15 - Computational geodesics

In this lecture

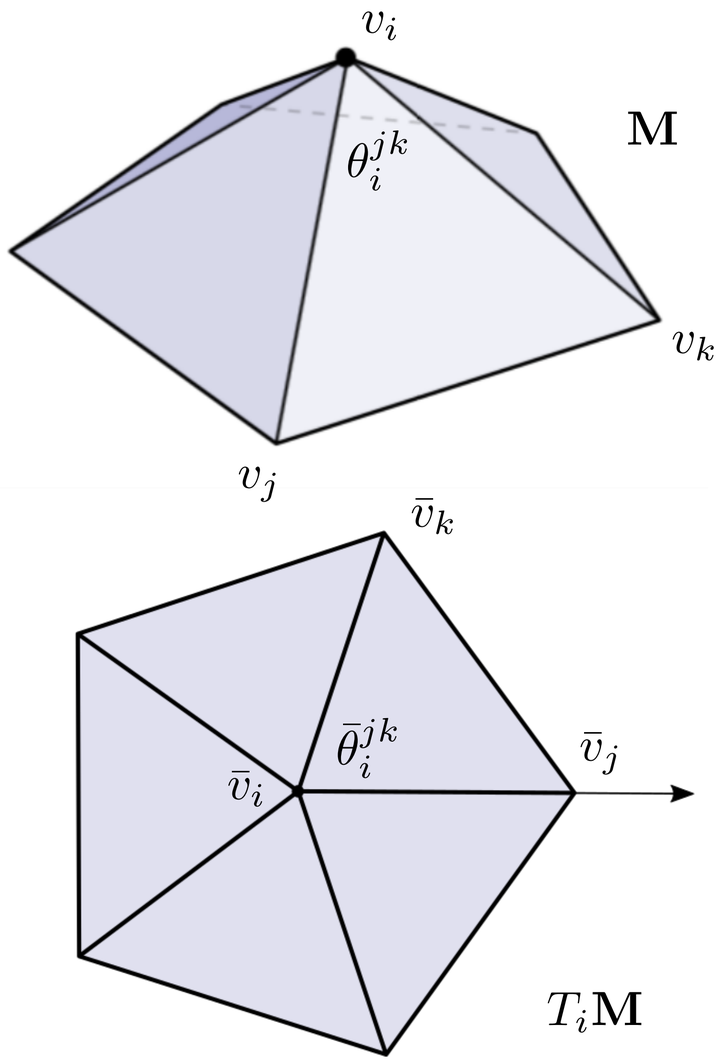
- Algorithms for computational geodesy:
  - on meshes
  - how to trace geodesics
  - how to find shortest paths
  - how to estimate the distance field

Geodesic Tracing

- We assume that the starting point is a vertex v_i (points on edges or inside triangles are easier to handle)
- Tangent direction $\mathbf{t}$ is given on the tangent plane $T_i\mathbf{M}$ at v_i
- Basic steps:
  1. project $\mathbf{t}$ on $\mathbf{M}$ and find the triangle f crossed by such projection
  2. propagate to the triangle adjacent to f by the crossed edge
  3. repeat step 2 until reaching distance $|\mathbf{t}|$

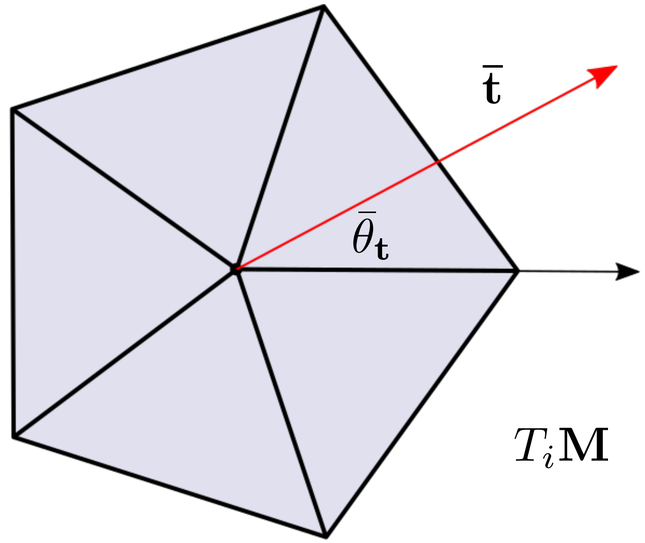
Geodesic Tracing

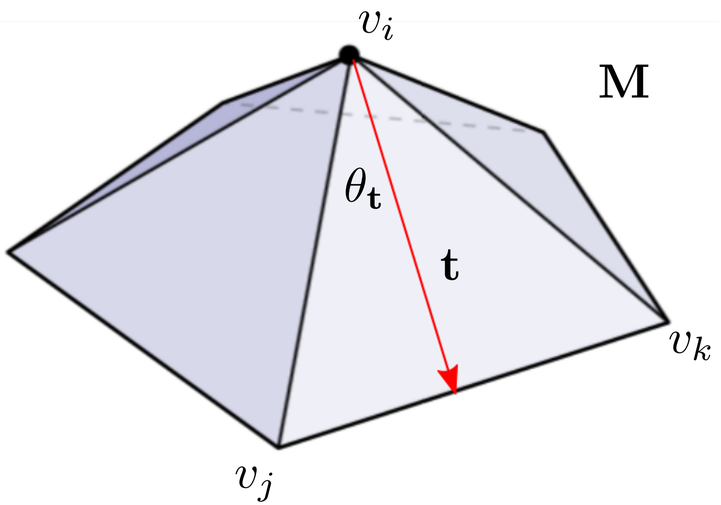
- Mapping the 1-ring to the tangent plane:
  - Compute the total angle $\boldsymbol{\Theta}_i$ about vertex v_i
  - Rescale all incident angles by $2\pi/\boldsymbol{\Theta}_i$
  - Map one edge to the x axis and distribute the others by rescaled angles



Geodesic Tracing

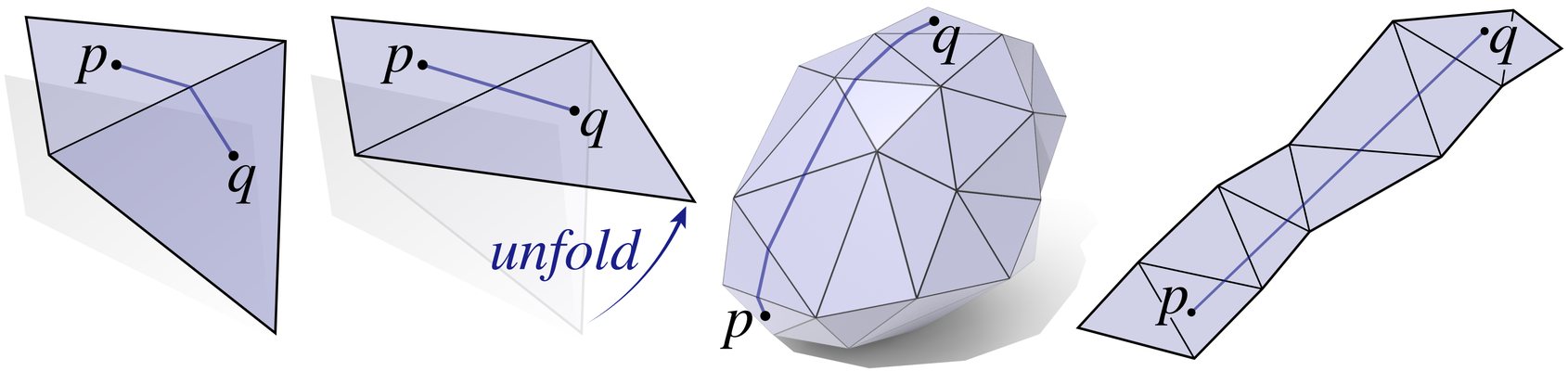
- Mapping the tangent direction to the mesh:
  - Find the triangle in tangent plane containing direction $\bar{\mathbf{t}}$
  - Rescale angle by $\boldsymbol{\Theta}_i/2\pi$
  - Map angle to the corresponding triangle of $\mathbf{M}$





