Fast Exact Geodesics on Meshes

Riade Benbaki
riade.benbaki@polytechnique.edu
Dariya Mukhatova
dariya.mukhatova@zimbra.polytechnique.fr

December 2020

1 Introduction

In this project, we implement the Exact Geodesics Algorithm presented by Mitchell, Mount, and Papadimitriou (MMP) on triangular meshes, for which an implementation is described in the paper by Surazhsky et al. [1]. The main idea of this implementation is to use a Dijkstra-like technique, operating on edges instead of vertices.

2 Method

The aim of the algorithm we are implementing is to find a shortest path on a triangular mesh. The main idea behind this algorithm is that it is possible to divide the entire mesh into different areas, such that points in the same area have a shortest path that goes through the same faces and vertices. Instead of looking for such a way to divide faces, we can simply work on edges, because, given a complete description of the the geodesic distance over edges, we can easily compute the geodesic distance of a point on a face, by simply taking the minimum distance over the edges composing this face.

When a shortest path crosses an edge, we can unfold the second face in the plane of the first one, and the resulting path will (should) be a straight line (otherwise it could be made shorter). In [2], it is proven that a shortest path can only go through vertices for which the sum of it's adjacent angles is superior or equal to 2π, called saddle vertices.

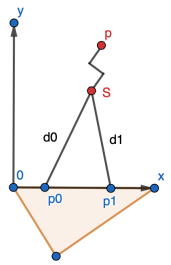

Figure 1: Window' structure

In the algorithm proposed by Mitchell, Mount, and Papadimitriou, the distance field over the edges is described by a list of windows, such that points on the same window have a geodesic path that goes through the same faces until it reaches the first vertex, which we call the pseudo-source, then follows a shortest path from the pseudo-source to the source. As mentioned above, pseudo-sources are either the source vertex, or a saddle vertex.

A window (and the distance field over it) can be fully described by a tuple of 6 variables (p_0, p_1, d_0, d_1, σ, τ). The first two variables describe the position of the interval along the edge. d_0 (resp. d_1) is the distance from p_0 (resp. p_1) to the pseudo-source s . τ is a binary variable to describe which face the path goes through. And σ is the length of the geodesic path from s back to the source p.

The algorithm consists in initially creating windows over the edges directly facing the source, and then iteratively propagating windows into opposing faces, until we have no more windows to propagate.

In the process of propagating a window, if the edge into which we propagate already has some windows, then a key operation is to only take the intersection interval in which the new window is better. This means that the propagation process can modify, delete, a create new windows.

3 Implementation

The most important step of the algorithm is propagation. It is therefore important to use a data structure that would allow us to access edges on opposing faces efficiently.

3.1 Data structures

We chose an edge based structure, where an (undirected) edge holds a list of windows on that edge. This list needs to be kept up to date through the entire run of the algorithm. In order to navigate from an edge on a face to the two edges in the opposing face, we use an adjacency map, mapping a directed edge to the third vertex in the face (in the same order). This results in a constant time for accessing the edges when a window needs to be propagated.

Another important data structure choice was the queue. As stated in [1], the algorithm works and produces the right results regardless of the order in which windows are processed. However, this influences the runtime significantly. In our implementation, we use a heap data structure, and compare the runtime with different order functions.

3.2 Window Propagation

The paper we are basing our implementation on does not show how the new propagated windows are computed, so we had to derive that on our own. Given a window $w = (p_1, p_2, d_1, d_2, \sigma, \tau)$ on the directed edge (v_2, v_1) (p_1 representing the distance from v_2 to the first end-point of the window), we need to compute the intersection w with the edge (v_2, v_3).

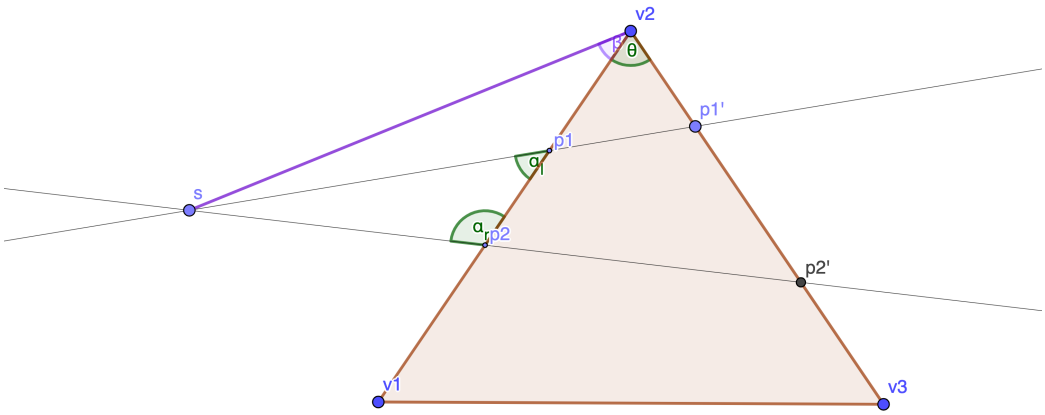

Figure 2: Window propagation

For this, we first need the angles α_l and α_r, which can be obtained by applying the Cosine Law to the triangle sp_1p_2

