Fast Computation of Distance Tables using Highway Hierarchies

Sebastian Knopp Peter Sanders Dominik Schultes Frank Schulz
Dorothea Wagner

July 25, 2006

Abstract

This technical report considers the problem of computing distances of shortest paths between all pairs of nodes from given sets of sources and targets. The straightforward method to solve this is to run Dijkstra's Algorithm once for every source: during one run the distances from one source node to all target nodes are determined. If we deal with large road networks and numerous sources and targets this method affords a lot of time. Hence, we are interested in faster algorithms for this problem.

For point to point shortest path queries in large road networks the Highway Hierarchies algorithm on average is about 8 000 times faster than Dijkstra's Algorithm. In this paper, we describe how this concept of very efficient and accurate route planning can be used to compute distance matrices. A first implementation of the presented approach already yields a speedup of over 1 500, relative to the time needed by Dijkstra's Algorithm.

1 Introduction

Given a weighted graph that represents a street network, a problem instance consists of a number of sources and destinations located in the graph. For these sets of nodes we want to know the distances from all sources to all destinations. Hence the result of such an $M \times N$ query is a matrix of distances. An entry of this distance table denotes the distance from the source corresponding to the current column, to the target corresponding to the current row.

The traditional way to compute distances in graphs is Dijkstra's Algorithm. To create a distance matrix, this single source shortest paths algorithm can be run once for every source node. It expands the search circularly around one source node and can be stopped after all targets have been found.

An important application for this algorithmic problem appears in the field of logistics, where the first step of tour planning for vehicles is the computation of such a distance table. Note that in practice the overall running time for tour planning often is dominated by the computation of the distance matrix if the plain version of Dijkstra's Algorithm is used. So this is a situation, where it is very interesting to speed up the calculation of shortest path distance tables. Another application is the computation of very large distance matrices for fast distance approximations based on table lookups.

Various recent publications deal with accelerating the determination of a single route with one source and one destination in large sparse graphs that represent road networks. Most of them perform a preprocessing step and with that precomputed information they are able to answer shortest path queries very quickly. One of the fastest known techniques in terms of preprocessing and query time is the concept of Highway Hierarchies. For the road network of Europe the preprocessing takes only 15 minutes and single pair queries can be computed in 0.76 ms on average.

In this work we will show how the Highway Hierarchies approach can be applied to the computation of distance matrices. The main idea is to make use of the fact, that the Highway Hierarchies search spaces are very small, so a lot of them can be stored contemporary in main memory.

2 Preliminaries

In this section we give basic definitions of graphs, a formal problem description and outline Dijkstra's Algorithm.

A directed graph $G = (V, E)$ is a pair of nodes V and edges $E \subset V \times V$. We will denote the number of nodes $|V|$ by n and the number of edges $|E|$ by m. We call $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{(v, u) \mid (u, v) \in E\}$ the *converse graph* of G.

A path P in the graph G is a sequence of nodes $(v_0, v_1, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$. If $0 \leq k < l \leq n$ then $P|_{v_k \to v_l}$ denotes the subpath $(v_k, \ldots, v_l)$ of P. Edge weights are given by a function $w : E \to \mathbb{R}_{>0}$. The length $l(P)$ of a path is the sum of the weights of its edges $l(P) = \sum_{i=0}^{n} w(v_i, v_{i+1})$. We call P a *shortest path* from s to t, if there is no P' with $l(P') < l(P)$ and its length is denoted by $d(P)$. The distance $d(s, t)$ of two vertices is the length of a shortest path from s to t.

The task we address in this work is formally described in the following. Given a set of sources $S = \{s_i \mid 0 \leq i < M\} \subset V$ and a set of targets $T = \{t_i \mid 0 \leq j < N\} \subset V$ we want to compute a distance matrix $d_{i,j} = D \in \mathbb{R}^{M \times N}$ such that $d_{i,j} = d(s_i, t_j)$ is the length of a shortest path from s_i to t_j. We will refer to this as the *$M \times N$ shortest path problem*. We can assume w.l.o.g. that $N \leq M$, else we swap S and T and consider the given problem instance in the converse graph.

The classic algorithm for the Single Source Shortest Path problem is the *Algorithm of Dijkstra*, which finds shortest paths form a source s to all vertices in the graph. During the algorithm every node takes one of the three states `settled`, `reached` or `unreached`. Initially all nodes are `unreached`. Nodes to those any path - not necessarily shortest - has been found are `reached`. A node v is `settled`, if a shortest path from s to v has been found, and the distance is exact. We call a tentative distance from s to v *exact*, if it is equal to the length of a shortest path from s to v.

