Guiding the Search for the Euclidean Shortest Path Problem

Daniel Koch, Stefan Funke

University of Stuttgart
Keplerstraße 7
70174 Stuttgart, Germany
{funke, daniel.koch}@fmi.uni-stuttgart.de

Abstract

We consider the problem of reducing the search space of algorithms which solve the Euclidean Shortest Path Problem by traversing a precomputed navigation mesh. Heuristics can be used to guide this traversal. We show how upper and lower bounds to the optimal path length can be combined into an independent heuristic which considerably reduces the search space of such an algorithm. In our experiments we use our heuristic in an existing routing algorithm and find that our approach yields a substantial speedup for complicated paths.

Introduction

The Euclidean Shortest Path Problem (ESPP) asks to find the shortest path in the Euclidean plane from a source point to a target point while avoiding any polygonal obstacles encountered on the way. This problem has many applications in the real world: Routing robots in factories, guiding humans in shopping malls, finding the best way for AIs in video games, guiding ships on the oceans of the planet and many more.

All of these examples have in common that they can be modeled the same way. By placing polygons into the Euclidean plane we can represent obstacles (walls, islands, tables, etc.) and given a source and a target somewhere in the free space, we are asked to find the shortest path between the two, while also avoiding the polygons, see Figure 1 for an example.

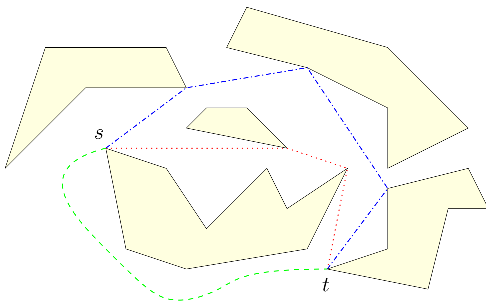

Figure 1: ESPP instance with three obstacle avoiding paths.

It is quite easy to realize that a solution to this problem only consists of straight edges between corners of polygons, which are mutually visible to each other. Assuming source and target are corners, one could create a so called *visibility graph* by connecting any two corners which have a direct sightline and simply run Dijkstra's algorithm (Dijkstra 1959) on it. While this approach works in theory, this graph can and often has a quadratic number of edges with respect to the number of corners, which makes it infeasible in practice.

Therefore other solutions have been developed, most notably an algorithm called *Polyanya* (Cui, Harabor, and Grastien 2017). It works by traversing a precomputed mesh, which subdivides the traversable space into convex polygons. Starting from the given source point, the algorithm propagates the search through the mesh in an A* manner, using the Euclidean distance to the target to guide the search. On small instances this algorithm is really fast, but it starts to struggle as the problem size grows, as shown in (Funke et al. 2024).

The Euclidean distance heuristic works quite well in many cases, but can also be quite misleading. For example think of a horseshoe, where entering it will never lead to the target, should it not be inside of it. This being only a simple example, there are many other cases where the Euclidean distance heuristic wrongly guides the search.

Our Contribution

Our goal is to guide the search more efficiently by, in a seperate step at query time, identifying the area where the shortest path cannot go through and avoiding it during the search, especially for long range queries on large instances. We do this by considering each cell of the navigation mesh and asking "Can a path starting from the source, going through this cell and ending at the target, be short enough to be the shortest path?". This entire procedure being being done before the run of the ESPP algorithm also means we can use an arbitrary ESPP algorithm and are not restricted to Polyanya.

Preliminaries

Upper Bounds

Upper bounds to the actual path length can be achieved by finding any path connecting source and target, as its length is obviously not shorter than the one of the shortest path. Still, we would like the upper bound to be as tight as possible and

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

to be calculated quickly. To that end we follow the approach of (Funke et al. 2024).

The authors first compute a constrained Delaunay triangulation (a *CDT*) (Chew 1989) of the traversable space. They achieve good approximate upper bounds by using the edges of that triangulation as a graph to route on. Techniques from road network path finding, such as contraction hierarchies (Geisberger et al. 2012), are used to accelerate these path computations, enabling queries in only a few milliseconds. Their approach only allows routing from corners, but we can extend this to arbitrary points by connecting source and target to their containing triangles.

