Space Efficient Approximation of Piecewise Linear Functions

Student thesis
at
Institute for Theoretical Computer Science, Algorithmics II
Universität Karlsruhe (TH)

of
Sabine Neubauer

supervised by:
Prof. Dr. Peter Sanders
G. Veit Batz

Abstract

In this work, we compute upper bounds of piecewise linear functions which is a special case of piecewise linear approximation. There are several application areas like pattern recognition or cartography. The goal is to reduce the number of sampling points while still preserving the characteristics of given data. The problem of approximating a given piecewise linear function with n sampling points by another piecewise linear function such that the euclidean distance between the two functions is limited by an error bound and the number of the resultant sampling points is minimum can be solved in $\mathcal{O}(n)$ time.

Therefore, we study and implement an algorithm of Imai and Iri [II87]. We describe the basic idea and all needed components in detail. In our experiments we examine the results and analyse the benefit for our application, time-dependent route planning.

Contents

1 Introduction

1.1 Motivation

Piecewise linear approximation has several applications in various fields of computer science like pattern recognition, motion planning or cartography. The goal is to reduce the complexity of given data as much as possible while still preserving the most important characteristics as the case may be the geometrically run of a street or river in the context of cartography.

Our application lies in the context of route planning. In order to offer time-dependent routing a hierarchical speedup technique, the so-called contraction hierarchies [Gei08], is generalized to networks with time-dependent edge weights [BDSV09]. The objective function in this case is travel-time. The edge costs are functions $f : \mathbb{R} \to \mathbb{R}$ represented as piecewise linear functions and fulfill the FIFO-property: $\forall \tau < \tau' : \tau + f(\tau) < \tau' + f(\tau')$, i.e. there is no overtaking. Because the complexities of these time-dependent edge weights grow with progressive contraction approximations are needed.

1.2 Problem Definition

Let $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$ be an ordered set of points in $\mathbb{R}^2$ with $x_1 < x_2 < \ldots < x_n$. Let $f : [x_1, x_n] \to \mathbb{R}$ be a function whose graph $y = f(x)$ in the xy-plane is a polygonal line connecting points $p_1, p_2, \ldots, p_n$. Then f is a *piecewise linear and x-monotone function*.

Such a function f with n sampling points should be approximated by another piecewise linear function g with a smaller number of links m. In general there are the following two kinds of problems in this context.

min-# problem: minimize m under consideration of a given error bound for the distance between f and g

min-ϵ problem: minimize the distance ϵ between f and g for a given number of links m

Furthermore, there are two categories depending on whether the sampling points of g are required to be a subset of the input function f. In the following, we focus on the min-#-problem and arbitrary approximated sampling points.

There are several ways to define the error bounds concerning the limited gap between the given function and the approximated function. On the one hand, it can be requested to compute an upper or a lower bound of the given function. But also approximations lying inside a corridor between a lower and an upper error bound are interesting. That means, it should be possible to define only one or both error bounds. On the other hand, the error bounds can be absolute or relative.

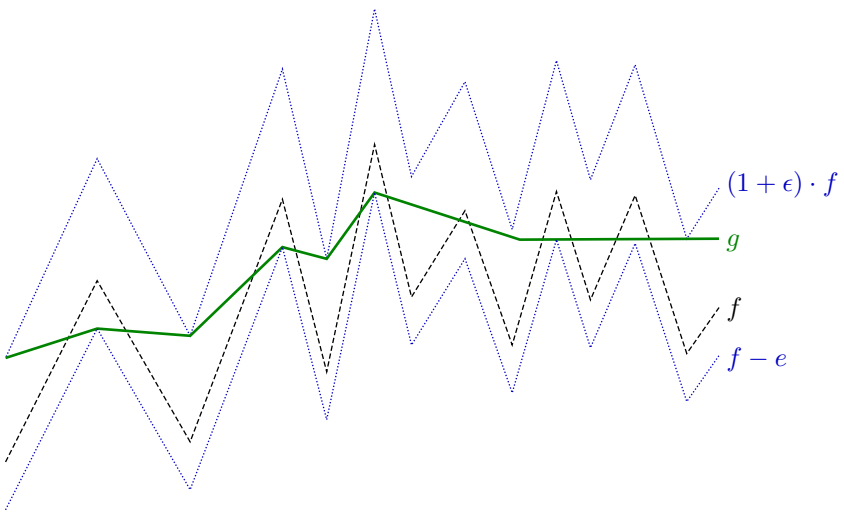

Figure 1: This figure illustrates the described model. A piecewise linear function f with $n = 14$ sampling points is plotted. This function is approximated given a relative upper error bound ϵ and an absolute lower error e. The resultant function g has $m = 8$ sampling points.

Figure 1 shows an approximation of a piecewise linear function lying inside a corridor around the given function. The upper bound is defined by a relative error ϵ whereas the lower bound is specified by an absolute error e. The given function f has $n = 14$ sampling points. This number is minimized to $m = 8$ sampling points of the approximated function g.

In the following sections, we present an algorithm which is adaptable to the different conditions. It is possible to compute upper or lower bounds as well as approximations with this algorithm. Furthermore, it can deal with relative and absolute error bounds. In this work, we focus on computing upper bounds satisfying a relative error bound. For a given piecewise linear function f and a relative error ϵ we compute a piecewise linear function $g \geq (1 + \epsilon) \cdot f$ with minimal number of sampling points which is an upper bound of f.

1.3 Related Work

Several papers concerning piecewise linear approximation of polygons have been published. One of the most famous algorithms solving this problem is the heuristic Douglas-Peucker [DP73] line simplification algorithm. But the points of the approximated function are restricted to be a subset of the input function. Suri [Sur86] described an algorithm allowing arbitrary points. The basic idea is to compute a minimum link path in a simple polygon using shortest path trees and visibility polygons. In order to achieve a runtime of $\mathcal{O}(n)$ methods of [GHL+86] are used. However, this algorithm is rather complicated.

There are also some papers focusing on the approximation of piecewise linear functions. In general, these algorithms are simpler because several characteristics of piecewise linear functions like x-monotonicity and simplicity can be utilized. On the one hand, Tomek [Tom74] gave two heuristic algorithms for this special problem. On the other hand, Imai and Iri [II87] have presented an algorithm running in $\mathcal{O}(n)$ which is optimal in matters of the output complexity. The algorithm is based on the same idea as [Sur86] and utilizes an algorithm for the convex hull problem.

1.4 Outline

We present the algorithm in [II87] and analyse the results. Furthermore, we study the advantages we can take of using this algorithm approximating the travel-time functions. In particular we make the following contributions:

- We illustrate the basic idea of the algorithm in [II87]. Moreover, we address all elements of the algorithm before formulating it and analysing the output. (Section 2)
- We give an overview of our experiments. On the one hand, we have implemented the algorithm of [II87] in order to compute upper bounds and study the results of our implementation. On the other hand, we generalize the algorithm. We compute piecewise linear approximations and show the benefit for our application, time-dependent route planning. (Section 3)

2 Algorithmic Details

2.1 Idea

The basic idea of the algorithm in [II87] is to compute a minimum link path through a tunnel. In our case, the tunnel is built by the function f itself and the upper error bound. Imagine a light source covering the entry of the tunnel. This light source illuminates a part of the tunnel and divide it in a visible part and several invisible parts. The intersection of boundaries of the visible part and the invisible part containing the exit of the tunnel is the so-called *window* where the first point of the approximated function lies. The remaining points are computed in the same way.

This idea is illustrated by Figure 2. In order to formalize it we need some definitions of different terms on visibility.

Definition 1 (visibility)**:**

point-to-point-visibility: Two points p and q are *visible*, if the line segment joining them lies inside P.

edge-to-point-visibility: A *point* p of a polygon P is *visible from an edge* e of P, if there exists a point q on e which is visible from p.

edge-to-edge-visibility: An *edge* e' is *visible from an edge* e, if there is a point on e' which is visible from e.

weak visibility polygon: The set of points in P which are visible from at least one point of the edge e is called the *weak visibility polygon* from P concerning e and is denoted by $VP(P, e)$.

Note: The visibility from e separates the polygon P into $VP(P, e)$ and several other polygons invisible from e.

Definition 1 is visualized by Figure 2. A tunnel, which is a closed polygon P, with an imagined light source covering the entry e of the tunnel is plotted. The weak visibility polygon $VP(P, e)$ from the entry e of the polygon P is shaded. Furthermore, the polygons invisible from e are denoted by $P_1, \ldots, P_5$.