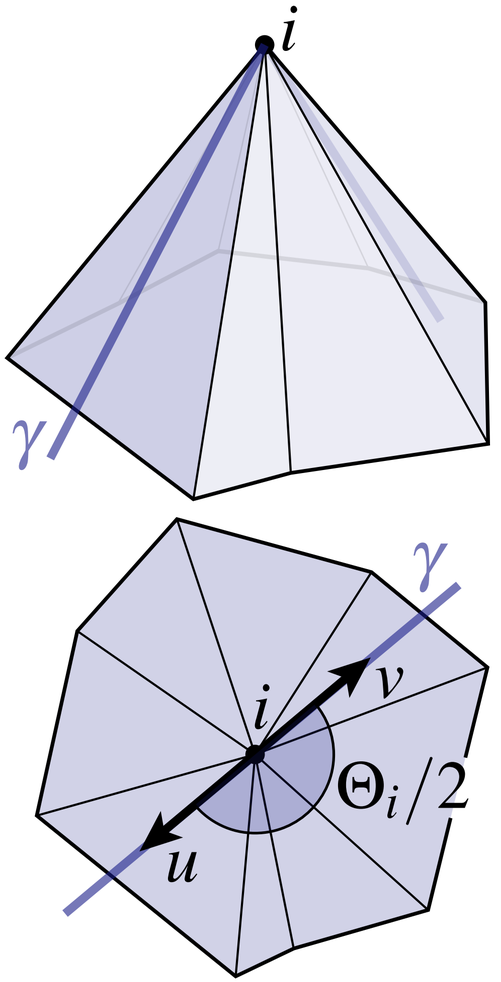
Geodesic Tracing

- Propagating the direction:
  - Find the outgoing edge of the direction in the current triangle
  - Unfold the neighboring triangle on the same plane and propagate a straight line through it
  - Repeated unfolding gives a triangle strip



Geodesic Tracing

- What to do if the geodesic crosses a vertex?
  - *straightest geodesic:* extend with line that halves the total angle [Polthier & Schmies 1998]
  - straightest geodesics do not distribute uniformly at saddle vertices
  - better uniform Geodesic Tracing exploits the fact that a geodesic is a normal curve [Biermann et al. 2002]



Single Source Geodesic Distance

- Three classes of algorithms:
  - Graph-based: restrict possible routes to paths in a graph; compute distance function on the graph
  - PDE-based: resolve a global PDE whose result approximates the distance function
  - Polyhedral wavefront propagation: propagate a front through the mesh while building a data structure that supports computing the exact polyhedral distance from any point

Single Source Geodesic Distance

- Graph-based:
  - Pro: fast, scalable; easy pre-computation; trade-off between accuracy and complexity
  - Con: approximated; fields just at vertices (nodes of graph)
- PDE-based:
  - Pro: fast after pre-computation; approximates distance on the smooth surface
  - Con: approximated; field just at vertices; pre-computation does not scale well
- Polyhedral wavefront propagation:
  - Pro: exact on polyhedra; field at all points; supports SSGP; no pre-computation
  - Con: slow

Graph-based techniques

Shortest path on graphs

- $G=(V,E)$ graph; V set of nodes, E set of arcs
- $w(e)$ weight associated to any edge (for us: geodesic length)
- Weight of a path $p=<v_0,v_1,\ldots,v_k>$ is the sum of weights on the arcs

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Shortest path between u and v:

$$\bar{p}_{uv} = \underset{p}{\operatorname{argmin}}\{w(p) : p \text{ path connecting } u \text{ to } v\}$$

- Length of shortest path: $\delta(u,v) = w(\bar{p}_{uv})$
- Warning: the shortest path is not necessarily unique!

Single source shortest paths

- Assume all weights are non-negative
- Let $s \in V$, it is well defined the *tree of shortest paths* rooted at s

$$G_s = (V_s, E_s)$$

- where
  - V_s is the set of vertices that can be reached from s
  - for each $v \in V_s$ the (only) simple path from s to v in G_s is a shortest path in G

Optimal substructure of shortest paths

- For each shortest path $p=<v_0, v_1, \ldots, v_k>$ and for each pair v_i, v_j with $i<j$ the sub-path $p_{ij}=<v_i, v_{i+1}, \ldots, v_j>$ is a shortest path between v_i and v_j
- Let s be a source and u, v be two nodes connected with an arc in G, then we have

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

- We can find shortest distances by expanding known shortest paths to neighbor nodes
- Since one node can be reached from different neighbors, we select the shortest among possible paths - all candidate paths must be visited

Graph relaxation

- For each node v let us define:
  - $d[v]$ candidate length of shortest path, initialized at +∞
  - $\pi(v)$ node preceding v in a candidate path, initialized empty
- Relaxation of an arc (u,v):

```
Relax(u,v,w)
 if d[v] > d[u]+w(u,v) then
  d[v] = d[u]+w(u,v)
  π(v) = ( π(u), (u,v) )
```

Graph relaxation

- If we repeatedly relax all arcs of E:
  - The value of $d[v]$ is always greater or equal than $\delta(s,v)$
  - If $d[v]$ reaches the value $\delta(s,v)$ then it becomes stable
- If $<s,\ldots,u,v>$ is a shortest path between s and v and we have $d[u]=\delta(s,u)$ then after relaxing edge (u,v) we have $d[v]=\delta(s,v)$

Dijkstra expansion

- Input: graph $G=(V,E)$, weight w, source s
- Output: graph $G=(V,E)$ where for each v we have $d[v]= \delta(s,v)$ and $\pi(v)$ points to his predecessor on the shortest path from s

- Shortest paths can be extracted next in optimal time by backtracking from destination to source following $\pi(v)$

Dijkstra expansion

```
Dijkstra(G,w,s)
 forall v do d[v] = +∞; π(v) = NIL
 d[s] = 0
 //S: vertices with a known shortest path
 S = ∅
 // Q: priority queue, using d as key
 forall v in V do Insert(Q,v)
 while Q ≠ ∅ do
   u = Min(Q)
   S = S ∪{u}
   forall v adjacent to u do
     Relax(u,v,w)
     if relaxed cost is smaller than previous cost then
       Insert(Q,v)
```

Dijkstra expansion

- Correctness: by absurd, assuming there is a node u that is added to S while $d[u] > \delta(s,u)$ … Exercise
- Complexity: $O((|V|+|E|) \log |V|)$ … Exercise
  - for a planar graph, $O(|V| \log |V|)$
  - the $\log |V|$ factor is due to the priority queue

SLF-LLL heuristics

- The priority queue of Dijkstra warrants optimal worst-case time, but it is expensive to maintain in practice
- For planar graphs and graphs used in geodesic algorithms, an algorithm using a double ended queue achieves better practical performances, while being not optimal in the worst case:
  - Small Label First (SLF): a new node is added either to the front or to the back of the queue, depending on its relative cost with respect to the first node in the queue
  - Large Label Last (LLL): nodes from the front of the queue that have a cost higher than the average are moved to the back of the queue