Heuristics are used to further improve the resulting paths. In their experiments the Funnel algorithm (Lee and Preparata 1984) showed the best trade-off of runtime vs. quality, so we will use it for our purpose. This algorithm finds the shortest path in the same homotopy class as the approximate path. The authors of (Funke et al. 2024) found that paths computed with this method are empirically less than 0.3% longer than the optimal path.

Lower Bounds

To compute lower bounds on the optimal path length we can not simply compute a path on the same instance. Any path we can calculate that way would obviously not be shorter than the optimal one. Therefore to obtain lower bounds we need a different approach. We use one which simplifies the instance, following the method in (Funke et al. 2024). There the authors use a routine to iteratively cut off convex corners from the obstacles to create a new routing instance, which they nicknamed the *shrunken* instance. This method ensures that optimal paths can only get shorter since traversable space is only ever added and never subtracted. Also these instances have fewer corners, which makes it feasible to compute a visibility graph to get exact distances.

This way we can compute lower bounds between corners of the shrunken instance. To accelerate the lower bound computations, we use Hub Labelings (Cohen et al. 2003) as in (Funke et al. 2025).

Polyanya

Polyanya is an algorithm for solving the ESPP, presented in (Cui, Harabor, and Grastien 2017). As a prerequisite it needs a navigation mesh, which can be anything that subdivides the traversable space into convex regions, like a CDT. For our application we will reuse the triangulation we computed for the upper bound calculation.

The algorithm maintains a priority queue of search nodes, just like Dijkstra's algorithm. But these search nodes contain not only the current corner together with the distance we had to travel so far, but also the direction in which we have to continue the search. Just like in the A* algorithm Polyanya also computes a lower bound on how long the rest of the path is. This lower bound is essentially the Euclidean distance to the target. These two values are combined just as in A*, thereby directing the search towards the target.

When expanding a search node, Polyanya propagates the search through the adjacent mesh cell. A new search node is then created for each edge of that cell, where a possible path could continue straight on and for every corner of that cell, where a possible path would make a turn. Polyanya traverses through the mesh this way until it encounters the target, at which point it terminates.

Restricting the Mesh

Our goal is to speed up ESPP algorithms by excluding parts of the navigation mesh which are known to be irrelevant for the shortest path, hopefully reducing the search space considerably.

Our approach is the following: We first calculate an upper bound $\mathrm{UB}(s,t)$ for the path length between a source s and a target t. As described in (Funke et al. 2024) we can do that quickly and with very little error. For each triangle M in the triangulation, which also is the navigation mesh for Polyanya, we ask: "Can this triangle lay on the shortest path between s and t?". We answer this question by calculating a distance $d(s,M)$ from s to triangle M and a distance $d(M,t)$ from triangle M to t. Both these distances should lower bound the shortest distance from respectively s or t to any point in M. Whenever we have $d(s,M) + d(M,t) > \mathrm{UB}(s,t)$, we know that M cannot lay on the shortest path and we can exclude it from the subsequent shortest path computation. A triangle not exluded is call *marked* and could potentially lay on the shortest path. This process ensures that triangles on the optimal path are never excluded and therefore optimality is preserved. In the following we describe how the distances $d(s,M)$ and $d(M,t)$ are calculated.

As the Crow Flies

The first method of calculating $d(s,M)$ is to ignore any obstacles and just use the Euclidean distance. There are two cases we have to distinguish, as shown in Figure 2.

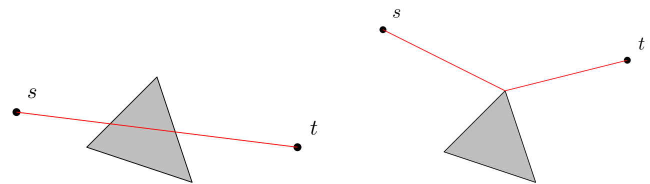

Figure 2: Euclidean lower bound when M does (left) or does not (right) intersect the line connecting s and t.

Should the line connecting s and t intersect M, the best lower bound we can get is just $d_e(s,t)$, denoting the Euclidean distance between the two points. Should the line not intersect the M, we have to calculate $\min_{i=1,2,3} d_e(s,M_i) + d_e(M_i,t)$ for all corners M_i of M as the shortest path through M now goes through one of its corners.

