Scalable Ultrafast Almost-optimal Euclidean Shortest Paths

Stefan Funke , **Daniel Koch** , **Claudius Proissl** , **Axel Schneewind** , **Armin Weiß** and **Felix Weitbrecht**
University of Stuttgart, Germany

Abstract

We consider the problem of computing high-quality Euclidean shortest paths amidst obstacles on a large scale. By transferring and adapting speed-up techniques from the road network setting, we are able to compute source target paths for problem instances with several million obstacle vertices within few *milliseconds* after moderate preprocessing. We show experimentally that for small instances where optimal solutions are easily available on average our computed paths are less than 0.3% longer than the optimum. For large instances a new lower-bounding technique shows that on average our computed paths are less than 2% longer than the optimum paths. We compare our approach with the current state-of-the-art on problem instances derived from the OpenStreetMap project.

1 Introduction

The Euclidean shortest path problem (ESPP) asks for the computation of a shortest path in the Euclidean plane avoiding a set of polygonal obstacles, see Figure 1. Here, three paths from s to t are depicted with the dashed/blue one shorter than the red/dotted one and the green one. Algorithms for this problem are usually analyzed and evaluated with respect to the complexity of the problem instance, which is characterized by the number n of obstacle vertices (in this example $n = 26$). The importance of the problem is quite evident; for example, when computing optimal walking paths in a pedestrian zone with many open space areas, we are not restricted to an underlying network of walkable paths but can essentially roam freely in the Euclidean plane, of course avoiding obstacles like buildings. In highly automated plants, we might allow mobile robots to move not only on predefined trajectories but cross a factory hall on a beeline path, see [Mac *et al.*, 2016] for a survey of approaches for the robotics domain. Lastly, ignoring the effects of wind and currents, routing ships on the planet's oceans could be viewed as an instance of ESPP on the sphere.

In this paper we focus on the offline variant of ESPP, i.e., we are allowed to aggregate auxiliary information about the problem instance, such that subsequent queries between vertices can be answered as fast as possible. We put special emphasis on a scalable solution which can deal with massive problem instances with several million obstacle vertices.

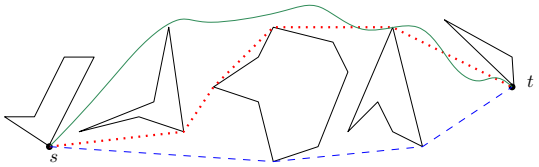

Figure 1: Instance of ESPP. Three obstacle avoiding s-t-paths.

1.1 Related Work

The efficient planning of routes in *road networks* has enjoyed tremendous attention in the research community over the last few decades. As a result, techniques have been developed that can compute provably optimal routes within *microseconds* even on continent-sized networks with hundreds of millions of vertices, for which a good implementation of Dijkstra's algorithm [Dijkstra, 1959] takes seconds. See [Bast *et al.*, 2016] for a survey. The continuous variant of the problem – ESPP – has enjoyed far less attention.

A natural way to solve ESPP is via the construction of a visibility graph, which can be computed in $O(n \log n + K)$ time where K denotes the number of edges of the resulting visibility graph, see [Ghosh and Mount, 1991]. Unfortunately, K even in practice often gets very large (in the worst case $K \in \Theta(n^2)$, so visibility-graph-based solutions tend not to scale on instances with millions of obstacle vertices. Another group of algorithms based on the idea of a continuous Dijkstra is able to beat the inherent $\Omega(n^2)$ lower bound of visibility-graph-based approaches. [Mitchell, 1996] achieved a running time of $O(n^{3/2+\delta})$, for any $\delta > 0$, which later was improved to an optimal $O(n \log n)$ in [Hershberger and Suri, 1999]. Unfortunately, these algorithms are extremely complex and we do not know of any robust implementation of them. A conceptually simple fully polynomial time approximation scheme (FPTAS) which guarantees a result of cost at most $(1 + O(\epsilon))$ times the optimum in running time $O(\frac{n}{\epsilon} \log \frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \log n))$ was presented in [Aleksandrov *et al.*, 2000]. It essentially computes a very fine discretization to meet the quality guarantee and then runs Dijkstra on that discretization. More on the practical side, Polyanya [Cui *et al.*, 2017] carefully instantiates A* on a navigation mesh and

hence does *not* require any precomputation (apart from an underlying navigation mesh). It computes the optimal shortest path and has been shown to outperform other state-of-the-art algorithms. [Shen *et al.*, 2022] combines Polyanya with so-called *compressed path databases (CPD)* [Strasser *et al.*, 2014] to obtain even faster (and still optimal) queries. Their approach requires the precomputation and storage of all-pairs-information for 'interesting' (convex) vertices, though, so this approach is not applicable for instances with millions of nodes. For dynamic settings, where even the precomputation of a navigation mesh is not feasible, [Hechenberger *et al.*, 2020] allows for efficient queries, though considerably slower than e.g. [Cui *et al.*, 2017].