SLF-LLL heuristics

```
SLF-LLL(G,w,s)
 forall v do d[v] = +∞; π(v) = NIL; d[s] = 0;
 add_front(Q,s);
 while Q ≠ ∅ do
   while d[front(Q)] > average cost in Q do
      move front(Q) to the back of Q
   u = pop_front(Q);
   forall v adjacent to u do
     Relax(u,v,w);
     if relaxed cost is smaller than previous cost then
      if d[v]<average cost in Q then add_front(Q,v)
      else add_back(Q,v);
```

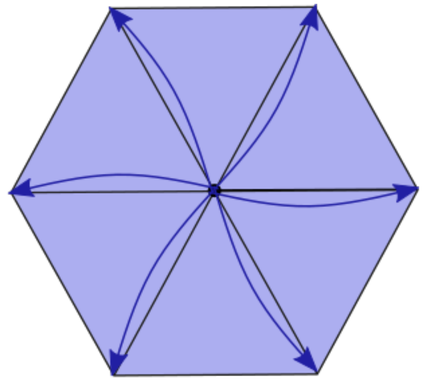
Which graph to use?

- The graph restricts the possible sources to its nodes, and the possible paths to graph paths
- A new source can be added by modifying the graph on-the-fly
- The distance field is defined only at the nodes of the graph
- The more nodes and arcs, the more accurate the result
- Trade-off: find a graph that achieves a good accuracy with a small number of nodes/arcs

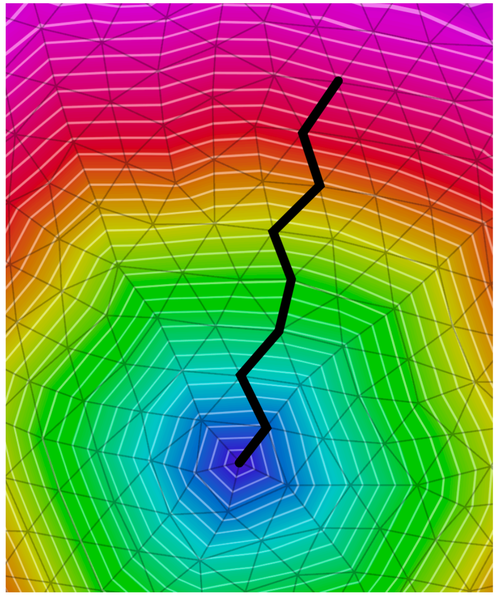
Which graph to use?

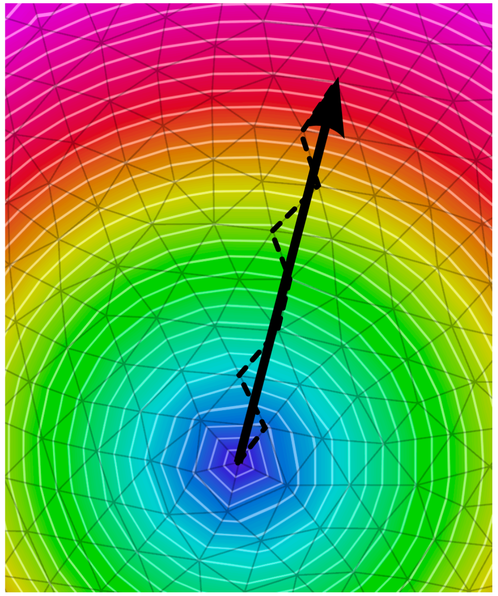
- In most cases, defining the distance field just at the vertices of a mesh is sufficient
- The distance field is interpolated inside triangles
  - linear interpolation is not accurate!
- A graph having just the vertices at nodes is ok
- Edges are more important: wiggly paths are highly inaccurate

Which graph to use?



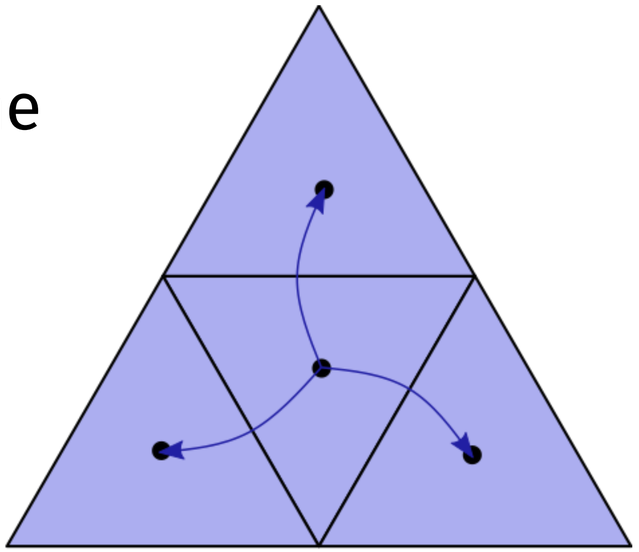
- Graph of edges (primal graph):
  - V vertices of the mesh
  - E edges of the mesh
  - Pro: easy and cheap, does not require any pre-processing
  - Con: wiggly paths, highly inaccurate





Which graph to use?

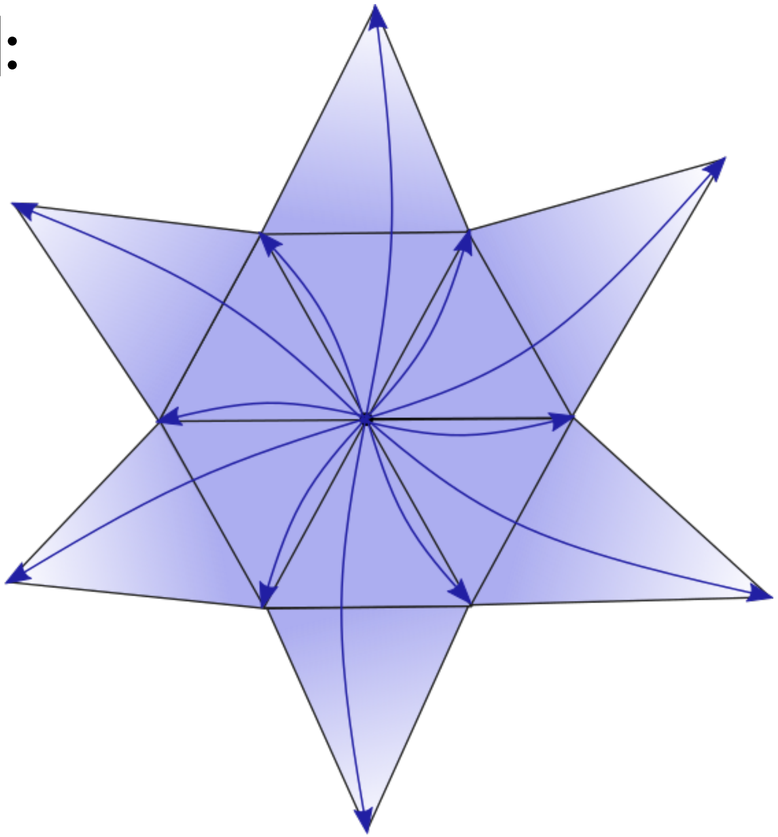
- Graph of adjacencies (dual graph):
  - V (centroids of) triangles of the mesh
  - E dual edges = adjacencies between triangles of the mesh
  - Pro: easy and cheap, does not require any pre-processing, valence three at all nodes
  - Con: wiggly paths, highly inaccurate



  - Often used to find initial guess to be refined further

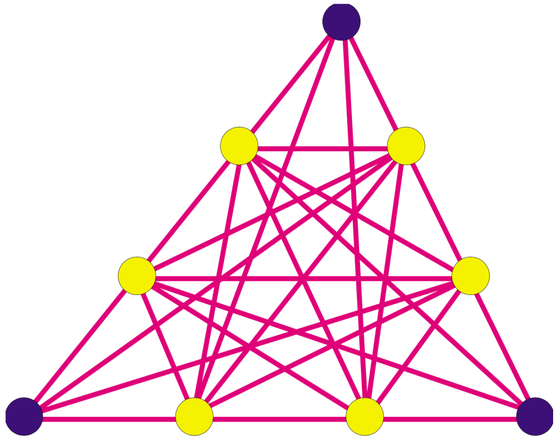
Which graph to use?

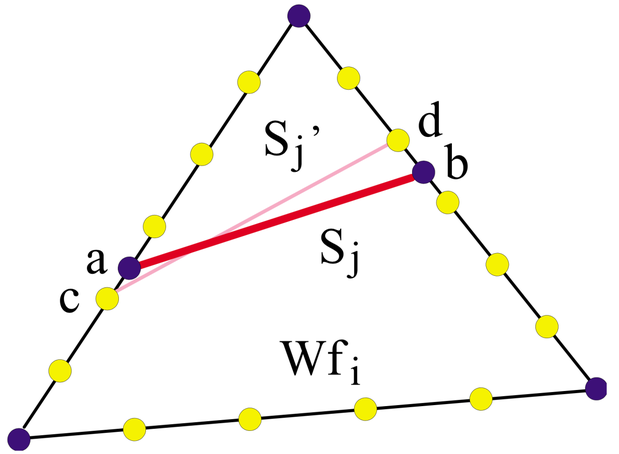
- Graph of primal and dual edges [Nazzaro et al., 2021]:
  - V vertices of the mesh
  - E edges of the mesh and dual edges: each vertex is connected with its neighbors and with the vertices opposite to it in the triangles adjacent to its 1-ring
  - easy and cheap, fast pre-processing
  - much better paths, moderately accurate (error 1-2%)



Which graph to use?

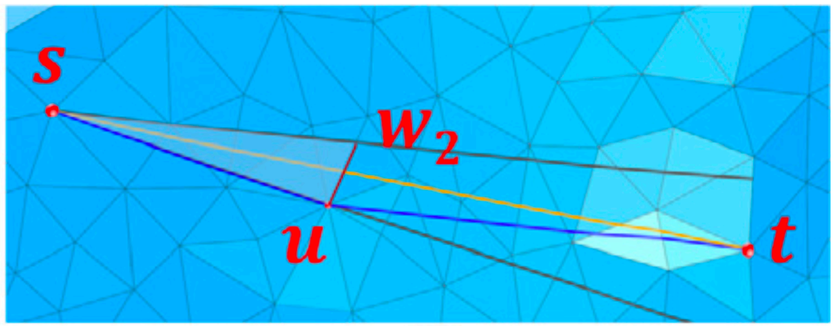
- Graph with Steiner points on edges [Lanthier 1999]:
  - edges of the mesh are split by distributing Steiner points along them
  - within each triangle, arcs joining all mutually visible points on its boundary are added
  - Pro: relatively easy to build, improves accuracy
  - Con: adds extra nodes; complexity explodes with the number of Steiner points per edge; query time slows down

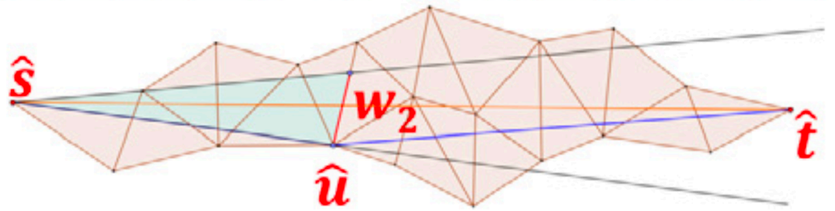




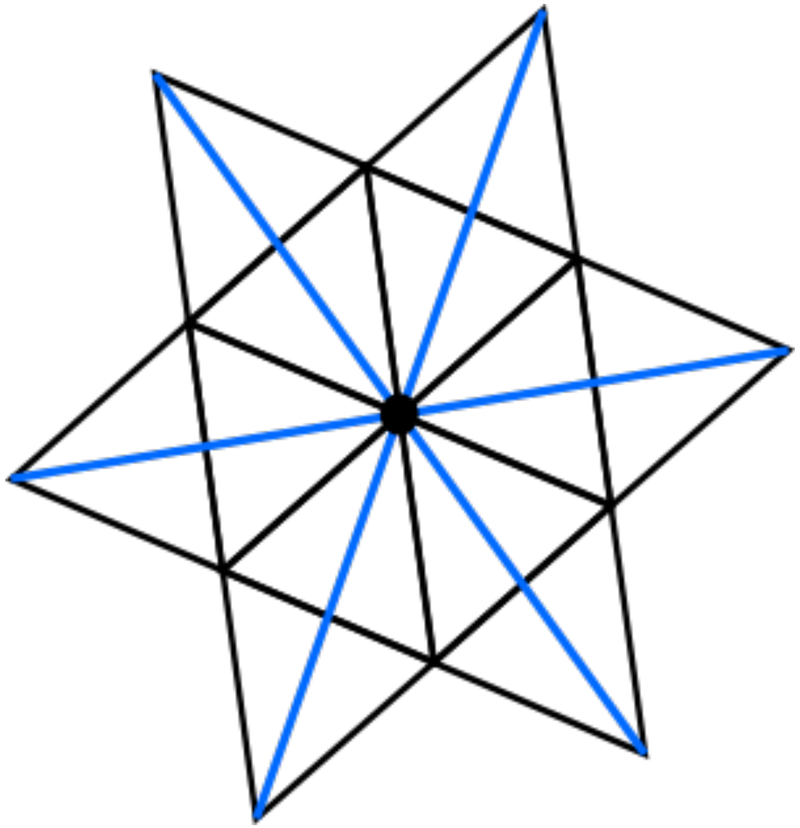
Which graph to use?

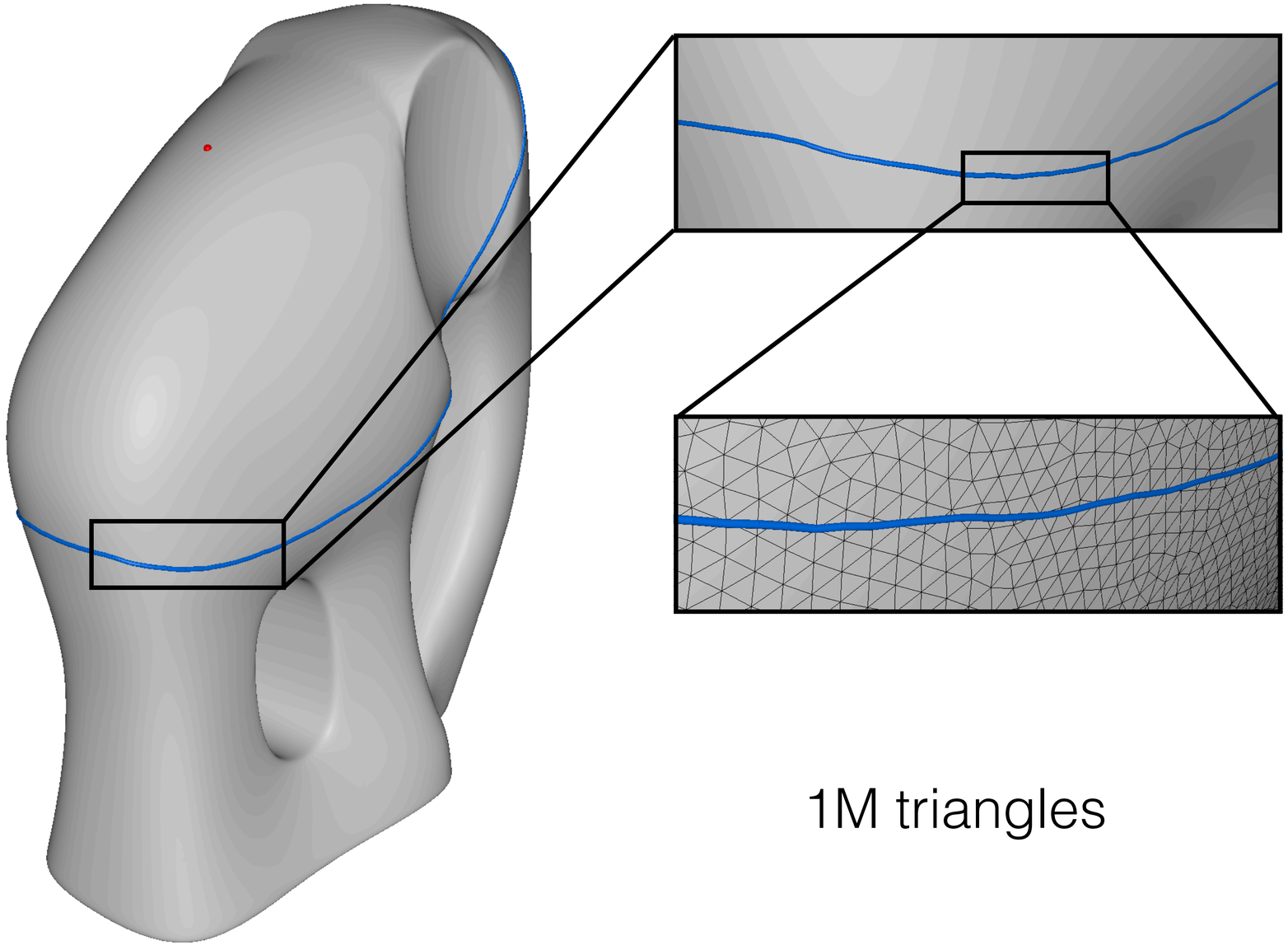
- Discrete Geodesic Graph [Wang et al. 2017, Adikusuma et al. 2020]:
  - V vertices of the mesh
  - E subset of arcs of the total graph
    - each arc is an exact shortest path between its endpoints
    - any exact path can be approximated with a path in the graph within a given tolerance ε
  - Pro: controlled accuracy, nodes are just vertices
  - Con: hard and slow to build, heavy storage (vertex valence can be 10^2-10^3), query time is slower





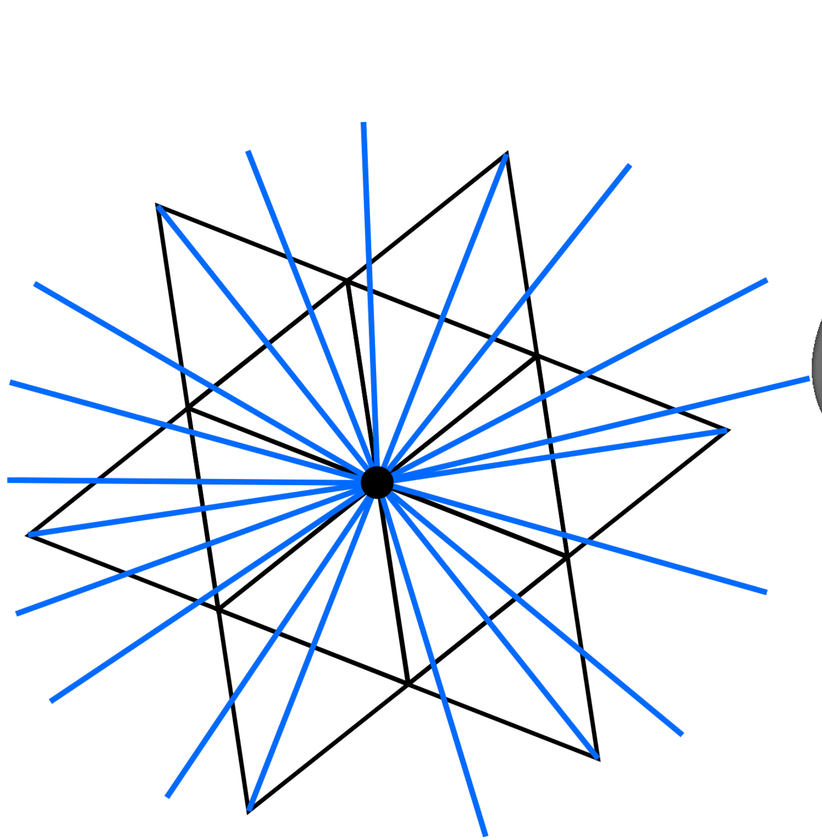
Which graph to use?

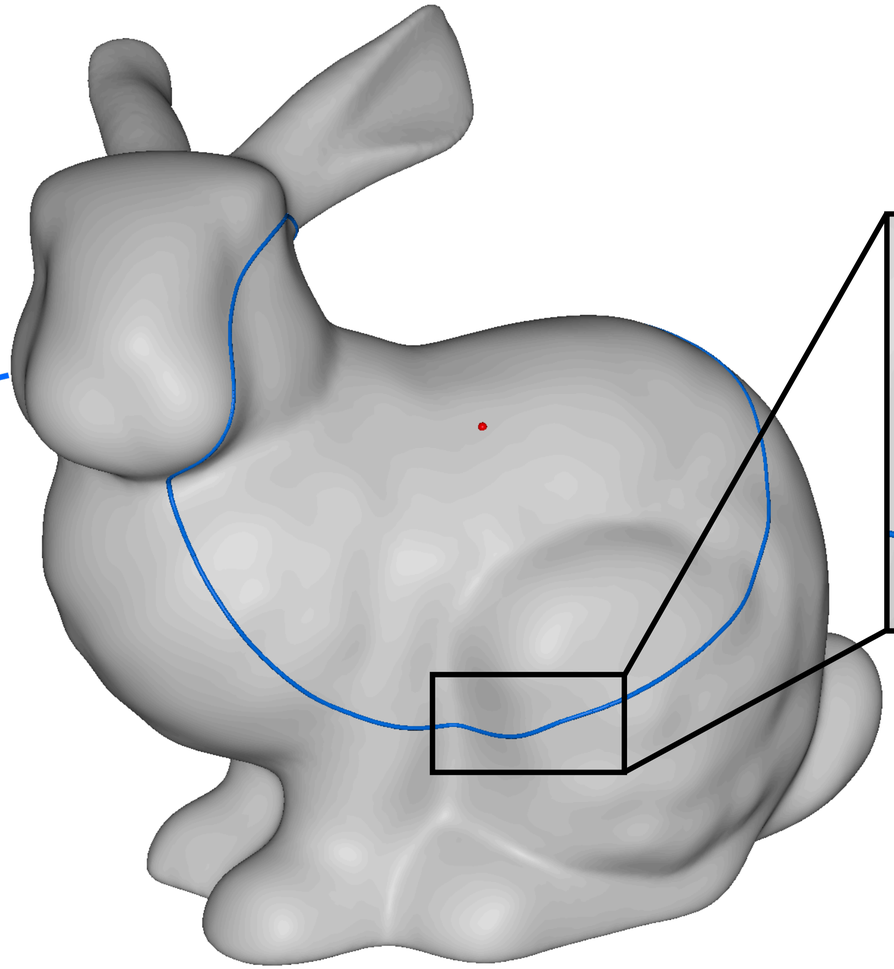


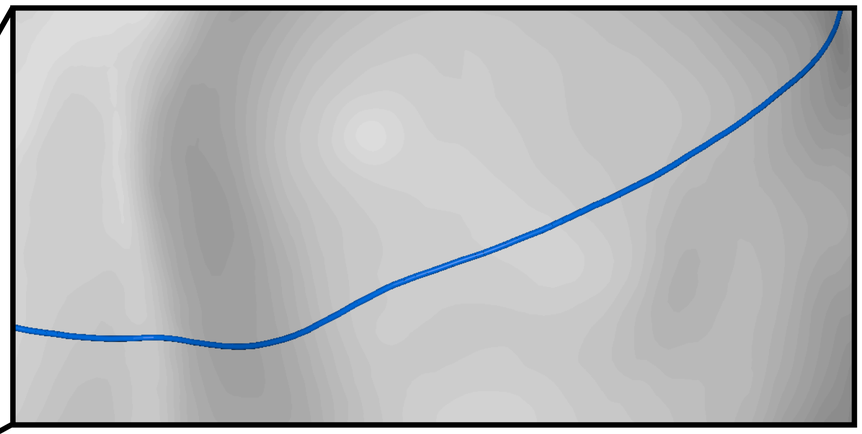


1M triangles

Which graph to use?

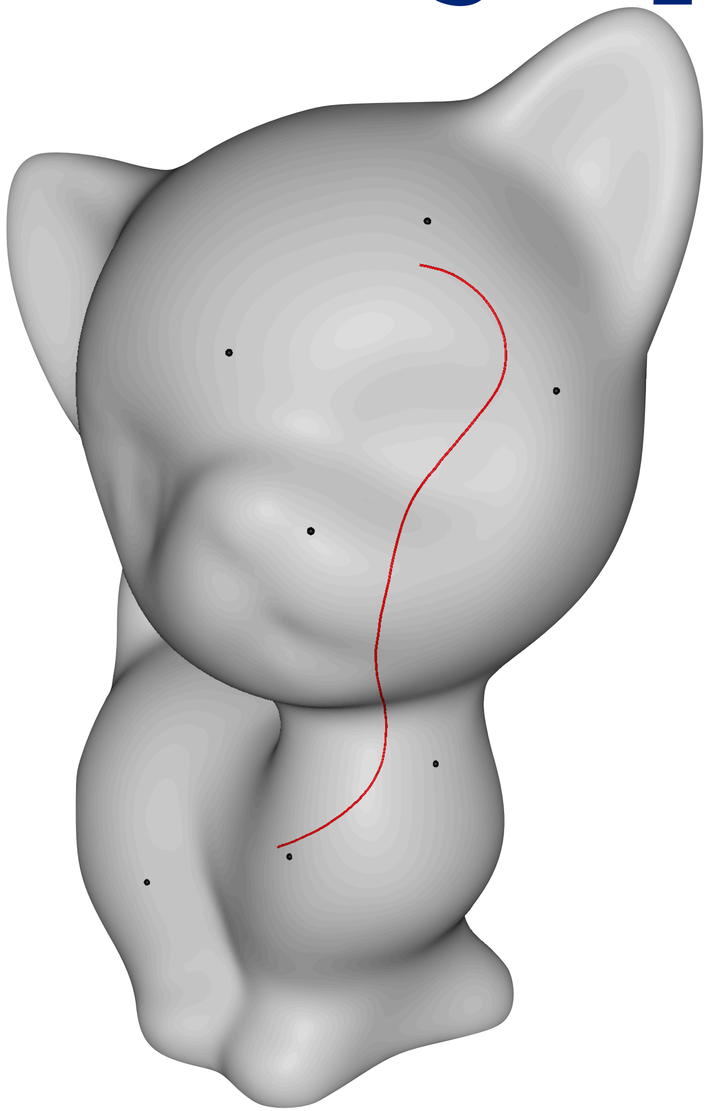




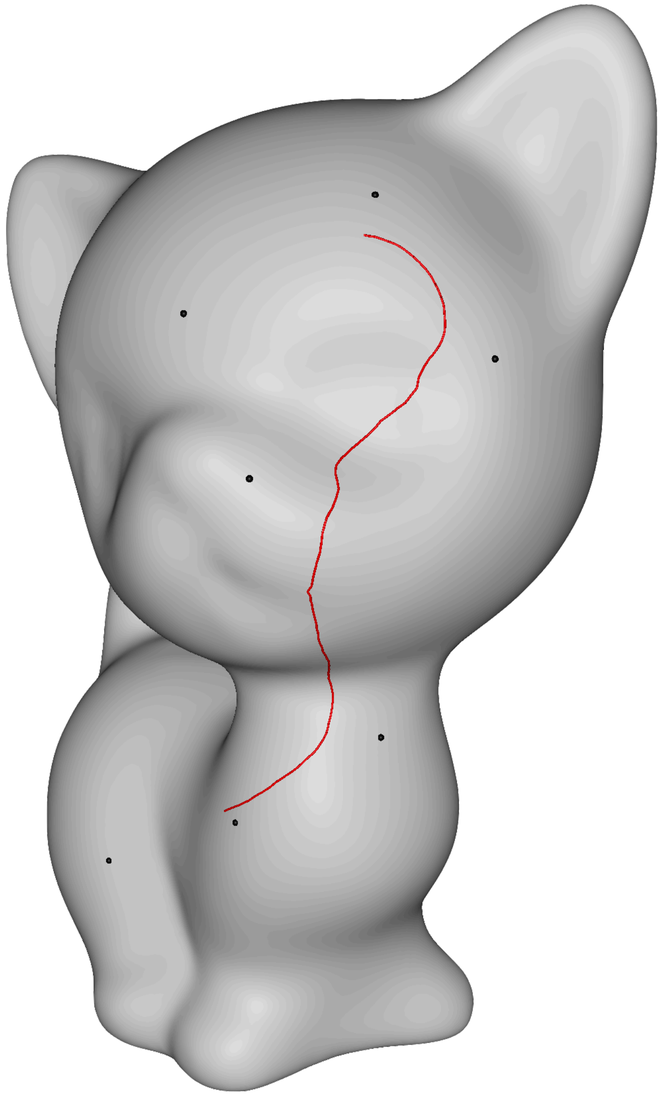


200k triangles
k=4

Which graph to use?



k=6



k=2

Exact polyhedral methods

Continuous Single Source Geodesic Distance/Path

- Input:
  - Mesh $\mathbf{M}$ and source s
- Output:
  - An explicit representation of the geodesic distance function

$$d_s : \mathbf{M} \longrightarrow \mathbb{R}$$

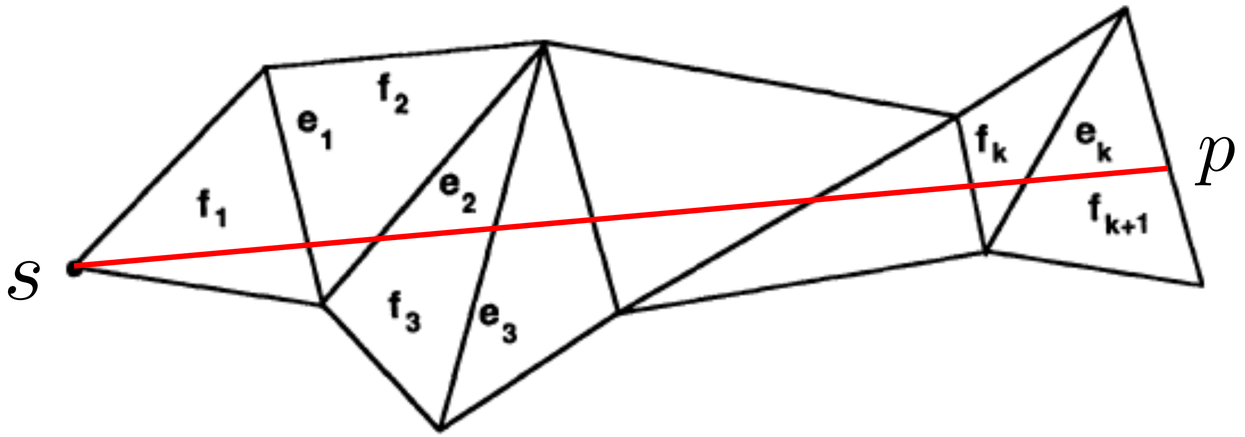
    computable at any point p on $\mathbf{M}$ (not just a vertex)
  - A explicit representation of geodesic shortest paths $\delta(s,p)$

Polygonal wavefront expansion

- Algorithm MMP [Mitchell, Mount, Papadimitriou, 1987] and many subsequent improvements
- Basic idea:
  - expand a front of edges starting at the neighborhood of source s until covering the whole surface
  - encode distance function d_s for points on each edge e as a set of *windows* splitting e

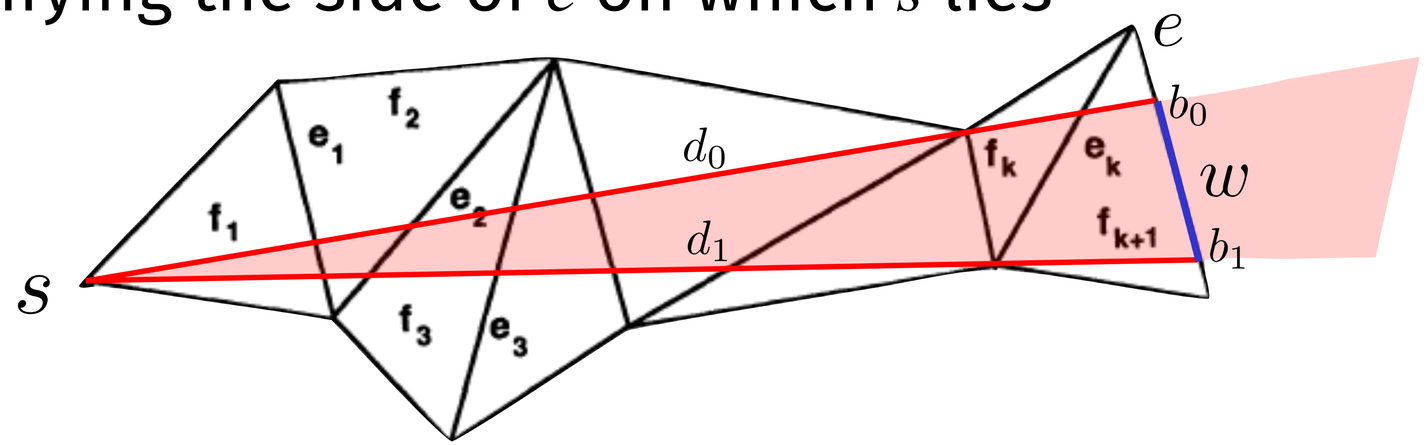
Geodesics through a triangle strip

- Assume s is a vertex (if not, add it as a vertex by splitting the triangle containing it on-the-fly)
- Consider a triangle strip starting at s and flatten it to the plane
- Assume there exists at least one straight line from s contained inside the strip and crossing all its triangles (if not, we are not interested in that strip)
- The folding of such line to $\mathbf{M}$ is a shortest geodesic connecting s to its other endpoint p



Geodesics through a triangle strip

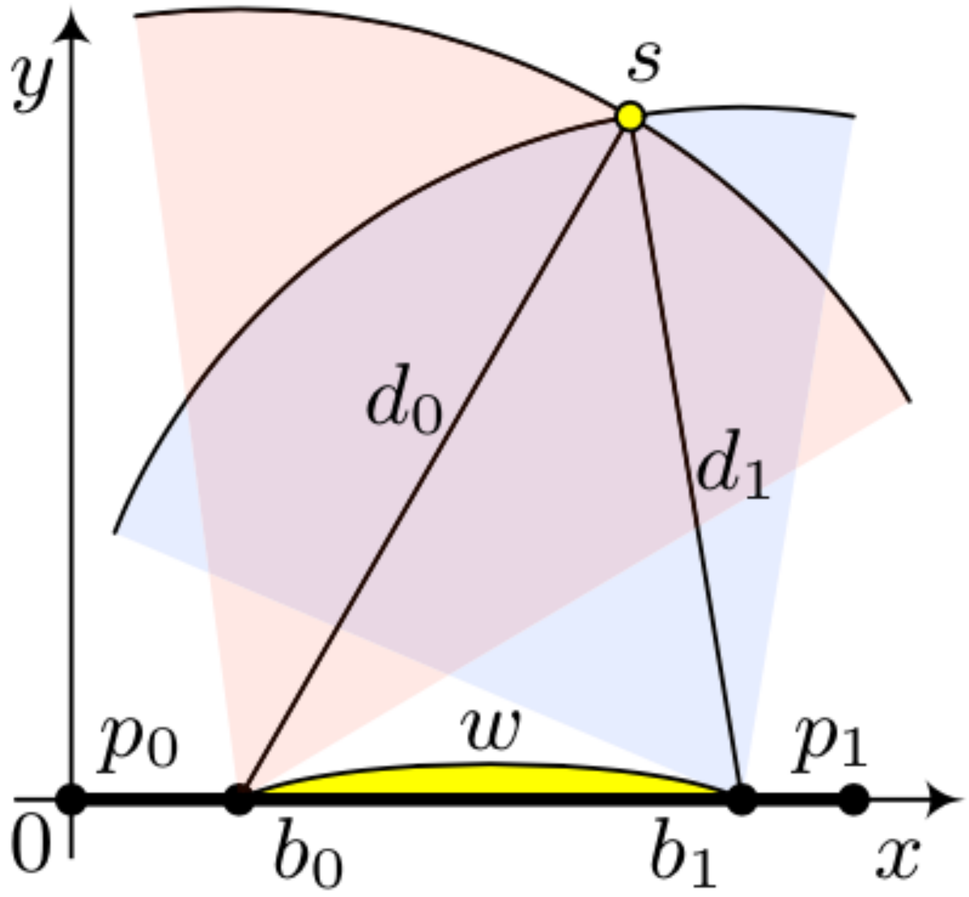
- The set of all lines from s through the strip, and piercing the same edge e on the opposite end of it, form a wedge, encoded as a *window* w on e:
  - parametric coordinates b_0 and b_1 of endpoints of w on e
  - distances d_0 and d_1 of endpoints of w from s
  - binary direction τ specifying the side of e on which s lies



