the limiting factor for S and T is chosen, the faster an unsuccessful query can be performed because of the smaller sizes of S and T.

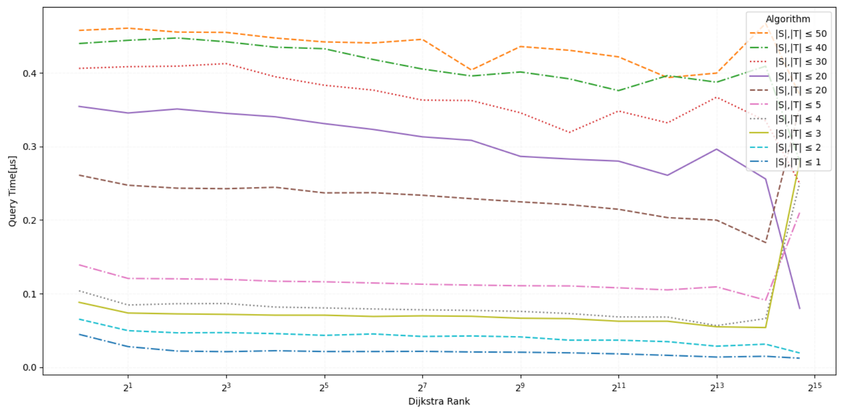

Figure 4.5: Runtime in microseconds per query for the Andorra graph

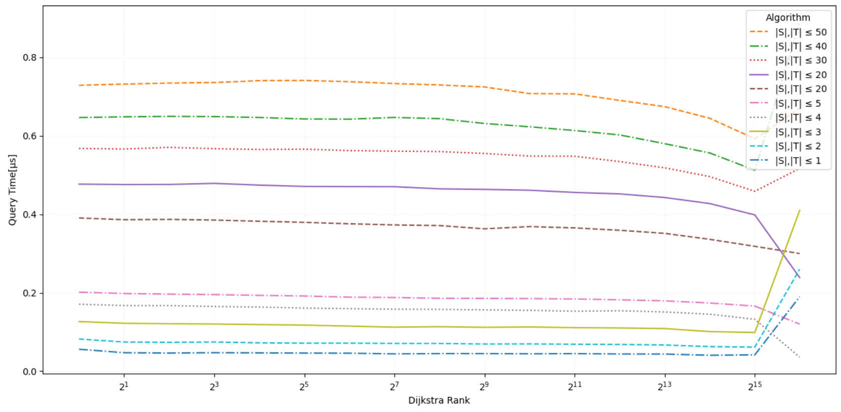

Figure 4.6: Runtime in microseconds per query for the Malta graph

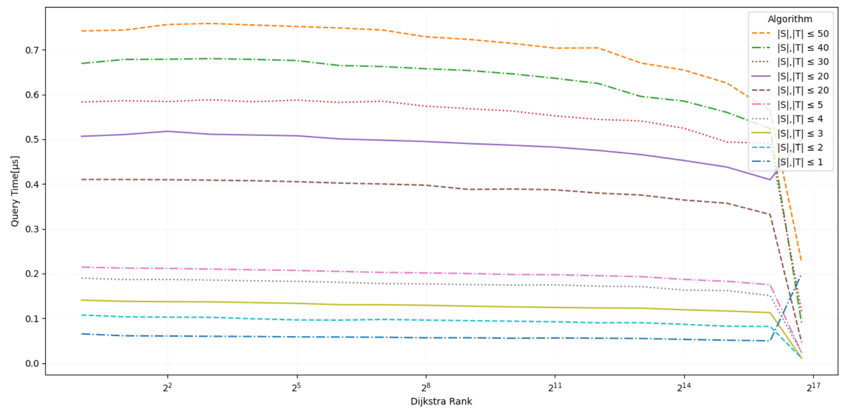

Figure 4.7: Runtime in microseconds per query for the Bremen graph

4.3.2 Success Rate

Figures 4.8, 4.9 and 4.10 show the success rate of queries for different size restrictions, as proposed in Section 3.2.1, depending on the dijkstra rank of the queries. The success rate p at dijkstra rank r, means that a random query with a dijkstra rank of r can be answered with a probability of p. It can be observed the higher the dijkstra rank of a query, the higher its probability to be able to answer the query. Almost all the restriction sizes are able to answer queries with high dijkstra ranks reliably, which makes the node selection based approach to shortest path distance computation useable even with strict restrictions on the number of items stored per node.