1.2 Our Contribution

In this paper we show how to adapt speed-up techniques for route planning from the network-constrained domain to the Euclidean plane. Based on a (refined) constrained Delaunay triangulation as a navigation mesh, the use of Contraction Hierarchies with local path optimization strategies allows computing paths which are less than 2% longer than the optimum within a few milliseconds even on problem instances with millions of obstacle vertices. We draw upon the OpenStreetMap project to be able to generate problem instances of almost arbitrary size and develop a lower bounding technique for very large problem instances for which optimal solutions cannot be easily computed.

2 Preliminaries

2.1 Visibility-Graph-based Solution

It is quite easy to realize that an optimal path in an ESPP instance consists of consecutive straight line segments and only changes direction at convex obstacle vertices, e.g., the green path in Figure 1 cannot be optimal. This simple observation immediately leads to an optimal (in terms of output quality) algorithm for ESPP if source and target are obstacle vertices (the case which we focus on): in a preprocessing phase we construct a so-called *visibility graph* which contains as vertices all obstacle vertices. For each pair v, w of mutually visible obstacle vertices, there is an edge $\{v, w\}$ in the visibility graph of weight $|vw|$. A query between vertices s and t can then be answered by running Dijkstra's algorithm on the visibility graph. While this yields optimal paths, the main problem with this approach is its space consumption. The visibility graph can easily have $\Theta(n^2)$ edges which makes this approach impractical for non-miniscule problem instances.

2.2 Contraction Hierarchies (CH)

Contraction Hierarchies (CH) [Geisberger *et al.*, 2012] are a technique to accelerate shortest path planning in *road networks*. In a preprocessing phase, the nodes of the weighted input graph $G(V, E, c)$ are contracted one after another. The contraction process for node v consists of removing v and all of its incident edges from G and inserting so-called *shortcut edges* between former neighbors u and w of v in case the only shortest path from u to w was uvw. We then call v the skip node of edge uw and the shortcut receives the weight $c(uw) = c(uv) + c(vw)$. This ensures that at any stage of the preprocessing, the shortest path costs between all so far uncontracted nodes are still the same as in the original graph.

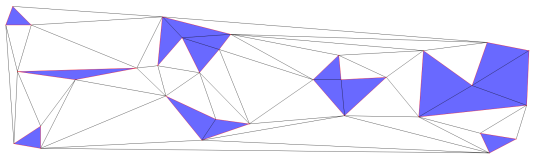

Figure 2: Constrained Triangulation of a set of (blue) polygons.

After all nodes are contracted, a new graph $G'(V, E \cup E^+, c)$ is constructed which consists of the original graph plus all shortcut edges E^+ that were created during the contractions. Furthermore, the order in which the nodes were contracted is stored by assigning each node v its rank $r(v)$ in that order. To answer shortest s-t path queries, a bidirectional Dijkstra run is conducted in the CH-graph G', where in the forward run from s only upward edges are considered, that is edges ab with $r(a) < r(b)$, and in the backward run only downwards edges with $r(a) > r(b)$. It is easy to see that this Dijkstra run settles the node with highest rank on the shortest s-t-path in both the forward and the backward run with the correct shortest path cost. If not only the shortest *distance* is of interest, one can store with each shortcut its two skipped edges during preprocessing and 'unfold' the shortest path after the bidirectional Dijkstra.

CH works correctly with any contraction order of the nodes. However, the speed-up over Dijkstra depends on the contraction order. There are very fast ordering heuristics which achieve excellent results in practice, we use the so-called *edge difference* from [Geisberger *et al.*, 2012] which reduces query times by four orders of magnitude compared to plain Dijkstra on continent-sized road networks after a few minutes of preprocessing.

There are other speed-up techniques for road networks, yet CH is still one of the most popular ones in practice due to its simplicity. Several other (even faster) speed-up schemes use CH as their basis. See [Bast *et al.*, 2016] for a survey. Note that CH does not work equally well for all graph instances, already more complex edge costs decrease the efficiency of CH by at least one order of magnitude in practice, e.g., see [Funke *et al.*, 2017].

2.3 Planar Subdivisions/Navigation Mesh

Having algorithms operate directly on a set of polygons in the plane seems somewhat challenging, as the set as such provides very little structure in terms of combinatorial proximity structure (e.g., determining nearby other polygon is non-trivial). Hence most approaches like [Aleksandrov *et al.*, 2000; Cui *et al.*, 2017] for solving the ESPP operate on a planar subdivision of the free space 'between' the obstacle polygons, sometimes also called *navigation mesh*. An example for such a planar subdivision/navigation map is shown in Figure 2 where a *constrained triangulation* of the blue polygons is depicted. A constrained triangulation is a triangulation of the polygon vertices where all polygon edges are also part of the triangulation.