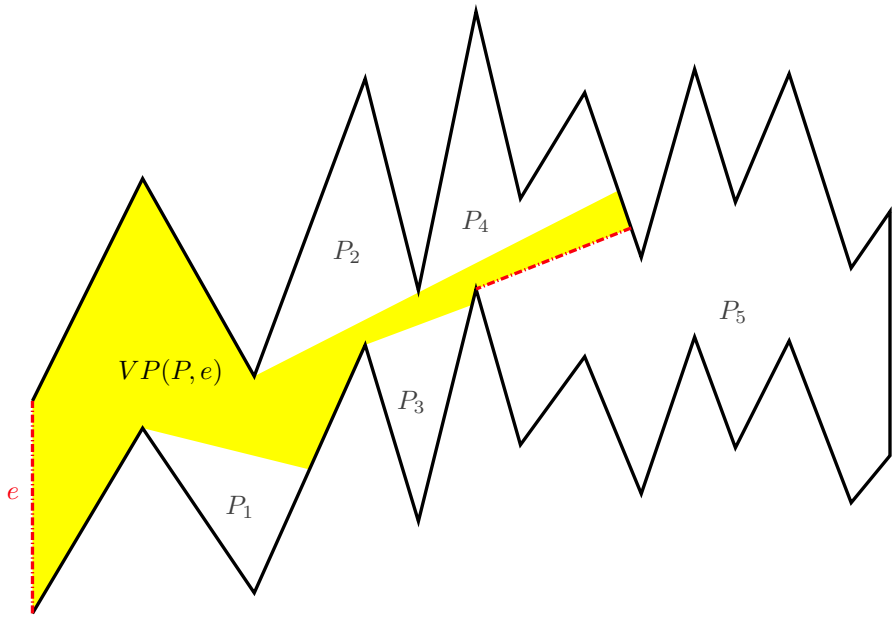

Figure 2: This figure depicts the basic idea of the algorithm. It shows the tunnel, which is a polygon P, with an imagined light source covering the entry e of the tunnel. This light source illuminates a part of the tunnel, the so-called visible polygon $VP(P, e)$, which is shaded. In this way the tunnel is partitioned in $VP(P, e)$ and several invisible parts $P_1, \ldots, P_5$. The first window determined by the intersection of boundaries of the visible polygon and the invisible part P_5 containing the exit of the tunnel is drawn. In the next step, we move the light source so that it covers this window. The remaining windows are computed in the same way.

Algorithm 1 puts the described idea in practice. First of all, the polygon P_1 which is the starting point of the algorithm is initialized and the entry of the tunnel e_1 is memorized (line 1-6). The condition in line 7 checks whether the exit of the tunnel is visible from the current edge e_i. If the exit is not visible, the next window e_{i+1} is computed. Furthermore, the remaining polygon P_{i+1} which is the base for the next step is determined and a further point of the approximated function q_i is appointed (line 8-11). If the exit of the tunnel is visible, the last two points of the approximated function are computed (line 12-13).

Algorithm 1: Basic idea

Input: function f defined by $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$, relative error bound ϵ
Output: approximated function $q_1, q_2, \ldots, q_m$

```
for j=1 to n do
    p_j^- ⟵ p_j ; // lower boundary of tunnel
    p_j^+ ⟵ (x_j, (1+ε)y_j) ; // upper boundary of tunnel
P_1 ⟵ polygon p_1^- p_2^- ... p_n^- p_n^+ p_{n-1}^+ ... p_1^+;
e_1 ⟵ [p_1^+, p_1^-];
i ⟵ 1;
while [p_n^+, p_n^-] is not visible from e_i in polygon P_i do
    e_{i+1} ⟵ window from e_i to [p_n^+, p_n^-] in polygon P_i;
    P_{i+1} ⟵ polygon invisible from e_i containing [p_n^+, p_n^-] in polygon P_i ;
    q_i ⟵ point of intersection of the line containing e_{i+1} and the edge e_i;
    i++;
m ⟵ i + 1 ;
find a point q_{m-1} on edge e_i and a point q_m on edge [p_n^+, p_n^-] which are visible from each other in polygon P_{m-1} ;
return q_1, ..., q_m
```

In [II87] Imai and Iri show that Algorithm 1 solve the given problem. They give a proof for the following theorem whereas they consider an absolute error bound.

Theorem 1. *The polygonal line $q_1 q_2 \ldots q_m$ obtained by Algorithm 1 is an approximate polygonal line with relative upper error bound ϵ having the minimum number of points.*

2.2 Convex Hull

The crucial point of Algorithm 1 is to compute the windows efficiently. Of course, one possibility is to compute the visibility polygons itself, but this leads to a runtime of $\mathcal{O}(mn \log n)$ where m denotes the output complexity and is rather slow. Hence, we use computational-geometric algorithms for the convex hull problem in order to avoid computing the visibility polygons explicitly. Thereby we achieve a runtime of $\mathcal{O}(n)$.

Definition 2 (convex hull)**:**

convex set: A set $M \subseteq \mathbb{R}^2$ is *convex*, if for any two points $p, q \in M$, and any $0 \leq \lambda \leq 1$, $\lambda p + (1 - \lambda)q \in M$.

convex hull: The *convex hull* $CH(P)$ of a polygon P is the smallest convex set containing P.

For computing the convex hull we utilize an algorithm of Sklansky [Skl72]. Bykat [Byk78] has detected that the algorithm does not always work. But Toussaint and Avis [TA82] have shown that Sklansky's algorithm works for a special class of simple polygons so-called weakly externally visible polygons. A polygonal chain like the given function belongs to this class of simple polygons.

Definition 3 (visibility)**:**

weak external visibility: A simple polygon P is called *weakly externally visible*, if for every point p on the boundary of P there exists an infinite half-line starting at p in any direction which intersects P only in p.

Note: Every point on the boundary of P is visible from some "observation" point external to P. If a simple polygon P is weakly externally visible, it can be completely surrounded by a circle.

Figure 3 illustrates Definition 3. It shows two polygons surrounded by a circle whereas the left one is not weakly externally visible in contrast to the right one. On the left hand side, there is no "observation" point o on the surrounding circle which is visible to p.

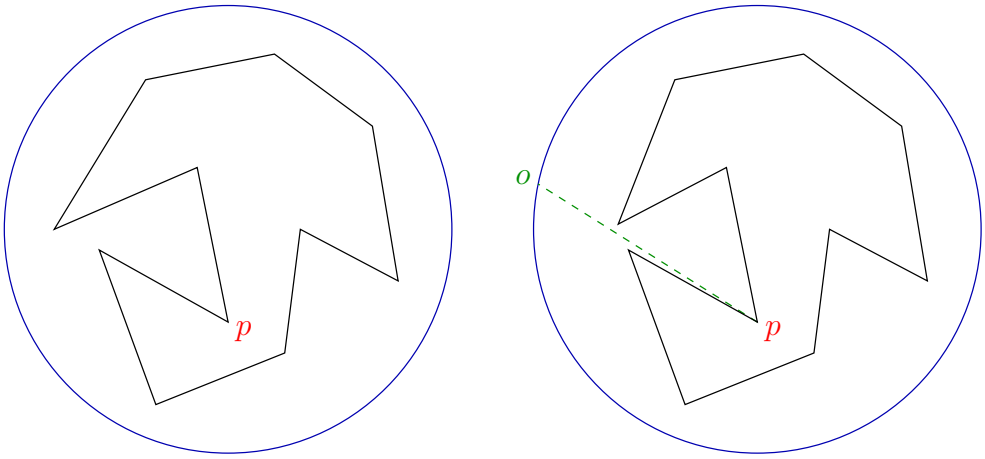

Figure 3: This figure depicts two similar polygons surrounded by a circle. The left one is not weakly externally visible, because it exists no infinite half-line starting at p in any direction which intersects the polygon only in p. On the right hand, the point p is visible from the "observation" point o on the surrounding circle. The right polygon is weakly externally visible.