`Reached` nodes are managed in a priority queue, which supports the operations *insert*, *decreaseKey* and *extractMin*. There is a function RELAX, that checks for an edge (u, v) if it can improve the path to v. The method *insert* adds an element to the queue. This happens when an edge to an `unreached` node is relaxed. Then this node is inserted into the queue with its tentative distance as key. Calls to *decreaseKey* are made if RELAX can improve the distance to a `reached` node. This step updates the key to the new tentative distance. Calls to *extractMin* are performed to get the smallest element of the queue. Elements obtained by this operation are known to be exact and can be set to `settled`.

3 Highway Hierarchies

In the following, we repeat the concept of Highway Hierarchies method as it is presented in [1]. The query part of the Highway Hierarchies algorithm is the core of our fast distance matrix algorithm that we present in section 4. We give an overview of the terminology of Highway Hierarchies and explain the query algorithm. The preprocessing data we need for $M \times N$ queries is exactly the same as for point to point queries, so for information about the construction process we refer to [1]. This section begins with the definition of a highway network and its core and continues with an explanation of the Highway Hierarchies query algorithm.

3.1 Definitions

A *highway hierarchy* of a graph G consists of a given number of levels $G_0, G_1, \ldots, G_L$, where $G_l = (V_l, E_l)$ and is defined inductively:

- *Base Case*: $G'_0 := G_0 := G$, the original graph.
- *First Step*: Definition of a *highway network* G_{l+1} of G_l.
- *Second Step*: *Contraction*: Bypassable nodes define the core G'_l of level l.

First Step (highway network). The definition of a *highway network* is based on neighbourhood radii, that are chosen for every node u in a level l: $r_l^{\leftarrow}(u)$ for the forward graph and $r_l^{\rightarrow}(u)$ for the converse graph. Those nonnegative reals define the *neighbourhood* with respect to the forward graph $\mathcal{N}_l^{\rightarrow}(u) := \{v \in V'_l \mid d_l(u,v) \leq r_l^{\rightarrow}(u)\}$ and analogously the neighbourhood with respect to the backward graph $\mathcal{N}_l^{\leftarrow}(u) := \{v \in V'_l \mid d_l(v,u) \leq r_l^{\leftarrow}(u)\}$.

The *highway network* $G_{l+1} = (V_{l+1}, E_{l+1})$ of a graph G'_l is the subgraph of G'_l induced by the set of edges E_{l+1} that is defined as follows: an edge $(u,v) \in E'_l$ belongs to E_{l+1} if and only if there are nodes $s, t \in V'_l$ such that the edge (u,v) appears in the canonical shortest path $(s, \ldots, u, v, \ldots, t)$ from s to t in G'_l with the property that $v \notin \mathcal{N}_l^{\rightarrow}(s)$ and $u \notin \mathcal{N}_l^{\leftarrow}(t)$.

Second Step (core). For given *bypassable* nodes $B_l \subseteq V_l$ the set S_l of shortcut edges that bypass the nodes in B_l is defined: for each path $P = (u, b_1, b_2, \ldots, b_k, v)$ with $u, v \in V_l \backslash B_l$ and $b_i \in B_l, 1 \leq i \leq k$, the set S_l contains an edge (u,v) with $w(u,v) = l(P)$. Then the *core* $G'_l = (V'_l, E'_l)$ of level l is defined by its node and edge sets: $V'_l := V_l \backslash B_l$ and $E'_l := (E_l \cap (V'_l \times V'_l)) \cup S_l$.

3.2 Query

Now we can describe the Highway Hierarchies query algorithm. This is a modification of Dijkstra's Algorithm and can be performed in the converse graph as well as in the original graph. We describe the query without considering any abort criteria and we only pay attention to expanding the search from one source node. This corresponds to the exploration of the complete search space as described on page 47 in [3], where bounds for the worst case bounds are constructed. This query algorithm is used in section 4 to obtain the desired distance matrix.

In addition to the tentative *distance* δ from the source the key of a node includes the search *level* l and the *gap* to the next applicable neighbourhood border. In order to sort the nodes by a priority, we comprehend the keys as triples $(\delta, -l, gap)$ and use lexicographical ordering.