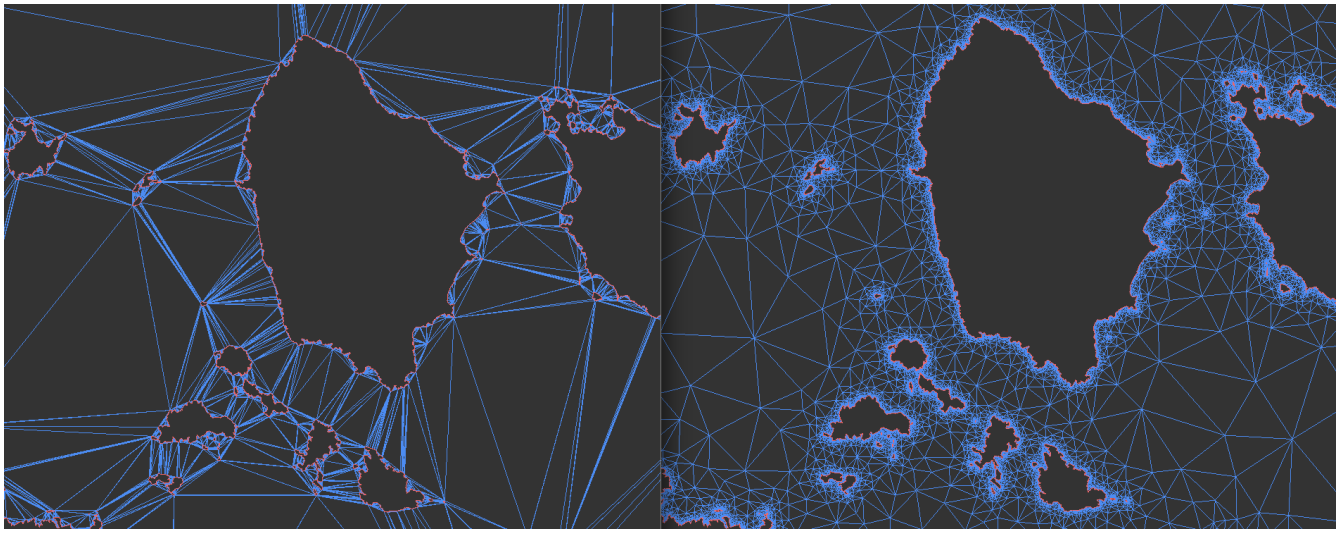

Figure 3: Constrained Delaunay Triangulation (left), after Delaunay refinement (right). Obstacle edges are red.

3 Discretization and Acceleration

The overall idea of our approach is to first compute a 'good' navigation mesh, such that computing shortest paths on that yields reasonably short paths. As shortest path computation on the mesh (simply treating the mesh as a graph with vertices and edges) resembles the network-constrained scenario, speed-up techniques like CH might be applicable.

3.1 CDT-based Discretization

A natural choice for a navigation mesh is the *Constrained Delaunay Triangulation (CDT)* [Chew, 1989] which is a triangulation of the vertex set preserving the obstacle edges, see Figure 3, left. While a CDT tries to avoid triangles with very small angles, this often cannot be avoided due to the constraining edges, which in turn leads to very thin triangles as can be seen in Figure 3, left. As we want to use the triangulation graph as a basis for our path computation, this would lead to paths which are rather different from the actual shortest path which we are aiming for. One way to ameliorate this problem is to allow the insertion of additional points into the triangulation to obtain 'well-shaped', non-skinny triangles. We employ Ruppert's algorithm [Ruppert, 1995] to obtain such a triangulation. It repeatedly inserts the circumcenters of skinny triangles as Steiner points until all triangles are well-shaped, see Figure 3, right. Due to the good-natured shape of the triangles, it seems intuitive that an optimal path can be transformed into a path in this triangulation with only slightly higher cost. The well-shaped triangles also facilitate the search for the shortest path between points that are not obstacle vertices (not the focus of this work).

3.2 CH Precomputation

We first consider the shortest path problem on the triangulation only, that is, we associate with every triangulation edge which is not in the interior of an obstacle its Euclidean distance. We then compute shortest paths between vertices using, for example, Dijkstra's algorithm. To speed-up such shortest path calculations we employ Contraction Hierarchies, see Section 2.2. It is not clear a priori, though, how well CH work for these instances, since there is not a natural hierarchy as in road networks (interstates vs. motorways vs. small highways etc.). Yet, our experiments will show that a considerable speedup of orders of magnitude compared to plain Dijkstra's algorithm can be achieved, though smaller than in the road network scenario.

So in a preprocessing step we perform the CH precomputation on the refined CDT of our problem instance. This allows us to answer subsequent queries on the triangulation graph quickly and exactly. It remains to investigate how well the resulting paths compare to the actual Euclidean shortest paths and how they could potentially be improved.

4 Path Optimization

The paths computed by the CH are quite certainly not optimal due to the loss in precision by the discretization, so in a postprocessing step we try to improve the paths computed on the navigation mesh.