Algorithm 2: CH-Polygon

Input: vertices of polygon P $p_1p_2 \dots p_n$ in clockwise order
Output: convex hull of P

```
1  j ⟵ index of vertex with minimum y coordinate;
2  k ⟵ j ;
3  finished ⟵ false ;
4  while not finished do
5      S ⟵ C(p_{(k+1) mod n}); // convexity indicator of vertex p_{(k+1) mod n}
6      if S > 0 then          // p_{(k+1) mod n} is a convex vertex
7          if (k + 2) mod n == j then
8              finished ⟵ true;
9          else
10             k++;
11     else                   // p_{(k+1) mod n} is a concave vertex
12         delete(p_{(k+1) mod n});
13         if k mod n ≠ j then
14             k−−;
15         else
16             k++;
17 return P;
```

The formulation of Algorithm 2 is based on [TA82]. The algorithm checks for each vertex successively whether it is part of the convex hull or not. The convexity indicator of vertex $p_{(k+1) \bmod n}$ in line 5 specifies whether it is about a right turn (convex vertex) or a left turn (concave vertex). It is defined as follows:

$$\begin{aligned} C(p_{(k+1) \bmod n}) = &(y_{(k+1) \bmod n} - y_{k \bmod n})(x_{(k+2) \bmod n} - x_{(k+1) \bmod n}) \\ &+ (x_{k \bmod n} - x_{(k+1) \bmod n})(y_{(k+2) \bmod n} - y_{(k+1) \bmod n}) \end{aligned} \tag{1}$$

The value of the convexity indicator is assigned to the variable S. The sign indicates whether the angle at $p_{(k+1) \bmod n}$ is convex. If $S > 0$ the angle is convex whereas when $S < 0$ it is concave. If $S = 0$ the three points $p_{k \bmod n}$, $p_{(k+1) \bmod n}$ and $p_{(k+2) \bmod n}$ are collinear and the point $p_{(k+1) \bmod n}$ is definitely no vertex of the convex hull of P.

Figure 4 shows the result of Algorithm 2 applied to the weakly externally visible polygon of Figure 3. The algorithm starts at point p_6 and checks for each vertex of the polygon in clockwise order whether it is convex. The points p_4, p_8 and p_9 are deleted because they are concave vertices. The remaining vertices form the convex hull of the given polygon.

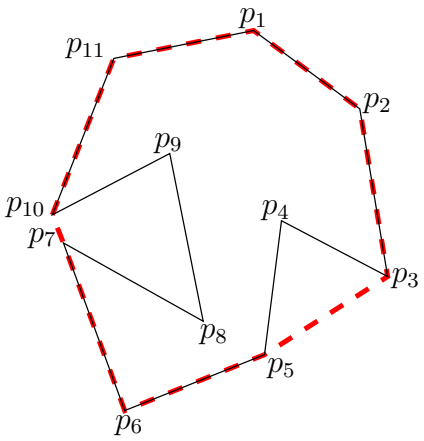

Figure 4: This figure presents the result of Algorithm 2 applied to the weakly externally visible polygon of Figure 3. The algorithm starts at point p_6 and checks for each vertex of the polygon in clockwise order whether it is convex. The points p_4, p_8 and p_9 are concave vertices. Hence, they are not part of the dashed polygon which depicts the determined convex hull.

Toussaint and Avis [TA82] show the correctness of Algorithm 2 for weakly externally visible polygons. This result is formulated in Theorem 2.

Theorem 2. *Given a weakly externally visible polygon P', Algorithm 2 finds the convex hull of P'.*

2.3 Windows

As described in Algorithm 1 we iteratively compute windows in order to find the points of the approximated function. Of course, it is possible to determine the windows by computing the visibility polygons explicitly. But there are some geometric properties of windows concerning the convex hulls of the function itself and the upper error bound which we utilize to compute the windows more efficiently. These properties are taken from [II87].

Definition 4 (window)**:**

window: The *window* in P from an edge e to another edge e' is defined as the intersection of boundaries of $VP(P, e)$ and the invisible polygon P' containing e'.

Let $[p^+, p^-]$ be a window with $p^+ \in [p^+_{i-1}, p^+_i]$ and $p^- \in [p^-_{j-1}, p^-_j]$. Let the polygon P' be built by the upper boundary $p^+ p^+_i p^+_{i+1} \dots p^+_n$ and the lower boundary $p^- p^-_j p^-_{j+1} \dots p^-_n$. The goal is to determine the window from edge $[p^+, p^-]$ to $[p^+_n, p^-_n]$ in P'.

For a piecewise linear function g given by the polygonal line of $u_1 u_2 \dots u_k$ with $u_i = (x_i, y_i)$ $CH^+(u_1 \dots u_k)$ denotes the convex hull of the region $\{(x, y) | y \geq g(x), u_1 \leq x \leq u_k\}$ and $CH^-(x_1 \dots x_k)$ denotes the convex hull of the region $\{(x, y) | y \leq g(x), x_1 \leq x \leq x_k\}$.

For three points u, v and w the angles $\angle^+ uvw$ and $\angle^- uvw$ are defined as in Figure 5. The angle $\angle^+ uvw$ identifies the angle between the edges $[v, u]$ and $[v, w]$ measured counterclockwise. The angle $\angle^- uvw$ is defined as the angle between the edges $[v, u]$ and $[v, w]$ measured clockwise.

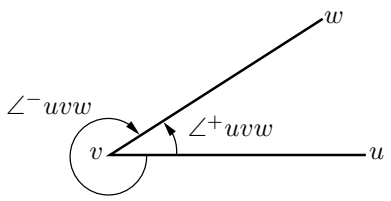

Figure 5: Definition of $\angle^+uvw$ and $\angle^-uvw$

The following two lemmas concerning the correlation of convex hulls and windows are extracted from [II87].

Lemma 1. *For $k \geq \max\{i, j\}$, the following three propositions are equivalent.*

(i) $[p_k^+, p_k^-]$ *is visible from* $[p^+, p^-]$

(ii) $CH^+(p^+ p_i^+ p_{i+1}^+ \dots p_k^+)$ *and* $CH^-(p^- p_j^- p_{j+1}^- \dots p_k^-)$ *have no common interior point (but they may touch each other)*

(iii) there are two separating lines (which coincide with each other in case the two convex hulls touch each other at more than one point) of $CH^+(p^+ p_i^+ p_{i+1}^+ \dots p_k^+)$ *and* $CH^-(p^- p_j^- p_{j+1}^- \dots p_k^-)$ *each of which supports the two convex hulls at points* r^+ *and* l^- *or at* r^- *and* l^+*, respectively.*

Figure 6 visualize the introduced terms of Lemma 1. Depending on the edge $[p^+, p^-]$ the convex hulls, the supporting points and the separating lines $[r^+, l^-]$ and $[l^+, r^-]$ are plotted exemplarily.

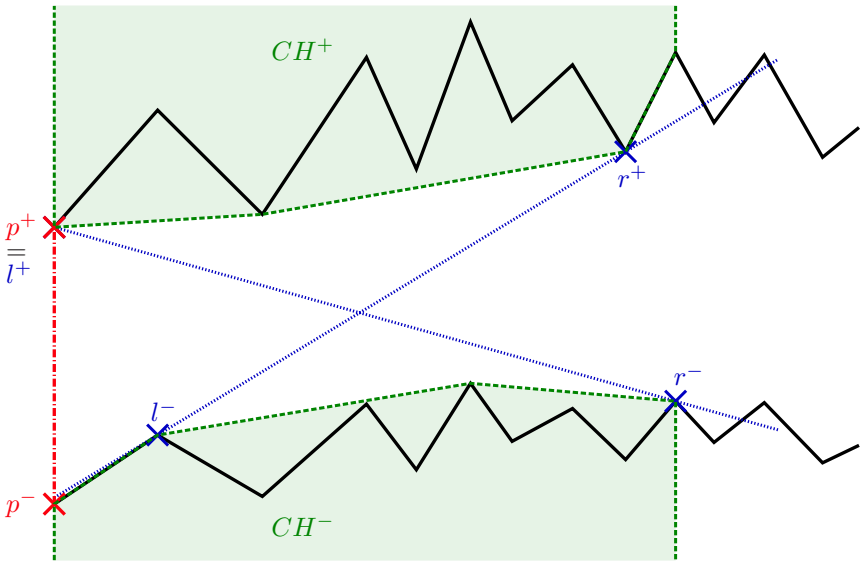

Figure 6: This figure illustrates the introduced terms depending on the edge $[p^+, p^-]$. CH^+ and CH^- denote the convex hulls and are depicted as shaded areas. r^-, r^+, l^- and l^+ identify the supporting points. The two separating lines are dotted.