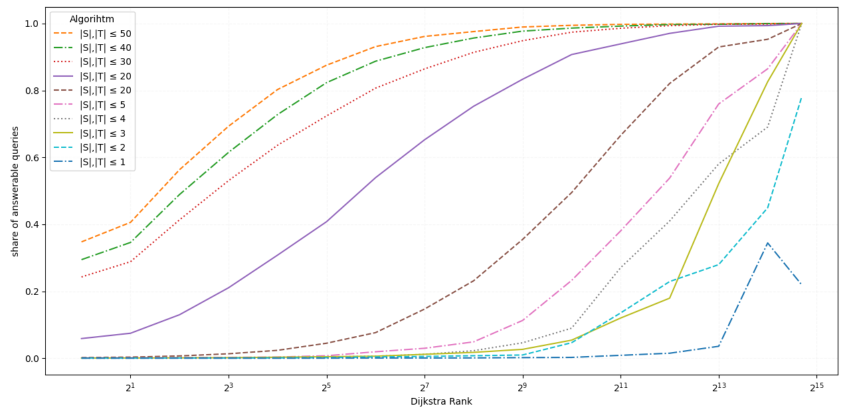

Figure 4.8: Success rate of different restricted node selection based queries for the Andorra graph

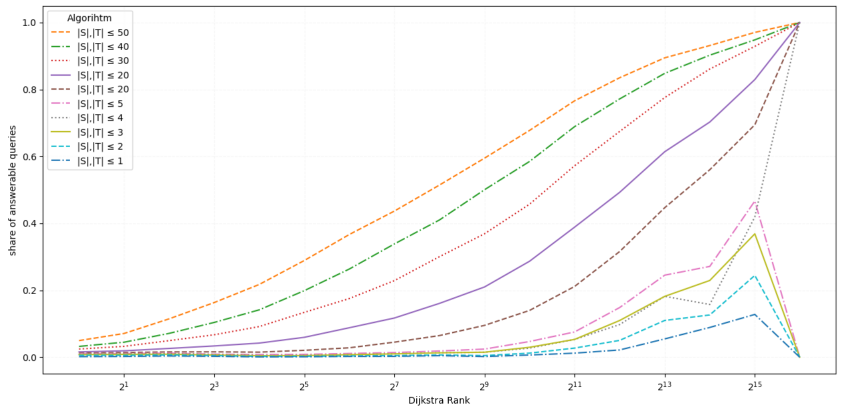

Figure 4.9: Success rate of different restricted node selection based queries for the Malta graph

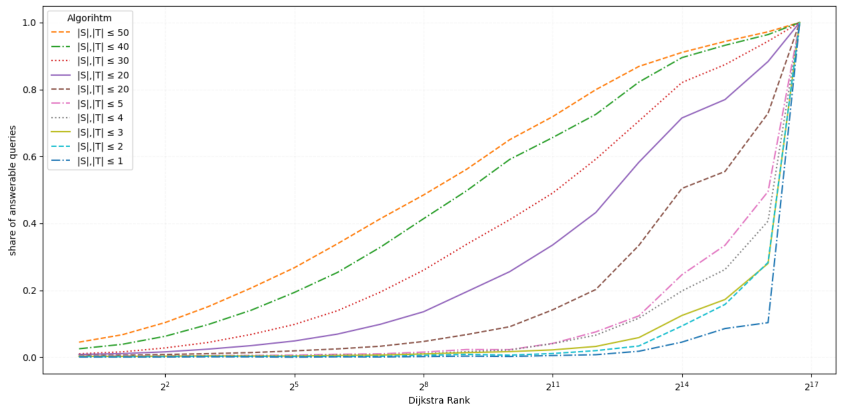

Figure 4.10: Success rate of different restricted node selection based queries for the Bremen graph

Table 4.4 shows the success rate of a random query for the three graphs for different size limits of the lists S and T. It can be observed that even for small size limits ≤ 5, depending on the graph, up to 70.5% of all shortest path distance queries can be answered. Even when the lists S and T are limited to a size of two, which corresponds to saving four integers per node, between 18.9% and 48.5% of all queries can be answered successfully.

$\lvert S\rvert, \lvert T\rvert$ / Graph	≤ 50	≤ 40	≤ 30	≤ 20	≤ 10	≤ 5	≤ 4	≤ 3	≤ 2	≤ 1
Andorra	0.997	0.996	0.990	0.976	0.884	0.705	0.650	0.616	0.485	0.209
Malta	0.934	0.913	0.867	0.806	0.614	0.423	0.386	0.308	0.222	0.086
Bremen	0.934	0.912	0.882	0.788	0.634	0.463	0.410	0.351	0.189	0.088

Table 4.4: Success rates for whole graphs

5 Conclusion and Outlook

In this thesis a new lookup based shortest path distance calculation scheme has been developed. The proposed query algorithm is able to find out if the shortest path distance for a given node pair (s, t) can be calculated, which makes it possible to prune the graph data calculated by the proposed preprocessing scheme in an arbitrary way, because once it is obvious that the query algorithm produces an obviously invalid result (`UNREACHABLE`), a backup routine applied to calculate the shortest path distance. Additionally, when compared to state of the art techniques for shortest path distance calculations a noticeable speed up can be observed. This makes the proposed shortest path distance query scheme able to compete with other state of the art techniques.