$$\alpha_l = \arccos \frac{(p_2 - p_1)^2 + d_1^2 - d_2^2}{2(p_2 - p_1)d_1}, \alpha_r = \arccos \frac{(p_2 - p_1)^2 + d_2^2 - d_1^2}{2(p_2 - p_1)d_2}$$

And using the Law of sines we have :

$$p'_1 = \frac{p_1 \sin(\alpha_l)}{\sin(\pi - \theta - \alpha_l)} = \frac{p_1 \sin(\alpha_l)}{\sin(\theta + \alpha_l)}, p'_2 = \frac{p_2 \sin(\pi - \alpha_r)}{\sin(\alpha_r - \theta)} = \frac{p_2 \sin(\alpha_r)}{\sin(\alpha_r - \theta)}$$

With $p'_1 = p'_2 = v_3$ if $\pi - \theta - \alpha_l \leq 0$ and $p'_2 = v_3$ if $\alpha_r \leq \theta$.

p'_1 and p'_2 are of course capped to $||v_2 - v_3||$ in case an intersection falls outside the edge. Once the two endpoints are computed, their respective distances to the pseudo-source. For a point x on the edge (v_2, v_3), it's distance to the pseudo-source is :

$$d(x, s) = \sqrt{x^2 + ||sv_2||^2 - 2x||sv_2|| \cos(\beta + \theta)}$$

With $||sv_2||^2 = d_1^2 + p_1^2 + 2d_1 p_1 \cos(\alpha_l)$ and $\beta = \arcsin(\frac{d_2 \sin \alpha_r}{||sv_2||})$.

In the implementation described in [1], edges adjacent to a saddle vertex or boundary edges need special treatment, given that a path can go through them. In our implementation, we use the approach described in [3], where instead of creating one window in the edge (v_2, v_3), we create three windows, corresponding to the three intervals $[v_2, p'_1], [p'_1, p'_2], [p'_2, v_3]$, where the pseudo-source of the left window is p_1, and the pseudo-source of the right window is p_2. The middle window has the same pseudo-source as the original window.

3.3 Windows intersection

After computing the newly propagated windows, the next step is to handle the intersection of the new windows with already existing ones.

In [1], only a generic case of intersection is mentioned. It is the case that corresponds to Figure 3, where if w_0 and w_1 are the two windows, we find the point p where the distance fields of the two windows are equal, then restrict the windows as done in the figure.

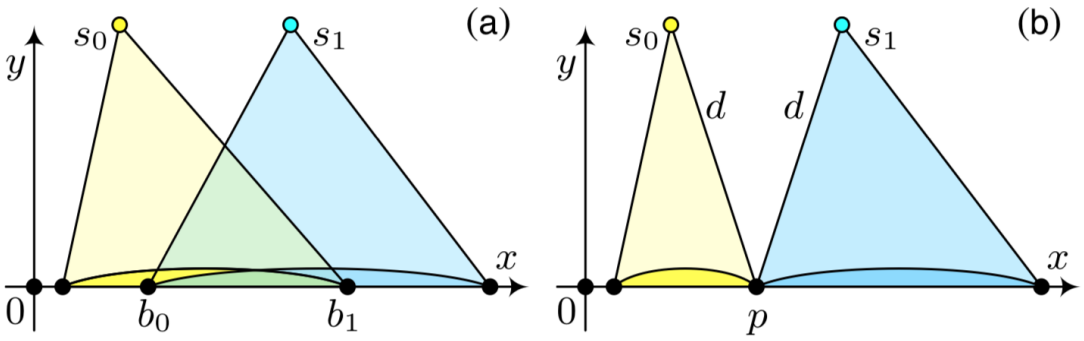

Figure 3: Generic case of intersection, from [1]

The point p is found as a solution to

$$\sqrt{(p_x - s_{0x})^2 + s_{0y}^2} + \sigma^1 = \sqrt{(p_x - s_{1x})^2 + s_{1y}^2} + \sigma^2 \tag{1}$$

which can be simplified as quadratic:

$$Ap_x^2 + Bp_x + C = 0 \tag{2}$$

$$A = (\alpha^2 - \beta^2) \quad B = \gamma\alpha + 2s_{1x}\beta^2 \quad C = 1/4\gamma^2 - ||s_1||^2\beta^2 \tag{3}$$

where

$$\alpha = s_{1x} - s_{0x} \quad \beta = \sigma_1 - \sigma_0 \quad \gamma = ||s_0||^2 - ||s_1||^2 - \beta^2 \tag{4}$$