Lemma 2. *For $k \geq \max\{i, j\}$, let us suppose that $CH^+(p^+p_i^+p_{i+1}^+ \dots p_k^+)$ and $CH^-(p^-p_j^-p_{j+1}^- \dots p_k^-)$ have no common interior point and let r^-, r^+, l^- and l^+ be the four supporting points as defined in Lemma 1. Then, we have*

(i) if $\angle^+ p_{k+1}^+ l^+ r^- \geq \pi$ and $\angle^- p_{k+1}^- l^- r^+ \geq \pi$, then $CH^+(p^+p_i^+p_{i+1}^+ \dots p_{k+1}^+)$ and $CH^-(p^-p_j^-p_{j+1}^- \dots p_{k+1}^-)$ have no common interior point

(ii) if $\angle^+ p_{k+1}^+ l^+ r^- < \pi$, then $CH^+(p^+p_i^+p_{i+1}^+ \dots p_{k+1}^+)$ and $CH^-(p^-p_j^-p_{j+1}^- \dots p_{k+1}^-)$ have a common interior point, and the window is $[r^-, v]$ where v is the point of intersection of line l^+r^- and $[p_k^+, p_{k+1}^+]$

(iii) the proposition obtained by interchanging superscripts $+$ and $-$ in (ii)

2.4 Algorithm

Based on the summarised results in the previous subsections we now formulate the complete algorithm based on [II87]. Algorithm 3 computes a piecewise linear function $g : [x_1, x_n] \to \mathbb{R}$ with minimum number of sampling points and $f(x) \leq g(x) \leq (1+\epsilon)f(x) \forall x \in [x_1, x_n]$ for another given piecewise linear function $f : [x_1, x_n] \to \mathbb{R}$ and an upper error bound ϵ. Starting with $p^+ = p_1^+$ and $p^- = p_1^-$ we check for each further point p_k whether $CH^+(p^+ \dots p_k^+)$ and $CH^-(p^- \dots p_k^-)$ have a common interior point. If the convex hulls intersect, we determine the window from $[p^+, p^-]$ to $[p_n^+, p_n^-]$ by using Lemma 2 and p^+ and p^- become the points of the window. Furthermore, we utilize the separating lines to ascertain the first point of the approximated function. Then we repeat these steps until $[p_n^+, p_n^-]$ is visible from $[p^+, p^-]$. Finally, we identify the last two points of the approximated function on the last two windows.

The update of the convex hulls is performed by line 13-18. The correctness of this step follows from the correctness of Algorithm 2. The condition $\angle^* p_i^* pt^*(p) > \pi$ in line 15 of Algorithm 3 is equivalent to the condition $S > 0$ in line 6 of Algorithm 2. Therefore, all conditions in Algorithm 3 whether an angle is smaller or greater than π can be evaluated without utilizing any trigeometric function. We only have to identify the position of one point to a line. This is similar to the convexity indicator specified in equation 1.

By utilizing Lemma 2 we check whether CH^+ and CH^- intersect or not. If CH^+ and CH^- have a common interior point, we compute a further point of the approximated function and the resulting window (line 19-32). After updating all variables, we continue with computing the next approximated point. If CH^+ and CH^- have no common interior point, we need to update the supporting points and separating lines (line 33-37) and proceed by extending the convex hulls by one further point of the tunnel. If the convex hulls cover the whole functions, we define the two last points of the approximated function (line 38-40).

The correctness of Algorithm 3 results from the correctness of the presented results in the previous subsections especially Algorithm 2 and Lemma 2. Because of the properties of the given function like x-monotonicity, the algorithm obviously runs in $\mathcal{O}(n)$.

The following theorem summarizes the above results. It is derived from [II87] whereas Imai and Iri consider the case of an absolute error bound.

Theorem 3. *An approximate polygonal line $p_1p_2 \dots p_n$ with relative upper error bound ϵ with the minimum number of points can be found in $\mathcal{O}(n)$.*

Algorithm 3: Approximate PWL-function

Input: function f defined by $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$, error bound ϵ
Output: approximated function $q_1, q_2, \ldots, q_m$

```
// Initialization
for j=1 to n do
    p_j^- ⟵ p_j;
    p_j^+ ⟵ (x_j, (1+ε)y_j);
foreach ∗ ∈ {+, −} do
    p^∗ ⟵ p_1^∗;
    l^∗ ⟵ p_1^∗ ; // supporting points
    r^∗ ⟵ p_1^∗;
    s^∗(p_1^∗) ⟵ p_2^∗ ; // right-adjacent point of p_1^∗ in CH^∗
    t^∗(p_2^∗) ⟵ p_1^∗ ; // left-adjacent point of p_2^∗ in CH^∗
j ⟵ 1;
nextWindow ⟵ false ;
for i = 3 to n − 1 do
    // Updating the convex hulls
    foreach ∗ ∈ {+, −} do
        p ⟵ p_{i−1}^∗;
        while p ≠ p^∗ and ∠^∗ p_i^∗ p t^∗(p) > π do
            p ⟵ t^∗(p);
        s^∗(p) ⟵ p_i^∗;
        t^∗(p_i^∗) ⟵ p;
    // Checking whether the convex hulls intersect or not
    foreach (∗, ⋄) ∈ {(+, −), (−, +)} do
        if not nextWindow and ∠^∗ p_i^∗ l^∗ r^⋄ < π then
            q_j ⟵ point of intersection of line l^∗ r^⋄ and [p^∗, p^⋄] ;
            j++;
            p^⋄ ⟵ r^⋄;
            p^∗ ⟵ point of intersection of line l^∗ r^⋄ and [p_{i−1}^∗, p_i^∗];
            s^∗(p^∗) ⟵ p_i^∗;
            t^∗(p_i^∗) ⟵ p^∗;
            r^∗ ⟵ p_i^∗;
            r^⋄ ⟵ p_i^⋄;
            l^∗ ⟵ p^∗;
            l^⋄ ⟵ p^⋄;
            while ∠^⋄ l^⋄ r^∗ s^⋄(l^⋄) < π do l^⋄ ⟵ s^⋄(l^⋄);
            nextWindow ⟵ true;
    // Updating the supporting points and separating lines
    if not nextWindow then
        foreach (∗, ⋄) ∈ {(+, −), (−, +)} do
            if ∠^∗ p_i^∗ l^⋄ r^∗ < π then
                r^∗ ⟵ p_i^∗;
                while ∠^∗ p_i^∗ l^⋄ s^⋄(l^⋄) < π do l^⋄ ⟵ s^⋄(l^⋄);
// Compute the two last approximated points
m ⟵ j − 1;
q_{m−1} ⟵ point of intersection of line l^− r^+ and [p^+, p^−] ;
q_m ⟵ point of intersection of line l^− r^+ and [p_n^+, p_n^−];
return q_1, ..., q_m;
```

2.5 Output: Upper Bound

As mentioned in Section 1 our goal is to minimize the number of sampling points of a given piecewise linear function $f : [x_1, x_n] \to \mathbb{R}$. Algorithm 3 computes an upper bound $g : [x_1, x_n] \to \mathbb{R}$ with $g <= (1 + \epsilon)f$ for a given relative error bound ϵ. The number of sampling points is minimum restricted to the x-range between x_1 and x_n. Hence, the approximated function fulfils all constraints.

In Figure 7 and 8 we plot an example taken from our application. The function defines the travel-time for a certain street section dependent from the moment of pulling into this section. The given function has 125 sampling points whereas many points lie on the lines. The determined upper bound has nine sampling points. That means, seven windows are determined. These windows are plotted in Figure 7.

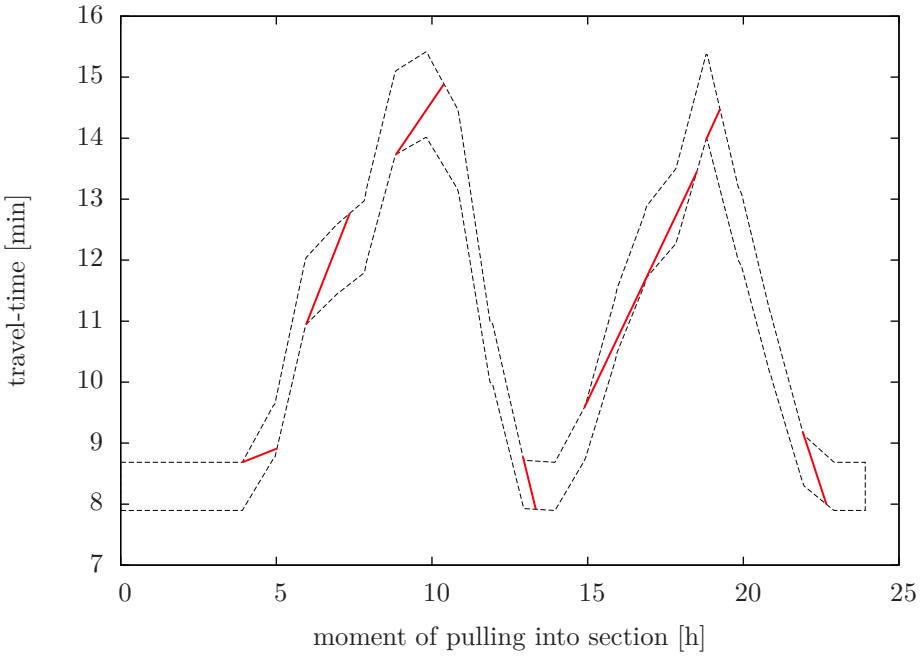

Figure 7: This figure illustrates the result of Algorithm 3. We approximate a piecewise linear function with 125 sampling points taken from our application and plot the tunnel (dashed) with the determined windows. The lower border of the tunnel is the given function which defines the travel-time for a certain street section dependent from the moment of pulling into this section.