Starting at a node s a local search in level 0 is performed, the gap to the next border is set to the neighbourhood radius of s in level 0. When a node v is settled, the gap of v is decreased by the length of the edge from the parent u of v. So we can detect if an edge crosses a neighbourhood border by checking if its new gap would be negative. In this case we ascend to a higher level l and we call v an *entrance point* to this level. An edge (u,v) is skipped, if its level is less than the new search level l (Restriction 1), otherwise v adopts the new search level l and the gap to the border of the neighbourhood of u in level l.

If an entrance point v to a level l is a bypassed node, we are outside the core and the gap of v is set to infinity. When a node $u \in V'_l$ is settled, the core is entered and the gap to the border of the level l neighbourhood is assigned to u. Once the core of a certain level was entered, no edges leading to a bypassed node are relaxed (Restriction 2).

4 Computing Distance Tables

In this section, we describe how the Highway Hierarchies query algorithm can be applied to solve the problem of finding $M \times N$ shortest paths. Even in very large graphs, with millions of nodes, only very few nodes are visited by the Highway Hierarchies query algorithm. Because of this small search space sizes we are able to store distances to all visited nodes for a lot of search spaces. This is the basic idea for our $M \times N$ Highway Hierarchy algorithm: we remember search spaces and intersect them to compute the desired distance matrices quickly.

4.1 Basic Algorithm

We start this section with introducing the terms of forward and backward search spaces for Highway Hierarchies: the *Highway Hierarchy Forward Search Space* $\mathrm{FWS}(s) \subseteq V$ for a node $s \in V$ is the set of nodes that are settled during a query that is performed as described in section 3.2 originating from a source node s in the graph G. Respectively the *Highway Hierarchy Backward Search Space* $\mathrm{BWS}(t) \subseteq V$ for a node $t \in V$ is the set of nodes that are settled during a query originating from a target node t in the converse graph $\overline{G}$.

If we omit the *abort on success* criterion of the Highway Hierarchies query algorithm for point to point queries, we can say that the shortest path distance from a source node s to a target node t is determined by identifying a node $v \in \mathrm{FWS}(s) \cap \mathrm{BWS}(t)$ with $d(s,v) + d(v,t) = \min\{d(s,v) + d(v,t) \mid v \in FWS(s) \cap BWS(t)\}$. In this way one can obtain the distance from s to t for all the pairs $(s,t) \in S \times T$. We will use this formulation of the Highway Hierarchies algorithm for point to point queries to obtain a fast method to obtain entries of the desired distance matrix. The next paragraph describes, how the search spaces can be intersected and how their minimum can be determined.

First, we initialise the distance matrix entries to infinity. We assumed that $N \leq M$, thus for all $n \in \mathrm{BWS}(t)$ we store the distances $d(n,t)$ to the target $t \in T$. This can be done with a backward search for every target node that maintains a data structure $b(v)$ for every node v where distances to targets are remembered. When a node v becomes settled during a search originating in t, we store the distance $d(v,t)$ in $b(v)$. After we have remembered the backward search spaces in this way, we can start searching the forward direction. During forward search no extra data has to be stored. Again we use the Highway Hierarchies query algorithm and modify it by introducing additional operations when a node becomes settled. At a node v we update all tentative entries of the distance

```
forall $(s,t) \in S \times T$
    $D(s,t) \leftarrow \infty$
forall $t \in T$
    BackwardSearch($t$)
        settle($v$) : store $d(v,t)$ in $b(v)$ at node $v$
forall $s \in S$
    ForwardSearch($s$)
        settle($v$) : forall $d(v,t) \in b(v)$
                          if $(d(s,v) + d(v,t) < D(s,t))$ then
                              $D(s,t) \leftarrow (d(s,v) + d(v,t)$
```

Figure 1: Highway Hierarchies algorithm for $M \times N$ Shortest Paths. During a forward or a backward search additional operations are performed when a node becomes settled. In the code listing above those modifications are denoted below the respective call of a Highway Hierarchy search algorithm.

matrix that can be improved by a path including the current node v. We can check this by examining for every $d(v,t) \in b(v)$ if $d(s,v) + d(v,t) < D(s,t)$ where $D(s,t)$ denotes the current distance matrix entry for the source s and the target t. An overview of the algorithm is presented in Figure 1.