Lower Bounds through Shrunken Instance

While the approach outlined so far works well in many cases, especially when marking triangles with a lot of free space around them, its use is limited in more complex settings. For example think of Fjords where the Euclidean

heuristic has little use as it does not correlate with the shortest path at all. For these cases we propose another algorithm to mark relevant triangles using the shrunken setting.

We first create a simplified setting as described in (Funke et al. 2024). After triangulating this instance, we find relevant triangles for the shortest path here. We then transfer the results to the original instance by marking triangles as relevant for the shortest path should they intersect a relevant triangle of the shrunken instance. Finding relevant triangles is only feasible in the simplified setting because we can quickly calculate exact distances here.

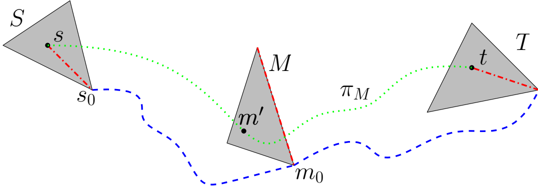

Figure 3: Marking some triangle M. Source s and target t are located inside triangles S and T. We ask for a triangle M whether it could be on the shortest path from s and t.

Figure 3 shows the process of asking "Can the shortest path from s to t go through triangle M?", and to answer this question we need to determine whether the shortest s-t-path which goes through M, say π_M, exceeds the upper bound. Calculating π_M exactly is difficult for several reasons. s and t do not need to coincide with corners of triangles, so we can not just use the visibility graph to calculate paths. We also do not know where π_M might cross M. Therefore we do not look for an exact solution, but a lower bound to the path length using the triangle inequality.

We can quickly and simultaneously compute distances from all corners of the triangle S, which contains s, to the corners of M. In fact, we can do that in the beginning for all triangles before looking to treat them individually. This now means, for each corner m_i, $i = 1, 2, 3$, of M we have calculated $\min_{j=1,2,3} d(s_j, m_i)$, where s_j are the corners of S and $d(s_j, m_i)$ denotes the length of the shortest Euclidean path between s_j and m_i.

Suppose we are interested in calculating the distance for a fixed corner m_0 of M with s_0 being the corresponding corner of S minimizing their distance. Unfortunately it does not hold that $d(s_0, m_0) + d(m_0, t_0) < d(s, t)$, with t_0 defined analogously. This is because s in general does not coincide with a corner of S and we do not know where exactly π_M passes through M. We correct for these two things using the triangle inequality. We begin with the source triangle, where the path from s_0 to m_0 through s is longer or equal to the direct path between the two:

$$d(s_0, m_0) \leq d(s_0, s) + d(s, m_0).$$

Now we do not know where π_M goes through M. Assume it goes through point m', which lays inside M. Again we can use the triangle inequality to obtain

$$\begin{aligned} d(s_0, m_0) &\leq d(s_0, s) + d(s, m_0) \\ &\leq d(s_0, s) + d(s, m') + d(m', m_0). \end{aligned}$$

Since we do not know where m' is, we can not calculate $d(m', m_0)$ exactly, but instead have to upper bound it. This can be done with the longest distance between any two points in M, which is achieved between two corners of M, let us call it l_M. This results in

$$d(s_0, m_0) \leq d(s_0, s) + d(s, m') + l_m.$$

We can repeat the same arguments for the triangle T, containing t. After adding the two inequalities we get

$$\begin{aligned} d(s_0, m_0) + d(t_0, m_0) \leq d(s_0, s) + d(s, m') + l_M \\ + d(t_0, t) + d(t, m') + l_M. \end{aligned}$$

Realizing that $d(s, m') + d(m', t) = d(s, t)$, we obtain the final bound after some rearranging

$$\begin{aligned} d(s, t) \leq d(s_0, m_0) + d(t_0, m_0) \\ - d(s_0, s) - d(t_0, t) - 2l_M. \end{aligned}$$

Finally, since m_0 was chosen arbitrarily we can calculate that bound for all corners of M and take the maximum of these values as the best lower bound.