Figure 8 also shows the example with 125 sampling points. Compared to Figure 7 this figure illustrates the approximated function with nine sampling points which is an upper bound of the given function.

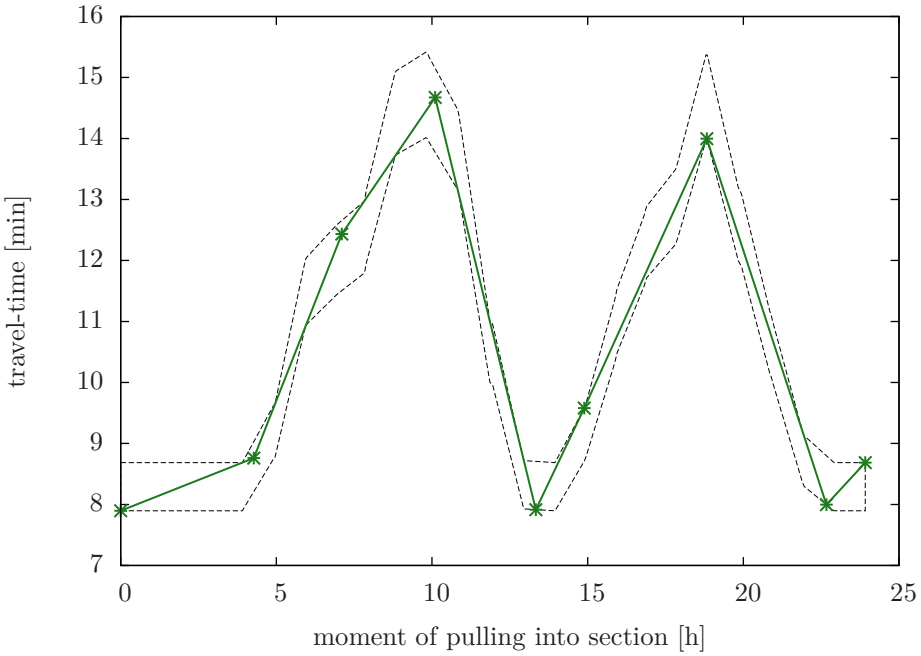

Figure 8: Same as Figure 7 but the resultant piecewise linear approximated function instead of the windows is plotted.

Furthermore, the given piecewise linear functions satisfy the FIFO-property. In our experiments Algorithm 3 always preserves this property on the computed upper bounds, but we do not have yet a proof for the following Hypothesis 1.

Hypothesis 1. *If the input function fulfils the FIFO-property, the approximated function, which is an upper bound of the input function, also satisfy the FIFO-property.*

3 Experiments

3.1 Environment

All experiments were conducted using one core of a Quad Core Intel Xeon E5345 Processor running at 2.33 GHz. The machine has 16 GB RAM and is running openSuSE version 10.3 with kernel 2.6.22.17 x86_64. The programs used for the approximation (Section 3.3) were compiled using gcc version 4.2.1 with optimization level 3 whereas we used gcc version 4.2.3 for the programs evaluating the benefit for our application (Section 3.4). The CPU was running in 64bit mode. This environment is the same as used in [BDSV09].

3.2 Instances

For our experiments described in the following sections, we used a real-world time-dependent road network of Germany provided by PTV AG for scientific use. The initial point of our computation is a preprocessed time-dependent contraction hierarchy of this graph. It has approximately 4.7 million nodes and 18.9 million edges. Further information about time-dependent contraction hierarchies can be found in [BDSV09]. The network reflects the traffic at midweek (Tuesday till Thursday) which leads to $\approx$ 12% time-dependent edges. That means, the edge weights of 2 246 790 edges are given as periodic, piecewise linear functions instead of constants specifying the time it takes to travel each edge by car. Figure 9 shows the cumulative distrubution function (CDF) of the given travel-time functions' number of sampling points which ranges between 4 and 8220. About 82% (99%) of the functions have less than 100 (1 000) points. In average each travel-time function has 88.7 sampling points.

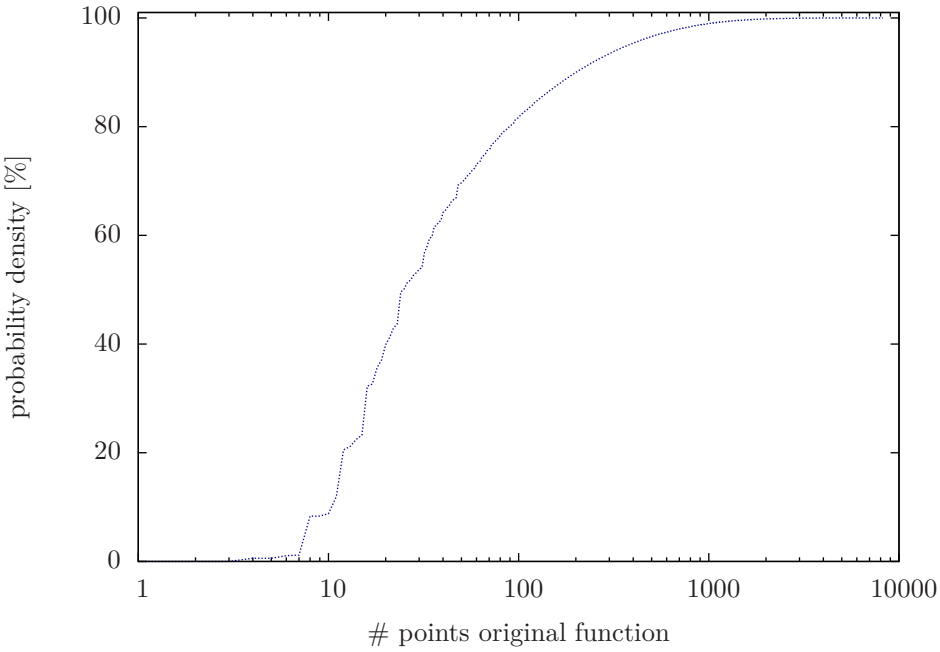

Figure 9: Plot of the cumulative distribution function (CDF) of the given travel-time functions' number of sampling points. The y-axis shows the possibility that a function have less than the given number of points.

The same network is also used in the experiments of [BDSV09]. Accordingly, the results are comparable.

3.3 Results - Computing Upper Bounds

In our experiments we approximate the travel-time functions of all given edges with more than two sampling points. All time-dependent edges of the input graph fulfill this condition. We repeat this computation for several relative upper error bounds. The results are presented in this section. In the following plots we only consider a relative upper error bound of 0.01 and 0.1. Further plots with other relative upper error bounds are depicted in the appendix section 6. The given numerical results contain all considered relative upper error bounds.

Figure 10 shows the average runtime per given point over 100 runs depending on the input complexity. Most of the values are arithmetic means because there are several travel-time functions with the same input complexity. The measured runtime is not absolutely linear in terms of the input complexity. This behaviour dues probably to memory operations.

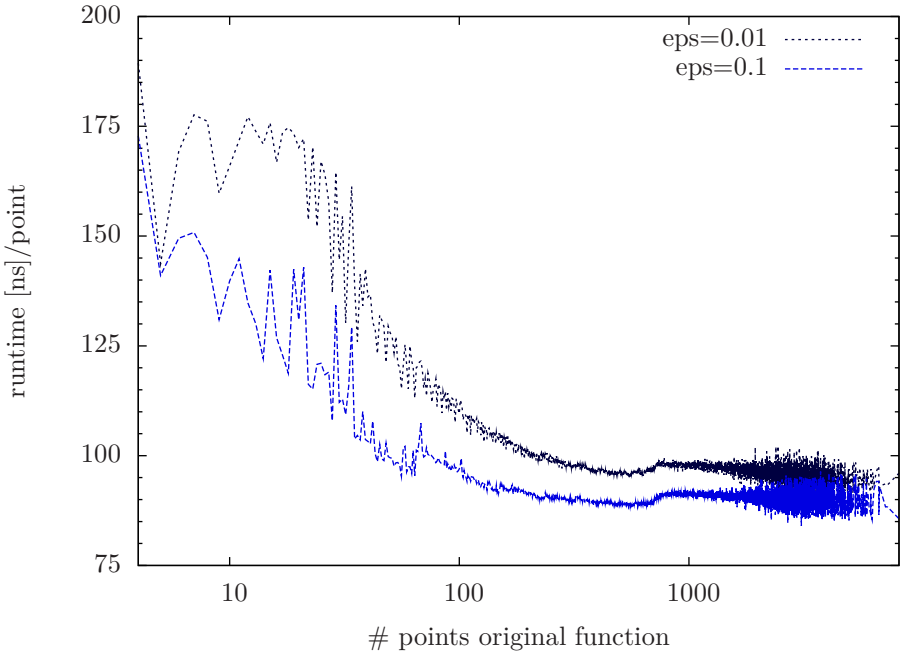

Figure 10: This figure shows the runtime per point of the original function depending on the input complexity for the relative upper error bounds 0.01 and 0.1.

Table 1 gives exact numerical results for several input complexities. For each considered relative upper error bound the total runtime per given point is denoted.

	n=10	**n=100**	**n=500**	**n=1000**	**n=2502**	**n=5000**	**n=8220**
0.010	166.0 ns	109.5 ns	96.2 ns	97.3 ns	95.3 ns	94.4 ns	96.0 ns
0.025	159.5 ns	102.7 ns	92.0 ns	94.7 ns	93.7 ns	88.9 ns	93.4 ns
0.050	149.0 ns	97.5 ns	89.3 ns	91.7 ns	89.2 ns	88.9 ns	94.1 ns
0.075	141.6 ns	95.1 ns	89.4 ns	91.1 ns	89.8 ns	90.2 ns	89.7 ns
0.100	139.8 ns	97.0 ns	89.1 ns	91.1 ns	89.1 ns	89.8 ns	85.6 ns

Table 1: Some explicit numerical results for easier comparison. For several relative upper error bounds and different input complexities n the total runtime per point is indicated.

Table 2 shows for each relative upper error bound the maximum and the minimum value of the measured runtime with the corresponding input complexities. The total runtime per point amounts between 49.6 ns and 914.1 ns.

	min total runtime	**max total runtime**
0.010	49.6 ns (10)	629.9 ns (20)
0.025	49.8 ns (10)	914.1 ns (12)
0.050	49.8 ns (10)	544.5 ns (8)
0.075	55.3 ns (10)	831.0 ns (13)
0.100	51.7 ns (10)	661.7 ns (41)

Table 2: Some explicit numerical results. For several relative upper error bounds the smallest and the greatest value of the measured runtime per point are indicated. The number in brackets identify an input complexity leading to this value. For minimum values the maximal corresponding input complexity is mentioned. For maximum values the minimum corresponding input complexity is given.

Except from the runtime we are also interested in the output complexity. In our experiments all approximated functions have less than 72 sampling points. Figure 11 illustrates the ratio of the output complexity to the input complexity. Most of the values of the output complexities are average values because several of the given functions have the same input complexity.

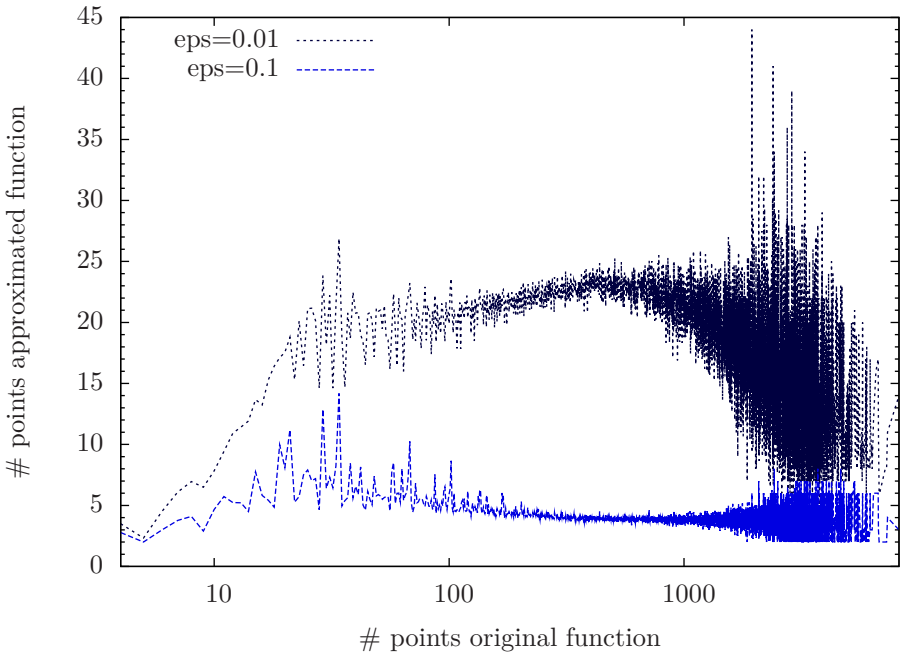

Figure 11: In this figure we plot the output complexity depending on the input complexity for an upper relative error bound of 0.01 and 0.1 respectively.

Table 3 gives exact numerical results for several relative upper error bounds. For each considered input complexity the average output complexity is denoted.

	n=10	**n=100**	**n=500**	**n=1000**	**n=2502**	**n=5000**	**n=8220**
0.010	7.8	19.9	22.5	19.9	14.3	7.0	14.0
0.025	7.1	14.6	11.7	9.8	7.3	4.0	8.0
0.050	6.1	10.0	6.3	5.6	4.3	3.0	7.0
0.075	5.1	7.4	4.6	4.1	4.0	3.0	5.0
0.100	4.7	6.0	3.7	3.5	3.0	2.0	3.0

Table 3: Some explicit numerical results for easier comparison. For several relative upper error bounds and different input complexities n the number of approximated sampling points is indicated.

Table 4 shows for each considered relative upper error bound the minimum and maximum value of the output complexity. Furthermore, the corresponding input complexity are mentioned like described above. The output complexity lies between 2 and 72.

	minimum	maximum
0.010	2 (718)	72 (207)
0.025	2 (2446)	56 (367)
0.050	2 (5048)	40 (61)
0.075	2 (6027)	34 (36)
0.100	2 (7283)	32 (36)

Table 4: Some explicit numerical results for easier comparison. For several relative upper error bounds the minimal and maximal output complexities are indicated. As in Table 2 the number in brackets identify an input complexity leading to this result.

Furthermore, we also consider the overall complexity of the used graph. It has a total of 215 858 628 sampling points as input. Table 5 gives some numerical results regarding the output complexity of the computed approximated graphs, the compression ratio and the saving for several relative upper error bounds ϵ. The output complexity lies between 29 865 037 (ϵ=0.1) and 53 308 993 (ϵ=0.01). The compression ratio ranges between 4.05 (ϵ=0.01) and 7.23 (ϵ=0.1). The saving is between 75.30% (ϵ=0.01) and 86.16% (ϵ=0.1).

	output complexity	compression ratio	saving [%]
0.000	215 858 628	1	0.00
0.010	53 308 993	4.05	75.30
0.025	45 986 640	4.69	78.70
0.050	38 375 260	5.62	82.22
0.075	33 018 612	6.54	84.70
0.100	29 865 037	7.23	86.16

Table 5: Compression ratio and saving for different relative upper error bounds.

3.4 Results - Benefit for Application

In order to analyse the benefit of approximations determined with the presented algorithm for our application we generalize Algorithm 3. We compute approximations lying in a tunnel defined by a relative upper and lower error bound ϵ. Hence, the approximated travel-times are always within a factor $1 \pm \epsilon$ from the true travel-time. We perform 1000 random s-t queries whereas the source s, target t and the starting time are picked uniformly at random. The option Stall-on-Demand was deactivated.

In the following, we study three aspects. We focus on space requirements, query times and exactness of the performed queries.

Space Requirements

For several error bounds we measured the needed space per node and determined the space overhead. The results are plotted in Figure 12.

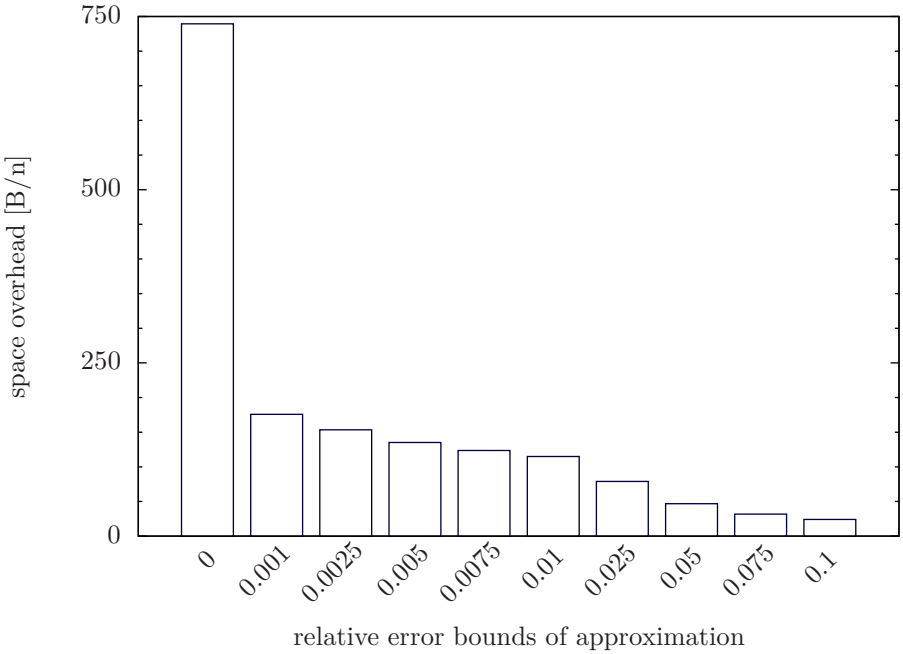

Figure 12: This figure shows the determined space overhead for several error bounds. The difference between the error bound ϵ=0 (no approximation) and the other values illustrates the saving.

In Table 6 the numerical results are indicated. The saving of space overhead in comparision to the original time-dependent contraction hierarchy ($\epsilon = 0$) is between 76.24% and 96.75%.

	output complexity	**space overhead** [B/n]	**saving** [%]
0.0000	215 858 628	739.4	0
0.0010	64 191 034	175.7	76.24
0.0025	58 226 735	153.5	79.24
0.0050	53 272 119	135.1	81.73
0.0075	50 163 074	123.6	83.29
0.0100	47 816 427	114.9	84.46
0.0250	38 038 689	78.9	89.33
0.0500	29 357 222	46.8	93.67
0.0750	25 291 908	31.8	95.69
0.1000	23 120 681	24.1	96.75

Table 6: Some explicit numerical results for easier comparison. For several relative error bounds the output complexity, space overhead and saving of space overhead are indicated.

Query Time

For the 1 000 random *s-t* queries we determine the average runtime for each approximated time-dependent contraction hierarchy and compare it with the query time on the original contraction hierarchy. The experiments show a reduction of the query times for error bounds $\epsilon \leq 0.05$. The results are presented in Figure 13.

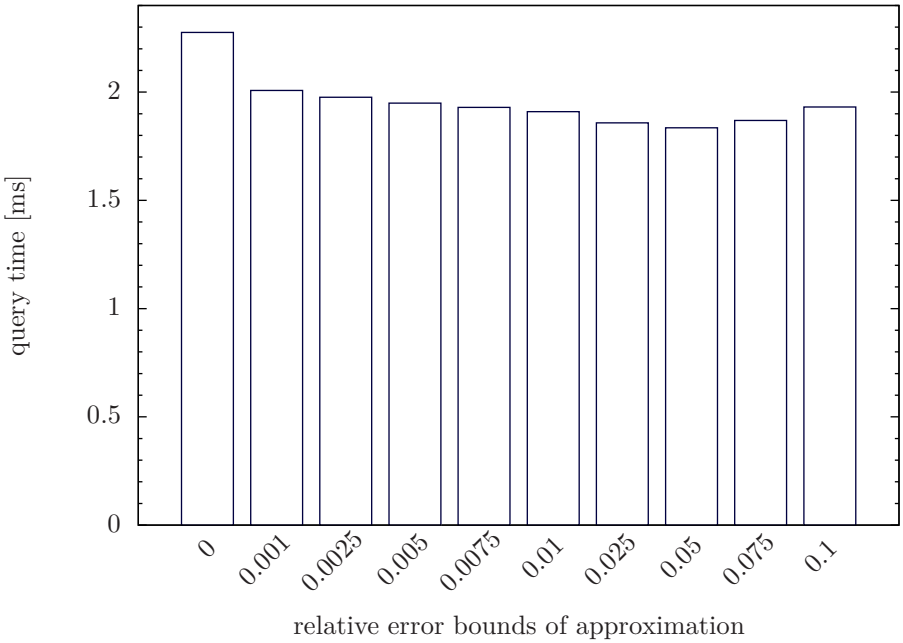

Figure 13: This figure illustrates the average query time for several error bounds. The saving is the difference between the value of ϵ=0 (no approximation) and the other values.

Table 7 gives the numerical results regarding query times. The query time profits from the approximation. It is between 2.28 ms (original contraction hierarchy) and 1.84 ms (approximated contraction hierarchy with error bound 0.05).

	query time [ms]
0.0000	2.28
0.0010	2.01
0.0025	1.98
0.0050	1.95
0.0075	1.93
0.0100	1.91
0.0250	1.86
0.0500	1.84
0.0750	1.87
0.1000	1.94

Table 7: Some explicit numerical results for easier comparison. For several relative error bounds the average query times are indicated.

Exactness

One goal of approximation is to preserve the most important characteristics as the geometrically run of the given function. This is very important in order to obtain the exactness. In our experiments we measured the differences between the computed travel-time and the exact travel-time. We determine relative and absolute values of the errors. Figure 14 shows the maximum and average absolute error for each considered approximation error bound. Figure 15 presents the same results for the relative error.

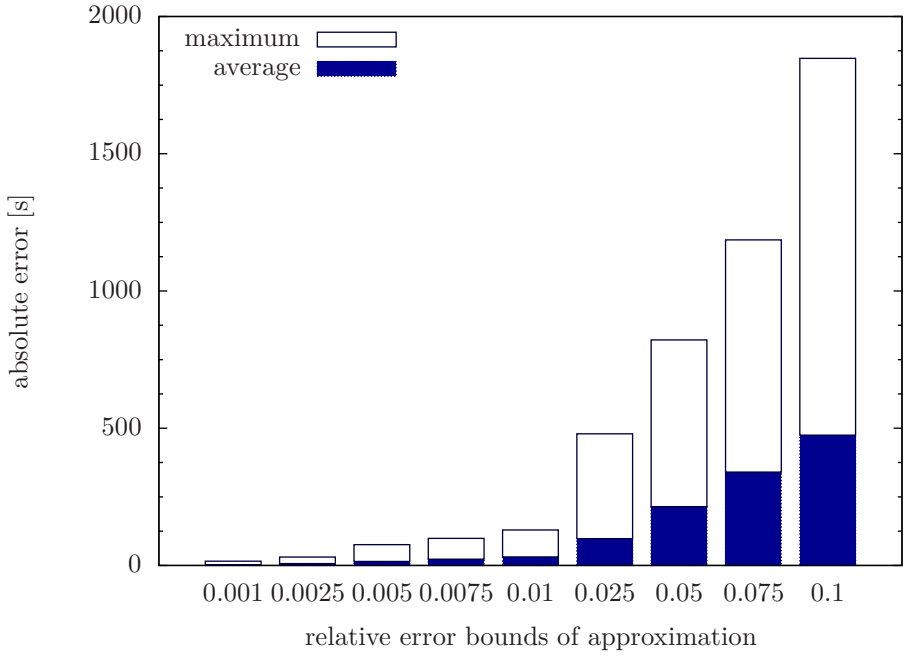

Figure 14: The absolute error of the performed queries.

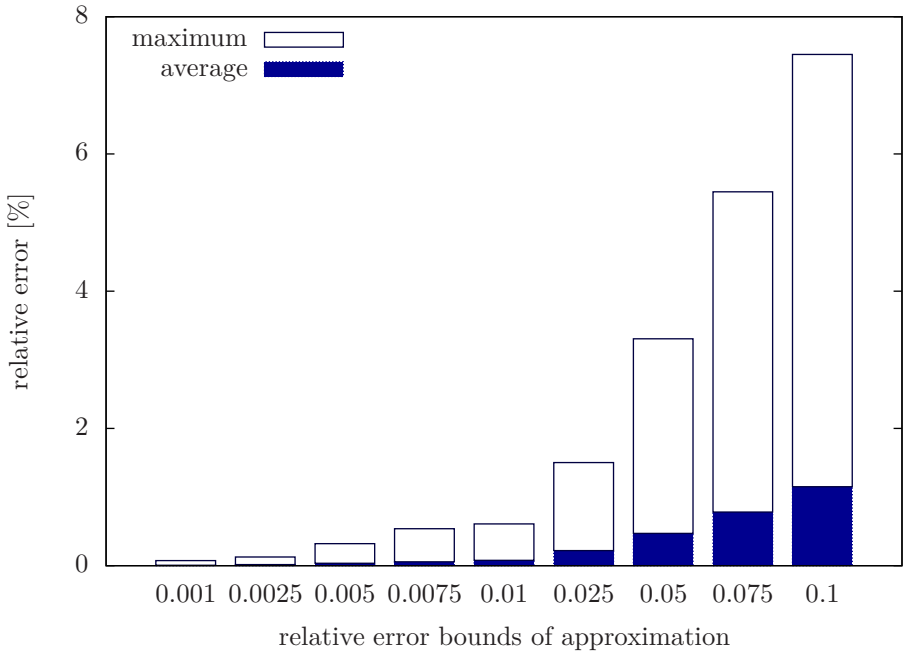

Figure 15: Like Figure 14 but with relative instead of absolute error.