However, this is only one possible case. The previous equation can have zero, one, two or an infinity of solutions in the intersection interval. To treat all theses cases, we chose to first divide the existing (old) window w_1 into three parts, corresponding to the left, the intersection and the right side. Of the

newly propagated window w_0, we only keep the intersection with w_1 as well (the other parts would be treated in an intersection with another window). This allows us to easily keep as an invariant through the entire run of the algorithm that a list of windows in a edge is either empty, or spans the entire edge perfectly (with no intersection).

Now that we have two window w'_1 and w'_0 spanning the same interval $[b_0, b_1]$ in the same edge, we handle all the possible cases as follows:

We assume that $w'_0.d_1 \leq w'_1.d_1$ (we can exchange them otherwise).

- The equation has zero or an infinity of solutions in the intersection range : This means that one window is always better. In this case we choose the best window by comparing the distances of endpoints and keep it, disregarding the other one (so in our case, we keep w'_0).
- The equation has two solutions $r_0 \leq r_1$ in the intersection interval : The intersection interval is split into 3 parts $[b_0, r_0], [r_0, r_1], [r_1, b_1]$, with the first and third intervals (respectively the second) having the same field as w'_0 (respectively w'_1)
- The equation has one solution in the intersection interval : we update the windows as explained in Figure 3 and [1].

Any modification done above needs to be properly managed : If an existing window is deleted (resp. modified), then we make sure to delete it (resp. modify it) in the queue, and any new window created is inserted into the list of windows in the edge as well as in the queue.

4 Results

We first try our implementation on a simple mesh (cube) in order to make sure all the computations are correct. We compare our results (distances) to the distances returns by the built-in function *igl :: exact_geodesic*. For small meshes, the results match exactly. However, as the size of the meshes grow, we start seeing some small differences, which we think are results of the heavy floating point calculations our approach uses.

We then run the algorithm on multiple meshes. For each mesh model, we use Loop subdivision algorithm in order to examine the variations of the runtime with the size of the mesh.

We compare two ways to process the windows. The first one (FIFO) consists in processing windows in the same order they are created. The second one (CEP for Closest End-Point) processes windows using the following order : $w_0 \leq w_1 \iff min(w_0.d_1, w_0.d_2) \leq min(w_1.d_1, w_1.d_2)$

Mesh	Faces	Runtime(FIFO) (ms)	Runtime(CEP) (ms)
Cube	12	0.454	0.272
	48	1.853	1.371
	192	7.227	3.698
	768	31.425	13.209
	3072	74.056	45.263
	12288	1021.51	153.432
Sphere	320	16.957	6.042
	1280	24.791	17.3
	5120	176.487	67.236
	20480	310.48	219.995
Octagon	8	0.239	0.133
	32	3.212	1.105
	128	3.889	2.845
	512	15.704	10.062
	2048	27.588	30.261
	8192	142.197	116.873

The CEP approach is clearly better and faster. In the original paper, the order function used takes compares the minimum distances over the entire edge, but computing this value (an approximation of it) is very expensive. We also observe that, as the size of the mesh grows, some windows become too small, and in some cases, imprecision in calculations lead to an inversion of the end points of the windows.

5 Conclusion

Our implementation of the MMP algorithm for exact geodesic paths uses a queue and starts by creating windows in edges facing the source and inserting them into the queue. And while the queue is not empty, we remove the first window (according to an order relation we can customize), and using 6-tuple encoding this window, we compute new windows in the edges in the opposing faces. For this, intersection between new and old windows need to be handled carefully, which is dome by handling multiple cases.

This implementation uses multiple floating-point calculations, which causes a lot of mistakes as the size of the mesh grows, and can even lead to errors on the long run, as some created windows become too small compared to the precision of floating-point arithmetic. One possible approach to avoid these problems is by using symbolic values instead of doubles, and only compute the value of a symbol when necessary. This would help in principle because we are mostly interested in comparing values, not computing them exactly, and comparing symbolic data can be optimized to reduce errors and compute only what's necessary.

References

[1] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe, "Fast exact and approximate geodesics on meshes," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 553–560, Jul. 2005. [Online]. Available: https://doi.org/10.1145/1073204.1073228

[2] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The discrete geodesic problem," 1987.

[3] D. Kirsanov, S. J. Gortler, and H. Hoppe. [Online]. Available: http://www-evasion.imag.fr/Membres/Franck.Hetroy/Teaching /ProjetsImage/2006/Bib/kirsanov$_e t_a l - rr - 2004.pdf$