We would like to keep $d(s_0, s)$, $d(t_0, t)$ and l_M as small as possible. These values directly depend on the triangulation, so making the triangles smaller would decrease these values. To that end we refine the triangulation using Ruppert's algorithm (Ruppert 1995), which repeatedly inserts the circumcenters of skinny or big triangles as Steiner points.

Lastly, in what we call the *mixed heuristic*, we take the maximum of the *Euclidean* and *Shrunk* heuristics.

Experiments

All our algorithms were implemented in C++, compiled with g++ 11.4.0, and executed on a Ryzen Threadripper 1950X 16-core machine with 256GB of RAM running Ubuntu Linux 22.04. For geometric computations we used the CGAL library (The CGAL Project 2023), in particular its geometry kernel, the exact geometric predicates, as well as the CDT code. Unless stated otherwise, averages and maxima were calculated over 100 trials. We did not implement the Polyanya algorithm ourselves, but instead used the implementation by the original authors (Harabor 2021).

In order to evaluate our algorithms on instances of (almost) arbitrary size, we used a Mercator projection of the coastline polygons of the OpenStreetMap project (OSM 2024). The complete planet data set contains 645,628 obstacle polygons (corresponding to continents and islands) with around 15M vertices in total. We used a shrunken instance with 0.25% corners of the original remaining.

To create queries, we picked source and target uniformly at random from the traversable space, i.e. the oceans of the planet. To find interesting queries, we forced them to be far apart, as these are the ones we are targeting. Such queries take longer and result in more complicated solutions as the chance of obstacles in the path is increased.

Preprocessing, which includes computing triangulations, visibility graphs, the contraction hierarchy and hub labeling, took in total 24 hours. When using only the Euclidean heuristic no additional preprocessing apart from the navigation mesh is needed. This only takes about 6 minutes.

Algo.	# marked shrunken triang.	# marked original triang.	time to mark
Eucl.	100,010	5,686,383	22.2ms
Shrunk	46,204	2,349,516	410.3ms
Mixed	41,241	2,125,591	411.5ms

Table 1: Average measurements of the triangle marking process. In total there are 306,613 shrunken triangles and 16,394,627 original triangles.

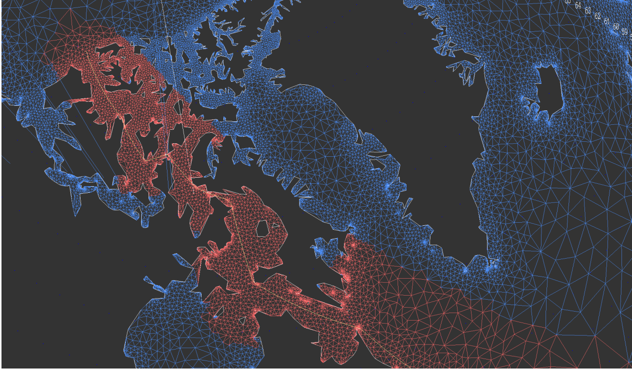

Figure 4: A typical example of the marked area (red) in the shrunken instance.

We first look at the marking process. Table 1 shows the average number of marked triangles and the time the marking process takes, while Figure 4 shows a typical example of the marked area in the shrunken instance. The results show the expected behavior, where our new approach works considerably better than the simple Euclidean algorithm. On average we only mark a sixth of all triangles while the Euclidean one marks a third. This improved quality comes at a cost though. The time to mark increases by a factor of 20, which mostly comes down to the time it takes to compute the one-to-all distances in the shrunken visibility graph as this process takes about 398ms on average. We will see in the following that this increased time is well spent and negligible in the greater scope. The mixed approach only brings a slight improvement in quality, but also does not take substantially more time than the other heuristics.

We examine the speedup of our algorithm using the different marking strategies in Figure 5 by comparing the runtime of Polyanya on the entire mesh with running the marking

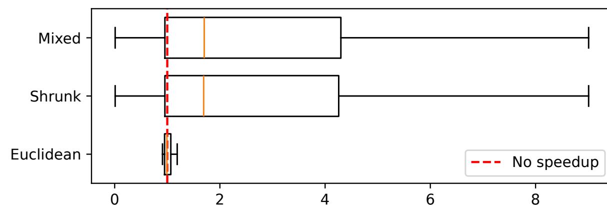

Figure 5: Speedups of the different marking methods. The box shows the first and third quartile, while the whiskers represent 1.5x the interquartile range.