Outlook

The preprocessing scheme proposed in this thesis takes a long time to complete. Additionally, the memory usage makes it unusable for big graphs. For the future, this needs to be addressed to make node selection based shortest path distance calculation applicable.

As a distance oracle for the preprocessing, a lookup table was used. Instead of a lookup table, another fast shortest path distance query scheme could also be applied, such as Hub Labels [DGSW14], Transit Nodes [BFSS07] or an interpolation between Contraction Hierarchies and Hub Labels [Fun20]. This would drastically decrease the memory usage of the preprocessing algorithm.

Another way to further decrease the memory usage of the preprocessing algorithm would be to get rid of the necessity to store all pairs $(s, t) \in V \times V$, which are currently needed to ensure that for any such pair (s, t) there exists a selection which is able to answer a shortest path distance query for nodes s and t. If this could be avoided, the memory consumption of the preprocessing algorithm could be small enough to even preprocess big graphs.

Currently, the nodes only store the information needed in two 64-bit integers. This could be further optimized by the usage of different encodings or other compressions, such as common prefix compression, the same way it can be done for Hub Labels [ADGW11].

Another possible optimization could be the merging of node selections. Maybe it will possible to merge two node selections (A, B) and (C, D) to a new node selection $(A \cup C, B \cup D)$ if a new portal node p_{new} is chosen for the merged selection. This could further minimize the number of node selections and therefore the amount of information stored per node, as well as the runtime of the query routine.

Furthermore, it needs to be researched if the information density stored per node can be further optimized by reduction to other optimization problems, such as the hitting set problem.

In this thesis, only grid graphs were considered, in which every node has at most four neighbors. It is common to also add diagonal cells in a grid as neighbors in a grid graph. This could lead to new challenges and optimization opportunities not covered by this thesis.

Bibliography

[ADGW11] I. Abraham, D. Delling, A. Goldberg, R. Werneck. “A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks”. In: Jan. 2011, pp. 230–241. ISBN: 978-3-642-20661-0. DOI: 10.1007/978-3-642-20662-7_20 (cit. on pp. 13, 43).

[BFSS07] H. Bast, S. Funke, P. Sanders, D. Schultes. “Fast Routing in Road Networks with Transit Nodes”. In: *Science (New York, N.Y.)* 316 (May 2007), p. 566. DOI: 10.1126/science.1137521 (cit. on pp. 13, 18, 43).

[CK95] P. B. Callahan, S. R. Kosaraju. “A Decomposition of Multidimensional Point Sets with Applications to <i>k</i>-Nearest-Neighbors and <i>n</i>-Body Potential Fields”. In: *J. ACM* 42.1 (Jan. 1995), pp. 67–90. ISSN: 0004-5411. DOI: 10.1145/200836.200853. URL: https://doi.org/10.1145/200836.200853 (cit. on p. 21).

[DGSW14] D. Delling, A. V. Goldberg, R. Savchenko, R. F. Werneck. “Hub Labels: Theory and Practice”. In: *Experimental Algorithms*. Ed. by J. Gudmundsson, J. Katajainen. Cham: Springer International Publishing, 2014, pp. 259–270. ISBN: 978-3-319-07959-2 (cit. on pp. 13, 18, 30, 35, 43).

[Dij59] E. W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on pp. 13, 16).

[FMS03] S. Funke, D. Matijevic, P. Sanders. “Approximating Energy Efficient Paths in Wireless Multi-hop Networks”. In: *Algorithms - ESA 2003*. Ed. by G. Di Battista, U. Zwick. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 230–241. ISBN: 978-3-540-39658-1 (cit. on p. 23).

[Fun20] S. Funke. “Seamless Interpolation Between Contraction Hierarchies and Hub Labels for Fast and Space-Efficient Shortest Path Queries in Road Networks”. In: *Computing and Combinatorics*. Ed. by D. Kim, R. N. Uma, Z. Cai, D. H. Lee. Cham: Springer International Publishing, 2020, pp. 123–135. ISBN: 978-3-030-58150-3 (cit. on p. 43).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *Proceedings of the 7th International Conference on Experimental Algorithms*. WEA’08. Provincetown, MA, USA: Springer-Verlag, 2008, pp. 319–333. ISBN: 3540685480 (cit. on pp. 17, 36).

[Stu12] N. Sturtevant. “Benchmarks for Grid-Based Pathfinding”. In: *Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 144–148. URL: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf (cit. on pp. 16, 22, 26, 28).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature