Assignment 4: Geodesic distance fields [and paths]

In this exercise you will

- Build a graph weighted with geodesic distances between vertices of a mesh.
- Use this graph to compute the geodesic distance field at all vertices from a set of sources.
- [Optionally extract shortest paths between pairs of vertices.]

Just one solution is requested, but you are encouraged to experiment with different solutions, as suggested in the following, and compare the results. No template is provided for this assignment.

1. Geodesic graph

Define a function that builds a weighted graph having the vertices of the input mesh as vertices, and arcs corresponding to primal and dual edges of the mesh, as exemplified in Figure 1; the weight attached to each arc is the geodesic distance between its endpoints.

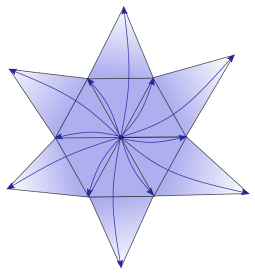

Figure 1. Arcs corresponding to primal and dual edges incident at a vertex.

The graph can be built as follows:

- Add as many nodes as the vertices of the input mesh.
- Cycle over the faces of the input mesh; for each face f cycle on its vertices; for each vertex v of f add arcs joining it to the other two vertices of f, and to the vertex v' belonging to the neighbor f' of f opposite to v and opposite to the edge shared by f and f' (see Figure 2).
- The weight associated to a primal arc is just the length of the corresponding edge. Since triangles f and f' don't lie on the same plane, the weight associated to a dual arc (v, v') can be computed with the following formula:

$$\sqrt{\|vw\|^2 + \|wv'\|^2 - 2\|vw\|\|wv'\|\cos(\alpha+\beta)}.$$

  Compute sine and cosine of angles α and β from the edges of the two triangles independently.
- Remember that the graph is undirected, so both arcs (v, w) and (w, v) must be present for each pair of connected vertices v, w.

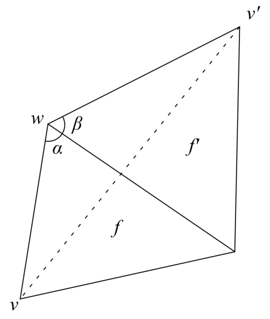

FIGURE 2. The dual edge vv' and angles necessary to compute its length.

There are several alternatives for the data structure to encode the graph:

- **Adjacency lists with plain Python lists:** The data structure consists of an array of lists of pairs of values: the outer array is indexed on the vertices of the mesh (nodes of the graph): the intermediate lists (or arrays) encode the adjacency lists, one for each entry of the outer array, containing one entry per arc incident on the corresponding node; adjacency lists can be built dynamically by appending arcs; the inner structures represent outgoing arcs (single adjacency) and consist of a pair (int, float) where the int encodes the index of the adjacent node and the float encodes the weight of the arc.
- **Using Networkx:** The package Networkx offers primitives for building and inspecting a graph, as well as algorithms, e.g., for computing distance fields and shortest paths.
- **Using SciPy sparse grahs:** A graph can be encoded with a sparse adjacency matrix using package SciPy; the package offers also algorithms working on such representations. See the tutorial. This latter implementation is possibly suited to code optimization using Numba.

Note: this construction algorithm requires cycling over mesh faces and may result slow on large meshes because of the interpreted nature of python. Numba optimization may greatly improve the performance.

Optional extensions: Try more than one implementation and compare them in terms of time performance. Try a solution with the simpler graph containing just primal edges (just remove the lines of code that add the dual arcs).

Relevant igl functions: `triangle_triangle_adjacency`.

Relevant numpy functions: `linalg.norm, dot, sqrt`.

Required output: print the output graph for a very simple mesh (e.g., cube.obj).

2. Geodesic solver

Define a function that takes in input:

- the graph built with the function defined above
- an array of integers, containing the indexes of the vertices of the input mesh to be taken as sources

and returns an array of values, containing for each vertex of the mesh its geodesic distance to the nearest source. Implement the function with the algorithm based on the SLF-LLL heuristics, as seen in class. For multiple sources, the algorithm is identical, except that all sources are added to the queue with zero distance during initialization.

Notes:

- The algorithm needs a double ended queue. This can be implemented in a straightforward way with standard python lists. Alternatively, the package collections offers a deque data structure, which might be more efficient.
- If the graph has been implemented with Networkx or SciPy sparse graphs, off-the-shelf solvers based on Dijkstra search are also available; these solvers can be added as an alternative to the previous (but implementation of the previous remains mandatory).
- Code optimization with Numba decorations can be tried, as in the previous function, for an implementation based on SciPy sparse graphs.

Relevant igl and numpy functions: None

Required output: plot of the mesh colored with the distance field (see Figure 3).

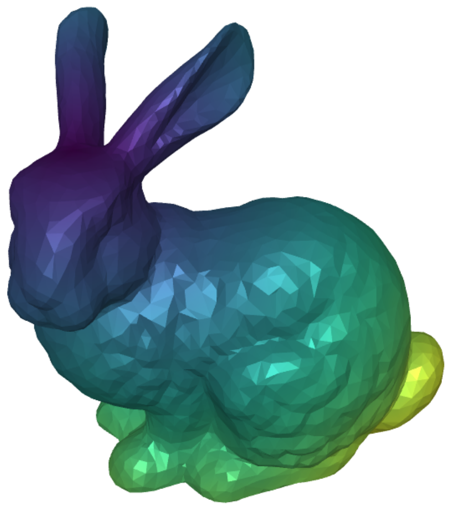

FIGURE 3. A mesh colored with distance field (single source on the forehead, in purple; farthest points on the tail, in yellow).

3. Shortest path between two points [optional]

Define a function that takes in input the mesh, the graph and a pair of its vertices and returns a shortest path connecting them. The path in output shall be encoded as an array of 3D points, which can be drawn on top of the mesh with meshplot. You may try the following algorithms:

(1) **Simple backtrack on the graph:** This solution is straightforward, however, you need to modify the solver above in order to implement it. The solver must return also a second array, which encodes for each vertex its predecessor on the path to the source. When the distance of a vertex is updated as the consequence of a Relax operation, also update its predecessor on this second array; the predecessor of all sources is null and must be initialized in the beginning together with the distance. The backtrack algorithm works as follows. Starting at the target point t, follow its predecessor and add the corresponding arc to the output path; iterate until a source is reached. Note that, while primal edges can be added directly to the output, a dual edge will generate two line segments, one inside face f and the other inside face f' in Figure 2; it is necessary to compute the intersection point between line segment vv' and the edge common to the two new triangles, and its corresponding point in 3D. The path generated with this algorithm is expected to be wiggly. Experiment also with the graph consisting of just primal edges.

(2) **Gradient descent:** the distance field is interpolated linearly inside each triangle of the mesh. Starting at the triangles in the star of the target t, find the triangle with the lowest values of the distance fields at the vertices u, v different from t; compute the gradient $\mathbf{g}$ of the distance field inside this triangle and compute the intersection p of edge uv with a line from t in direction $\mathbf{g}$; iterate from p following the gradient in the adjacent triangle until a triangle with a vertex at a source is reached. See Figure 4 for a visual explanation.

(3) The solver can be modified further by allowing for an early exit when the target t is reached. A straight exit as soon as t is reached provides an approximated result (which is optimal in most cases, though); in order to find the actual shortest path, one can continue the search without expanding nodes whose weight is larger than that of t: in this way, most nodes are just unloaded from the queue without further processing.

(4) Further optimization can be achieved by adding an A* heuristic to the graph search algorithm: the cost associated to a generic node n in the queue is substituted with the sum of its estimated geodesic distance from s (as before) plus its Euclidean distance from t. This strategy prioritizes the expansion of the search in the direction towards t, which is reached earlier.

Relevant igl function: `vertex_triangle_adjacency` (only for the gradient descent method).

Required output: plot the path over the mesh (see use of `add_lines` in igl tutorial - Chapter 1).

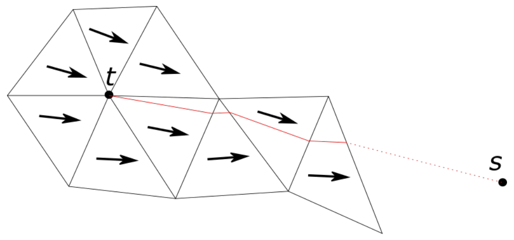

FIGURE 4. Gradient descent from target t to source s. The gradient inside each triangle is obtained by linear interpolation of the distance field at its vertices, as obtained from the solver.