The basis of these postprocessing strategies is a routine to decide whether two vertices are mutually visible. For example, in Figure 4, when trying to decide whether P_1 and P_2 are mutually visible, one starts traversing the triangulation along the segment $\overline{P_1P_2}$, in the example visiting triangles t_1, t_4, and t_5 before being blocked by an obstacle. P_1 and P_3 are seen to be mutually visible by traversal of triangles t_2 and t_3. The cost of such a check is essentially linear in the number of traversed triangles. We also use this routine to compute the visibility graph (which provides us with the ground truth of shortest path distances).

Given a path $P_1P_2 \dots P_k$ which is optimal in the discretization, we aim at decreasing its length.

4.1 Full-Optimization

This improvement strategy computes the visibility graph restricted to the node set $\{P_1, \dots P_k\}$ with the visibility edges

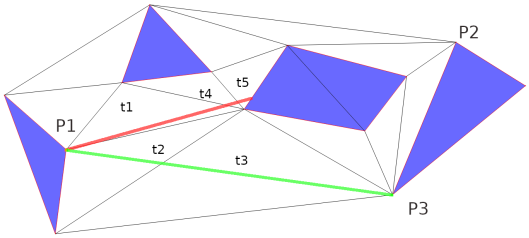

Figure 4: Visibility in a triangulated domain with obstacles (blue).

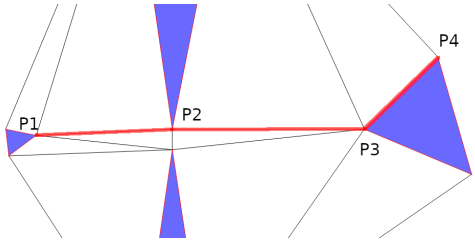

Figure 5: The path $P_1P_2P_3P_4$ from the discretization would be shortened by Greedy-Opt to $P_1P_3P_4$ whereas Full-Opt yields the better solution $P_1P_2P_4$

bearing their Euclidean distances. In this graph we run Dijkstra's algorithm. This clearly yields the shortest path connecting P_1 to P_k amongst all paths that can only turn at $\{P_1, \dots P_k\}$. Unfortunately, this does not come for free: $\Theta(k^2)$ visibility checks have to be performed (at query time) which might be too costly, in particular for large k.

4.2 Greedy-Optimization

The greedy strategy tries to avoid excessive calls to the visibility check at the cost of a worse solution quality. Starting with P_1 we determine the minimum $j_1 > 1$ such that P_1 and P_{j_1} are *not* mutually visible; clearly $j_1 > 2$. After that we repeat on the path $P_{j_1-1} \dots P_k$ etc. This clearly requires only $O(k)$ visibility checks, each of which could still be quite expensive, though. In the experiments we see that the loss in quality is not that bad, though. For an example of the greedy algorithm exhibiting worse quality than full optimization, see Figure 5.

4.3 Funnel

In [Lee and Preparata, 1984] the authors developed an algorithm (later called *funnel algorithm*) to compute the shortest path between two points within a given simple n-vertex polygon in time $O(n \log n)$, which later was adapted to compute the shortest path of a given homotopy type in a polygonal domain with obstacles. In our case we can simply interpret the union of non-obstacle triangles touched by the edges of our path as the polygon and use their algorithm to compute the optimum path from P_1 to P_k in time $O(k \log k)$. Note that in contrast to the previous two strategies, the resulting path might turn at vertices other than vertices of the path itself (in that sense, it is more flexible). On the other hand, the funnel algorithm can only return a path of the same homotopy type as the original path, which is not a restriction for the two visibility-based approaches. So funnel does not dominate the others, yet, the funnel algorithm has the advantage of a guaranteed running time of $O(k \log k)$ when improving a path consisting of k segments.

5 Shortcut-Optimized-CH (SO-CH) and Lower Bounds

While path optimization is quite effective in terms of quality of the solution, in particular its *Full-Opt* variant is quite expensive to perform at query execution, where it then dominates the overall query time as the CH-supported query is typically very fast. Hence it is natural to investigate whether the path optimization could (at least partly) be performed as a preprocessing step.

Furthermore, while we will be able to compare to the actual optimum path for smaller problem instances via the visibility-graph-based baseline, this is becomes difficult as soon as the problem instances get larger. Yet, for those instances we would also like to have at least a provable upper bound on the deviation from the optimum.

5.1 Shortcut-Optimized CH (SO-CH)

In the course of the CH construction shortcuts are created which represent shortest paths in our CDT. If we are willing to invest some more preprocessing time, we can tune the constructed CH further as follows: Consider a shortcut (u, w). If u and w are mutually visible, we simply update the cost of the shortcut (u, w) to its Euclidean distance. To make full use of the updated distances, though, we also better update shortcuts which do not have mutually visible endpoints but contain as subpath an updated shortcut. This can be achieved by considering all shortcuts ordered increasingly according to smaller level of their vertices. When processing shortcut (u, w) we first check for mutual visibility; if so, set its cost accordingly. Otherwise set the cost of (u, w) to the sum of the costs of of (u, v) and (v, w), where (u, v) and (v, w) are the edges bridged by shortcut (u, w).

While this scheme – which we call *SO-CH* – increases preprocessing times, our experimental results show that the resulting path quality (even without any optimization during query execution) improves considerably.

5.2 Lower Bound Computation

In order to lower bound the length of shortest paths between two obstacle vertices, we compute the shortest path between them in a heavily simplified problem instance, as seen in Figure 6. Simplification is achieved separately for each obstacle polygon by repeatedly removing polygon vertices (and replacing the two incident edges with only one edge) until some specified removal threshold is met: 99.5% of vertices for polygons with more than 2000 vertices, 95% otherwise.

We only cut off convex corners to guarantee that any path which doesn't intersect obstacles in the original instance also does not intersect obstacles in the simplified instance. Candidate corners to be cut off are maintained in a priority queue ordered according to the area which would be removed when cutting off the respective vertex. To cut off corners in a balanced way we add the priority with which a corner was removed to its neighboring two corners. Note that in order for

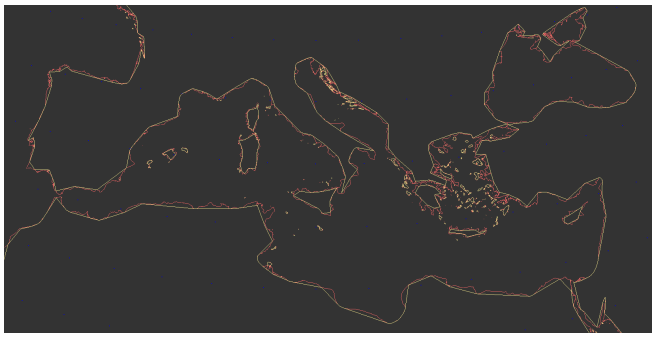

Figure 6: Simplified polygons (yellow) around the Mediterranean sea vs. full polygons (red).

a corner v to be considered convex it must not only exhibit an interior angle of at most π, but the interior of the triangle defined by the vertex and its two neighboring vertices u and w must not be intersected by any of the remaining polygon edges. The latter condition is verified by traversing a constrained triangulation of the polygon from u to w and checking for intersected constraint edges.

Finally we discard all polygons with less than 4 remaining vertices and we compute a visibility graph for the simplified instance which has reasonable space consumption due to the much lower vertex count. Now a simple Dijkstra query on the visibility graph provides lower bounds.

6 Experiments

We implemented all algorithms in C++ and evaluated them on a Ryzen 9 5900x 12-core CPU/128GB RAM running Ubuntu Linux 22.04. We use the CGAL library [The CGAL Project, 2023], in particular its geometry kernel, the exact geometric predicates, as well as the constrained Delaunay triangulation code. Unless stated otherwise, averages and maxima were calculated over 1000 trials. Source code and data sets are available on a companion page [Funke, 2024].

To be able to evaluate our algorithms on instances of (almost) arbitrary size, we use a Mercator projection of the coastline polygons of the OpenStreetMap project [The OpenStreetMap Project, 2024]. The complete *Planet* data set contains 645,628 obstacle polygons (corresponding to continents and islands) with around 15M vertices in total. For this large graph, computing the visibility graph is infeasible, so we extracted three smaller problem instances *Aegaeis*, *Medi*(terranean), and *Pata*(gonia). See Table 1 for their characteristics including the visibility graph sizes (and its parallel construction time). The visibility graph for the Pata instance is relatively small due to the high density of obstacles/islands (also see Figure 7). Quite obviously it is infeasible to compute the visibility graph for the Planet instance even with considerably more powerful hardware than what was available.

6.1 Discretization

The first step of our approach is the construction of a navigation mesh. We evaluated two strategies: an 'ordinary' CDT as well as a variant with additional Delaunay refinement as described in Section 3.1. Construction of the CDT for the Planet data set took 178s, its refinement 409s (based on the CGAL code for constructing CDT in the plane). In Table 1 we also state the number of triangles before and after refinement. For example in the Pata instance, the number of triangles increased from 1.1 million to around 3.5 million after refinement.

	Aegaeis	Medi	Pata	Planet
# vertices	207k	316k	1.0M	15.4M
# edges VisGraph	310M	721M	315M	-
time VisGraph	159s	466s	240s	-
# CDT triangles	217k	329k	1.1M	16.4M
# refined triangles	864k	826k	3.5M	58.3M

Table 1: Test instances: number of vertices, size of respective visibility graph, construction time of visibility graph (if applicable). Number of triangles in CDT and refined CDT.

		Avg.	Max.	with Greedy
Aegaeis	Unrefined	107.7%	122.2%	101.7%
	Refined	104.8%	111.5%	100.4%
Medi	Unrefined	107.8%	129.2%	101.4%
	Refined	104.8%	109.5%	100.4%
Pata	Unrefined	104.0%	113.9%	101.0%
	Refined	104.1%	108.4%	100.4%

Table 2: Relative path lengths on refined and unrefined triangulation, average and maximum, average after improvement. 100% corresponds to the length of the optimal path.

It remains to see whether refinement – and hence increased graph size – is actually worth the effort. To analyze the effect of refinement we compared shortest paths of 1000 random source target pairs with the ground truth visibility graph, see Table 2. The maximum deviation from the optimum is considerably worse for the unrefined graph. On average, the refined graph fares considerably better for Aegaeis and Medi and slightly worse on Pata. In anticipation of the results for improvement steps (Section 6.4), choosing the refined graph is even more worthwhile. Here the refined version fares consistently better. Hence from now on, all our experiments will only use the refined triangulation.

6.2 CH Acceleration

To accelerate queries on the navigation mesh we precompute a CH on the triangulation graph (refined). Table 3 shows that the number of shortcuts created is about the same as the number of original edges in the graph. This is very similar to CH constructions on road networks, the preprocessing time is considerably higher, though; for the planet graph it takes more than 30 minutes (multithreaded); on a road network of comparable size, it would take around 5 minutes. The speedup compared to Dijkstra on the largest graph is about 3 orders of magnitude (almost 2 seconds vs 0.89 milliseconds), which is about one order of magnitude less than typically experienced in the road network context. Overall we were quite pleased with the achievable speedup, even though there is no obvious hierarchy in our setting. Yet, some vertices probably appear in quite a lot of shortest paths, e.g., the Cape of Good Hope or the Strait of Gibraltar; this might suffice to make CH

	Aegaeis	Medi	Pata	Planet
#Orig.Edges	2.8M	4.2M	11.6M	185.8M
#Shortcuts	2.6M	3.9M	11.9M	195.4M
Time	16s	25s	96s	1,914s
Dijkstra	37.7ms	66.6ms	96.3ms	3,406.8ms
CH Query	0.13ms	0.15ms	0.28ms	0.89ms

Table 3: Contraction Hierarchy: Number of created shortcuts, construction time, query time vs. Dijkstra query time.

	Aegaeis	Medi	Pata	Planet
construction	3.1s	5.1s	20.5s	80.2s
CH-query	0.13ms	0.15ms	0.28ms	0.89ms
SO-CH-query	0.11ms	0.13ms	0.23ms	0.68ms
CH-query	104.8%	104.8%	104.1%	-
SO-CH-query	101.2%	101.1%	100.8%	-

Table 4: SO-CH construction time, query time and path lengths compared to optimal path.

acceleration effective.

6.3 Shortcut Optimized CH (SO-CH)

Let us now evaluate the shortcut improvement scheme. Table 4 shows construction time, query time and improvement over the standard CH. First, we can see that the construction (for a given CH) is very fast when compared to the rest of the preprocessing, namely building the CH. In terms of query times, SO-CH even improves upon standard CH queries. This might be due to the fact that fewer edges have to be relaxed. Most importantly, SO-CH results in significantly shorter paths. In all cases path lengths are reduced by at least 3% on average (no ground truth for the Planet data set).

6.4 Path Optimization

A key ingredient of our technique is the improvement of the path obtained from the triangulation graph. Three schemes were presented in Section 4: Full, Greedy, and Funnel.

The quality of the results is shown in Table 5. We compare the length of the resulting (improved SO-CH) paths to the paths returned by a standard CH as well as by SO-CH, both without improvements. For example, on the Pata data set, the standard CH returned a path about 4.1% longer than the optimum on average. SO-CH without improvement lowered that already to less than 1%, Full, Greedy and Funnel reduced that

		Aegaeis	Medi	Pata
standard-CH	Avg	104.8%	104.8%	104.1%
SO-CH	Avg	101.2%	101.1%	100.8%
Full	Avg	100.2%	100.2%	100.3%
Full	Max	102.3%	102.2%	102.1%
Greedy	Avg	100.3%	100.3%	100.3%
Greedy	Max	103.8%	102.7%	102.1%
Funnel	Avg	100.1%	100.1%	100.2%
Funnel	Max	102.2%	101.7%	100.9%

Table 5: Relative path lengths after improvement. 100% corresponds to ground truth.

	Aegaeis		Medi		Pata	
	Avg	Max	Avg	Max	Avg	Max
Full	0.6	2.8	1.3	11.1	8.2	79.0
Greedy	0.1	0.4	0.1	0.7	0.5	3.1
Funnel	0.1	0.2	0.1	0.4	0.3	1.5

Table 6: Improvement times in milliseconds

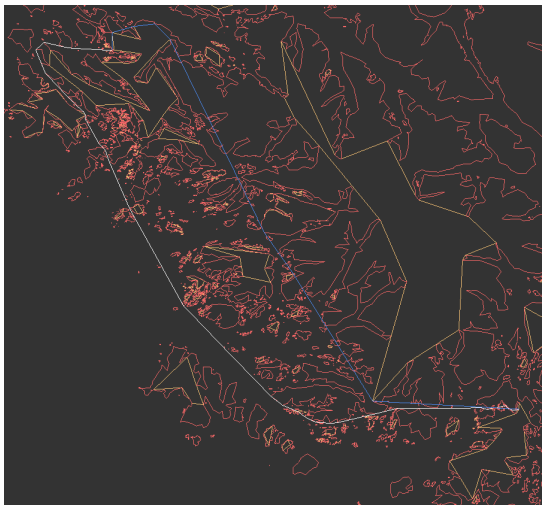

Figure 7: Example for bad lower bound: Original obstacles (red), shrunken/pruned obstacles (yellow), actual optimal path (white), optimal path in shrunken instance (blue).

even further to 0.3%, 0.3% and 0.2% respectively. Overall, in terms of average and maximum error, Funnel seems to fare best with Full getting really close. There are big differences with respect to the running times, though, as can be seen in Table 6. Even on average, Full is considerably worse than the other two approaches. This comes as no surprise as there are potentially $\Theta(k^2)$ visibility checks required, each of which might take quite long itself. In contrast to that, Funnel guarantees a near-linear running time below 2 milliseconds even in the maximum. So overall, the Funnel improvement fares clearly best amongst the improvement strategies.

6.5 Lower Bounds

As described in Section 5.2 we construct a 'shrunken' version of our problem instance to obtain lower bounds; this takes 19 seconds for the whole planet data set, which reduced the number of vertices to around $188k$ – a size amenable for the construction of the visibility graph which could be done in 9 seconds (fully using 12 CPU cores). In Table 7 we see that the resulting visibility graph has around 173 million edges. The respective smaller subgraphs have accordingly fewer nodes and edges. To assess the quality of the resulting lower bounds, we sampled 1000 source-target queries and compared their lengths in the shrunken and the original visibility graph (only possible for the smaller subgraphs). While on average the lower bound is more than 90% the length of the actual shortest path, for some queries the lower bound is as bad as only 30.7% of the actual length; this mostly stems from short range queries where the relative difference becomes large as soon

	Aegaeis	Medi	Pata	Planet
# nodes	2,760	4,078	14,176	188,269
# edges	506k	829k	1.8M	173M
Average	96.7%	96.7%	92.0%	-
Min	30.7%	48.1%	38.4%	-
Max	100.0%	100.0%	100.0%	-

Table 7: Characteristics of visibility graphs of shrunken instances. Quality of the respective lower bounds.

	Aegaeis	Medi	Pata	Planet
Quality (GT)	100.1%	100.1%	100.2%	–
Quality (LB)	104.5%	103.9%	109.5%	101.6%
SO-CH Query	0.11ms	0.13ms	0.23ms	0.68ms
Improvement	0.06ms	0.08ms	0.27ms	0.69ms
Total Query	0.17ms	0.21ms	0.50ms	1.37ms

Table 8: Quality of computed paths (vs ground truth and vs lower bound), SO-CH-query time, improvement time, total query time.

as one of the involved obstacle polygons is shrunken or even dropped. For queries inside a deeply nested area like Patagonia, our lower bounding technique also reaches its limits as can be seen in Figure 7. Here, in the shrunken instance, paths get considerably shorter due to shrunken or even pruned (due to small size) obstacle polygons. We want to emphasize that the shrunken planet data set was used to obtain lower bounds also for the small data sets, even though higher quality lower bounds could easily be achieved with still very moderate memory consumption by shrinking less.

6.6 Putting It All Together and Comparison With Existing Approaches

In Table 8 we summarize the performance of our approach (using SO-CH and the Funnel improvement) for all our data sets. Except for the largest planet data set where the construction of the visibility graph is infeasible, we compare to the actual optimum path length. For the smaller data sets the average deviation from the optimum is around 0.1% only. CH query and improvement times are together below *one millisecond*. For the large planet data set, we compare to the lower bound computed via the visibility graph of the shrunken polygons. Here we have an average deviation of 1.6% from the lower bound but conjecture that in reality we are much closer to the optimum. Note that when comparing to the lower bound only, the planet queries appear better than the ones on the smaller data sets. This is due to random queries typically being long range with large parts of the (near) optimal path traversing open space which can be easily be optimized by our improvement step. Even on this largest data set, a query can be answered in less than 2ms.

We ran Polyanya, a state-of-the-art solver for solving ESPP *exactly*, on our data sets. While running the experiments on the smaller data sets was quite hassle-free, on the largest Planet data set, we ran out of memory for many of the queries, which is why for this Planet data set, the running times were taken over only 100 queries that could be completed. For comparison: the maximum memory allocation for all steps (precomputation and query) of our pipeline is less than 64GB of RAM. In any case, running times of Polyanya were orders of magnitude slower than our approach, which is not surprising, though, as no precomputated information (apart from the navigation mesh) can be used during the query answering phase.

	Aegaeis	Medi	Pata	Planet
Average	20.5ms	70.3ms	1.1s	56.6s
Max	722.5ms	1.84s	4.7s	449.5s

Table 9: Polyanya query times.

	avg. time	max time	edges	avg. quality	max quality
Aegaeis	56.3s	57.5s	$1.3 \cdot 10^{12}$	100.7%	108.0%
Medi	69.5s	70.5s	$1.2 \cdot 10^{15}$	101.0%	107.1%

Table 10: The FPTAS approach with $\epsilon = 0.5$ on the same data sets.

In Table 10 we examined the FPTAS [Aleksandrov *et al.*, 2000] approach on our data sets. As to be expected, the FPTAS graph gets extremely large, even for a moderate choice of $\epsilon = 0.5$. There is a very bad dependence on small angles in the navigation mesh; they lead to a very fast increase of vertices and edges in the discretization. Depending on the instance, such small angles are unavoidable in spite of techniques like Delaunay refinement. Even for Aegaeis, the FPTAS graph contains an extremely large number of edges. The query times in Table 10 could only be achieved via an implicit representation of the graph (and applying further pruning techniques as described in [Aleksandrov *et al.*, 2000]). Clearly, this approach is not suitable for larger graphs even though the quality of the resulting paths is reasonably good.

7 Conclusions

We presented a preprocessing scheme that allows for the ultrafast answering of *approximate* Euclidean shortest paths queries even for large problem instances. Our approach combines Delaunay mesh refinement with speed-up techniques from the network-constrained domain, local improvement techniques as well as a lower bounding technique, which allows us to give instance-based quality guarantees even when the optimum solution is not easily computed. Compared to state-of-the-art algorithms that can compute the optimum solution for such problem sizes, we reduce the query times from *minutes* to *milliseconds* with moderate preprocessing effort. Experiments show that our results are less than 2% above the optimum solution. For smaller problem instances we can even measure a deviation of roughly 0.1% from the optimum on average. We conjecture this to hold true for the larger problem instances as well, for which we have only a lower bound to the optimum solution. We want to note that computing Euclidean shortest paths amidst obstacles in $\mathbb{R}^3$ is NP-hard [Canny and Reif, 1987]. So approaches like ours which provide (provably good) approximations might be the preferred solution strategy compared to exact solutions in dimensions higher than 2.

References

[Aleksandrov *et al.*, 2000] Lyudmil Aleksandrov, Anil Maheshwari, and Jörg-Rüdiger Sack. Approximation algorithms for geometric shortest path problems. In *STOC*, pages 286–295. ACM, 2000.

[Bast *et al.*, 2016] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering*, volume 9220 of *LNCS*, pages 19–80. 2016.

[Canny and Reif, 1987] John F. Canny and John H. Reif. New lower bound techniques for robot motion planning problems. In *FOCS*, pages 49–60. IEEE Computer Society, 1987.

[Chew, 1989] L. Paul Chew. Constrained delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.

[Cui *et al.*, 2017] Michael Cui, Daniel Damir Harabor, and Alban Grastien. Compromise-free pathfinding on a navigation mesh. In *IJCAI*, pages 496–502. ijcai.org, 2017.

[Dijkstra, 1959] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Funke *et al.*, 2017] Stefan Funke, Sören Laue, and Sabine Storandt. Personal routes with high-dimensional costs and dynamic approximation guarantees. In *SEA*, volume 75 of *LIPIcs*, pages 18:1–18:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[Funke, 2024] Stefan Funke. Algorithms Research Group. https://www.fmi.uni-stuttgart.de/alg/research/, 2024. Accessed: 2024-05-01.

[Geisberger *et al.*, 2012] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[Ghosh and Mount, 1991] Subir Kumar Ghosh and David M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20(5):888–910, 1991.

[Hechenberger *et al.*, 2020] Ryan Hechenberger, Peter J. Stuckey, Daniel Harabor, Pierre Le Bodic, and Muhammad Aamir Cheema. Online computation of euclidean shortest paths in two dimensions. In *ICAPS*, pages 134–142. AAAI Press, 2020.

[Hershberger and Suri, 1999] John Hershberger and Subhash Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM J. Comput.*, 28(6):2215–2256, 1999.

[Lee and Preparata, 1984] D. T. Lee and Franco P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[Mac *et al.*, 2016] Thi Thoa Mac, Cosmin Copot, Trung Tran Duc, and Robin De Keyser. Heuristic approaches in robot path planning: A survey. *Robotics Auton. Syst.*, 86:13–28, 2016.

[Mitchell, 1996] Joseph S. B. Mitchell. Shortest paths among obstacles in the plane. *Int. J. Comput. Geom. Appl.*, 6(3):309–332, 1996.

[Ruppert, 1995] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[Shen *et al.*, 2022] Bojie Shen, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. Fast optimal and bounded suboptimal euclidean pathfinding. *Artif. Intell.*, 302:103624, 2022.

[Strasser *et al.*, 2014] Ben Strasser, Daniel Harabor, and Adi Botea. Fast first-move queries through run-length encoding. In *SOCS*, pages 157–165. AAAI Press, 2014.

[The CGAL Project, 2023] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.6 edition, 2023.

[The OpenStreetMap Project, 2024] The OpenStreetMap Project. OpenStreetMap. https://www.openstreetmap.org/, 2024. Accessed: 2024-05-01.