Table 8 shows the numerical results concerning the exactness of the performed random s-t queries. The error increases rather fast with the error bound of the approximation. Consequently, we have to find a trade-off between travel-time functions with less sampling points and exactness of queries.

	max abs. err [s]	**max rel. err** [%]	**avg abs. err** [s]	**avg rel. err** [%]
0.0010	15.788	0.074	2.829	0.006
0.0025	30.676	0.126	7.007	0.016
0.0050	75.659	0.320	14.577	0.036
0.0075	98.452	0.539	22.958	0.057
0.0100	129.407	0.608	31.146	0.077
0.0250	479.781	1.503	97.485	0.220
0.0500	821.673	3.307	214.098	0.470
0.0750	1 186.005	5.447	340.245	0.782
0.1000	1 847.365	7.449	474.786	1.151

Table 8: Some explicit numerical results for easier comparison. For several relative error bounds used for approximation the different determined errors are indicated.

4 Future Work

We do not know yet whether the upper bounds computed by Algorithm 3 preserves the FIFO-property. In our experiments the approximated function always fulfills the FIFO-property, if the input function satisfy it and we compute upper bounds. The challenge is to proof Hypothesis 1 or to find a counter-example.

Furthermore, our experiments show that the FIFO-property is not preserved, if we compute piecewise linear functions lying inside a corridor around the given functions. If an approximated function violates the FIFO-property, it should be adjusted. This is currently not supported.

The current implementation only minimizes the output complexity. On the other hand, it is interesting to minimize also the gap between given function and approximated function. The task is to find an approximated function with minimum number of sampling points minimizing the error.

5 Conclusion

In this work, we computed upper bounds of piecewise linear functions. Therefore, we implemented a linear time algorithm of Imai and Iri [II87]. In our experiments we approximated travel-time functions of a time-dependent contraction hierarchy. We analyzed the results and study the benefit for our application, time-dependent route planning.

The experiments show that in practice the runtime of the approximation algorithm is not absolutely linear probably because of memory operations. But the complexity reduction is remarkable. Our application takes advantage of less sampling points per edge. But of course the approximation of the travel-time functions leads to a cognizable error. Consequently, we have to find a trade-off between less sampling points and exactness of queries.

6 Appendix

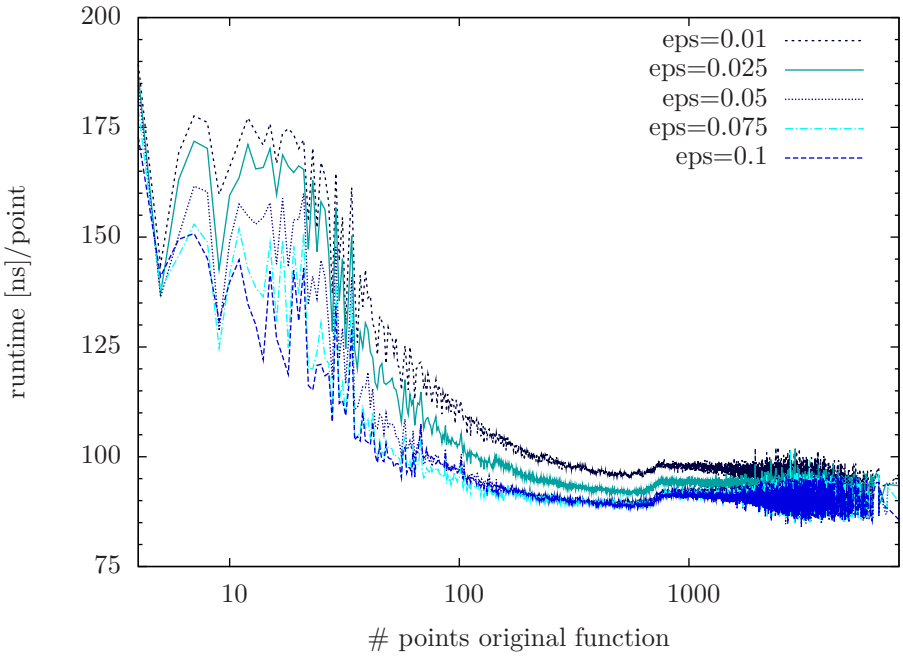

Figure 16: This figure shows as Figure 10 the runtime per point of the original function depending on the input complexity for all considered relative upper error bounds.

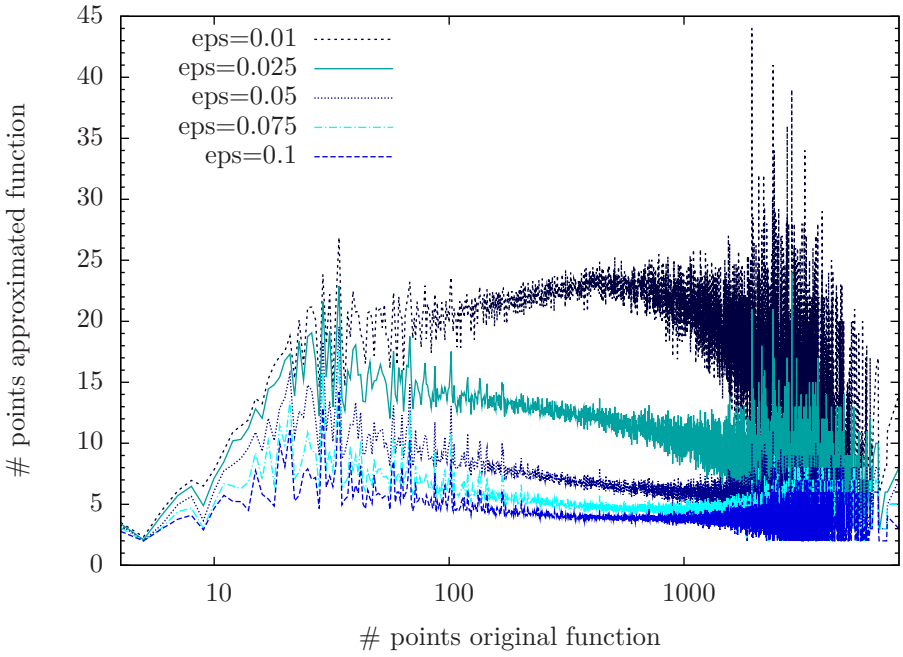

Figure 17: In this figure we plot as in Figure 11 the output complexity depending on the input complexity.

References

[BDSV09] Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, 2009.

[Byk78] A. Bykat. Convex hull of a finite set of points in two dimensions. *Inf. Process. Lett.*, 7(6):296–298, 1978.

[DP73] D.H. Douglas and T.K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10:112–122, 1973.

[Gei08] R. Geisberger. *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks.* Diploma thesis, Fakultät für Informatik, Universität Karlsruhe (TH), 2008.

[GHL+86] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 1–13, New York, NY, USA, 1986. ACM.

[II87] H. Imai and M. Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of information processing*, 9(3):159–162, 1987.

[Skl72] J. Sklansky. Measuring concavity on a rectangular mosaic. *IEEE Trans. Comput.*, 21(12):1355–1364, 1972.

[Sur86] S. Suri. A linear time algorithm with minimum link paths inside a simple polygon. *Comput. Vision Graph. Image Process.*, 35(1):99–110, 1986.

[TA82] G. T. Toussaint and D. Avis. On a convex hull algorithm for polygons and its application to triangulation problems. *Pattern Recognition*, 15(1):23–29, 1982.

[Tom74] I. Tomek. Two algorithms for piecewise-linear continuous approximation of functions of one variable. *IEEE Trans. Comput.*, C-23(4):445–448, 1974.