4.2 Optimisations

Asymmetry due to Entrance Points. The time we spend during the algorithm is not only dependent on the search space sizes. Also the time spent per node is relevant. If a lot of nodes have attached a lot of distance information, the time needed to update the distance table and to walk through distances attached to one node can dominate the query time. We tackle this issue by introducing an asymmetric approach. We reduce the number of nodes with an attached backward distance by stopping the backward searches earlier: the search can be stopped after all entrance points to the topmost level are settled. The forward search is not aborted and still expands completely in the topmost level.

To show that this technique preserves the correctness of the algorithm we consider a node $v \in \mathrm{FWS}(s) \cap \mathrm{BWS}(t)$ lying on a shortest path P from s to t. If v is settled during a backward search with the new entrance point restriction, everything works as usual. If not, we consider the entrance point $w \in V'_L$ to the topmost level, that is a predecessor of v in the shortest path tree originating at t. We know that $w \in \mathrm{FWS}(s)$ because the forward search not aborted. Hence, we have a node on the shortest path P from s to t that is found by both, the forward and the backward search and thereby we know that the correct shortest path distance is written to the distance matrix.

Level Restriction. The asymmetry, occurring with the application of the entrance point restriction, is strengthened if we restrict the search to a certain maximum level γ. We do not ascend higher than that certain level. This restriction can be regarded as running the query on a graph with a Highway Hierarchies preprocessing with topmost level γ.

Together with the entrance points restriction of the preceding paragraph the level restriction further reduces the size of the backward search spaces. The forward search spaces and the overall search space grow, but we spend less time per node. So the choice of the level we abort at, is a parameter that has to deal with this tradeoff. Dependent on the problem size different abort levels are useful. The bigger the matrix is, the smaller we can choose the maximum level, because for larger matrices we have more backward distances to maintain and to compare with. We also save storage space for the backward distances in main memory which could be a limiting issue for very large matrices.

Column Maximum. Until here, we continued every forward search, until the search space is fully expanded. We can use the abort criterion of the point to point Highway Hierarchy algorithm as well to stop a forward search earlier. To do so we maintain the column maximum $\alpha(s) := \max\{D(s,t) \mid t \in T\}$ for a source node during a forward search from s. That is the maximum of the tentative distances from the column corresponding to s in the distance matrix. If the minimum element of the priority queue u during a forward search has a key $\delta(u) \geq \alpha(s)$ we know that we can not improve any of the tentative distances in the current column any more. Hence we can stop the current forward search.

Sorting Backward Distances. To further reduce the time spent per node we can sort the distances attached to nodes according to their backward distance. So we consider the backward distances $b(v)$ at a node v in ascending order. Once we regard a distance $d(v,t) \in b(v)$ with $d(s,v) + d(v,t) \geq \alpha(s)$ we are finished for this node, because the ascending order guarantees that no tentative distance can be improved for all further nodes of $b(v)$.

5 Experimental Results

We implemented a first version with the basic ideas of the presented $M \times N$ algorithm. Although this is just a rudimentary version, not including the refinements described in section 4.2, the results are promising. The implementation uses the static graph data type of LEDA 5.0 and was compiled with the GNU C++ compiler 3.4.4 using optimisation level 3. The experiments were run on a 32-Bit computer with 1024 MB of main memory and an Intel Celeron processor clocked at 2.66 GHz.

Computing a quadratic distance matrix of 100 randomly chosen nodes in the road network of Germany took more than 16 minutes with Dijkstra's Algorithm, the Highway Hierarchies $M \times N$ algorithm finished after 0.75 seconds. In a second experiment we used locations from a real world use case: 173 nodes located on the street network of the four countries Belgium, Germany, Luxemburg and the Netherlands. Here Dijkstra's Algorithm needed more than 41 minutes. Using Highway Hierarchies, the distance table was computed after 1.34 seconds. The present results of our experiments already state that the Highway Hierarchies method allows a speedup of an order of magnitudes for the calculation of a distance matrix.

References

[1] Dominik Schultes and Peter Sanders. Engineering Highway Hierarchies. *14th European Symposium on Algorithms (ESA)*, 2006.

[2] Dominik Schultes and Peter Sanders. Highway Hierarchies Hasten Exact Shortest Path Queries. *13th European Symposium on Algorithms (ESA)*, 2005.

[3] Dominik Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. *Master-Arbeit, Universität des Saarlandes*, 2005.