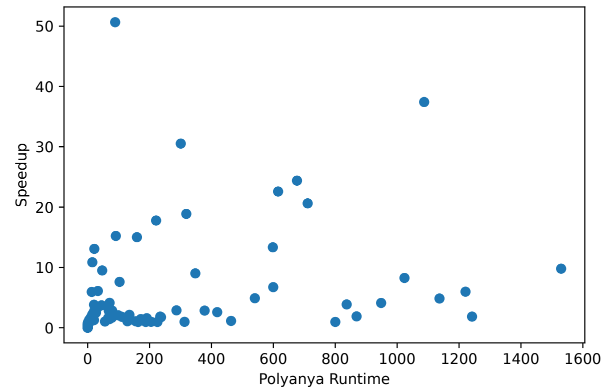

Figure 6: Plot of Polyanya runtime on the original graph against speedup.

algorithm and Polyanya on the restricted mesh.

We first consider the Euclidean marking process. It is not surprising that we do not achieve a speedup here as Polyanya guides its search using the Euclidean distance heuristic. Therefore we mostly exclude triangles Polyanya would not have looked at in the first place and do not actually reduce the search space by computing the restricted mesh. Sometimes using the Euclidean heuristic can lead to a small slow down, which can be explained by two reasons. Firstly, marking the mesh takes time, albeit very little. Secondly there can be some negative effects at the edge of the marked area. The excluded triangles might create a more complicated geometry which can lead to more Polyanya search nodes. We observed the latter in 2 out of 100 queries.

The marking processes of the other two strategies are similar in runtime, mesh size and speedup, both resulting in a median speedup of 1.7 and similar variances. This does not tell the entire story though. Figure 6 plots the runtime of the original Polyanya algorithm against the speedup achieved by the mixed heuristic. There is a cluster of queries under 300s, of which only 30% achieve a speedup over 2. Roughly 78% of queries taking longer than 300s achieve a speedup of over 2 with a median speedup of 9.4. This shows that complicated queries we are targeting achieve a good speedup. On simple queries where the Euclidean heuristic is quite good, reducing the mesh size does not create a big benefit.

Conclusion

We propose a new heuristic to guide the search of mesh based ESPP algorithms. Using a combination of upper and lower bounds we reduce the search space of such algorithms considerably. Experiments show that this reduced search space directly translates to a speedup in query time.

Further research could implement the ideas presented here directly in the search algorithms and not as a step before the actual run of that algorithm. This might result in another slight speedup as the two steps are not independent anymore.

One could also use other meshes than the CDT. Especially Polyanya could benefit from that as it performs vastly different on other types of meshes, as shown in (Cui, Harabor, and Grastien 2017).

References

2024. *OpenStreetMap*. https://www.openstreetmap.org/.

Chew, L. P. 1989. Constrained Delaunay Triangulations. *Algorithmica*, 4(1): 97–108.

Cohen, E.; Halperin, E.; Kaplan, H.; and Zwick, U. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.*, 32(5): 1338–1355.

Cui, M.; Harabor, D.; and Grastien, A. 2017. Compromise-free Pathfinding on a Navigation Mesh. 496–502.

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

Funke, S.; Koch, D.; Proissl, C.; Schneewind, A.; Weiß, A.; and Weitbrecht, F. 2024. Scalable Ultrafast Almost-optimal Euclidean Shortest Paths. In Larson, K., ed., *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, 6716–6723. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Funke, S.; Koch, D.; Proissl, C.; Staib, C.; and Weitbrecht, F. 2025. Fast and Stronger Lower Bounds for Planar Euclidean Shortest Paths. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-25*. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Geisberger, R.; Sanders, P.; Schultes, D.; and Vetter, C. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transp. Sci.*, 46(3): 388–404.

Harabor, D. 2021. Polyanya. https://bitbucket.org/dharabor/pathfinding/src/master/anyangle/polyanya/. Accessed: 2025-02-28.

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

Ruppert, J. 1995. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. *J. Algorithms*, 18(3): 548–585.

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