Distance function in a window

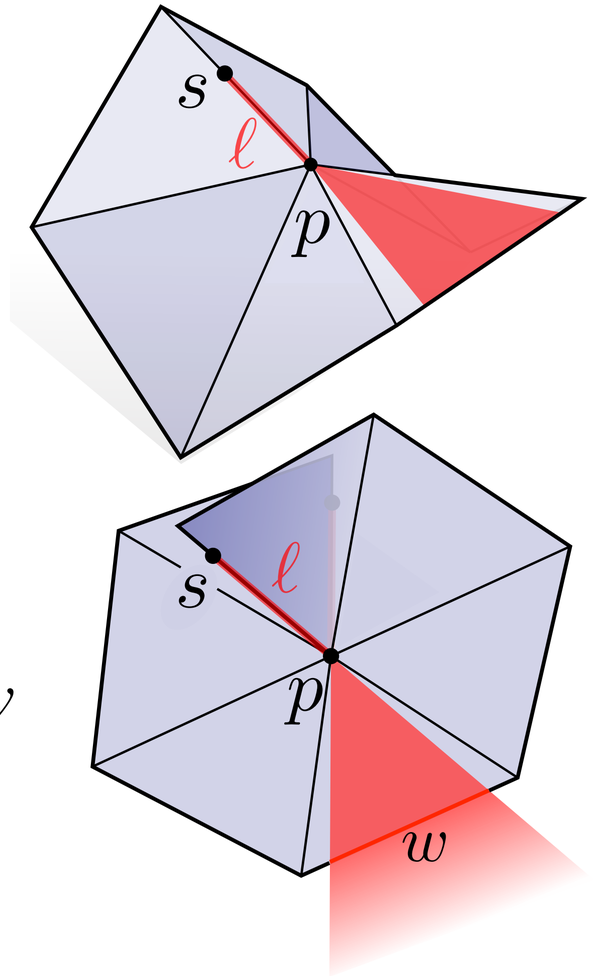
- Coordinates (x_s, y_s) of s are computed by intersecting the two circles centered at endpoints b_0 and b_1:
  - $(x_s\text{-}b_0)^2+y_s^2=d_0^2$
  - $(x_s\text{-}b_1)^2+ys^2=d_1^2$

- $d_s(x)=\text{sqrt}((x_s\text{-}x)^2+y_s^2)$ for $x \in [b0,b1]$



General case

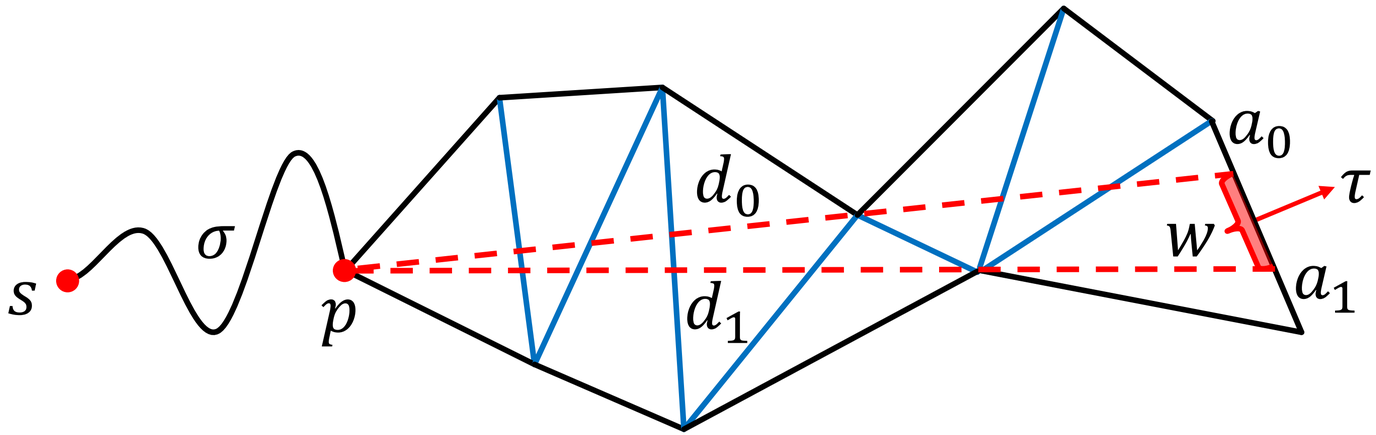
- Not all points can be reached from straight lines through strips
- What happens at saddle vertices?
  - the unfolding of the 1-ring overlaps
  - points in the red wedge cannot be reached by straight lines from s
  - we need to turn about p to reach the red window w
  - line ℓ is a mandatory route to all points of w



General case

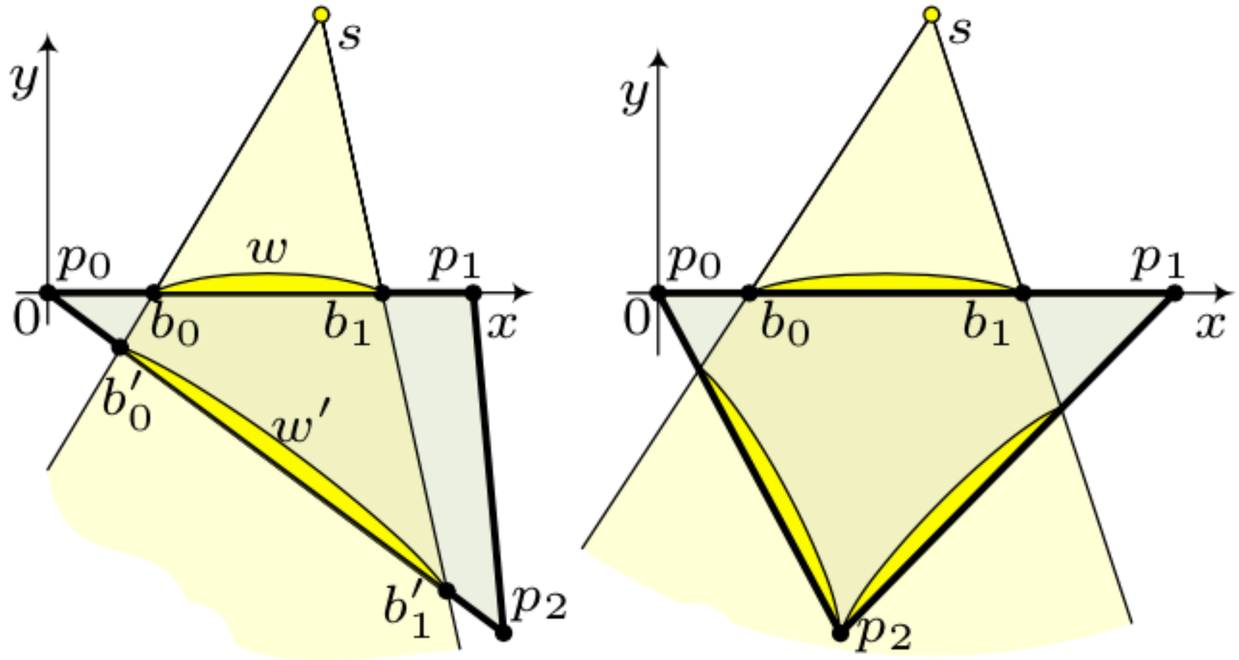
- We decompose the paths to a generic window w into a *funnel* consisting of two parts:
  - a polygonal path of length σ from source s to *pseudo-source* p
  - a wedge from p to w (as before)
- The distance function for all x in w is encoded as:

$$(b_0, b_1, d_0, d_1, \sigma, \tau)$$



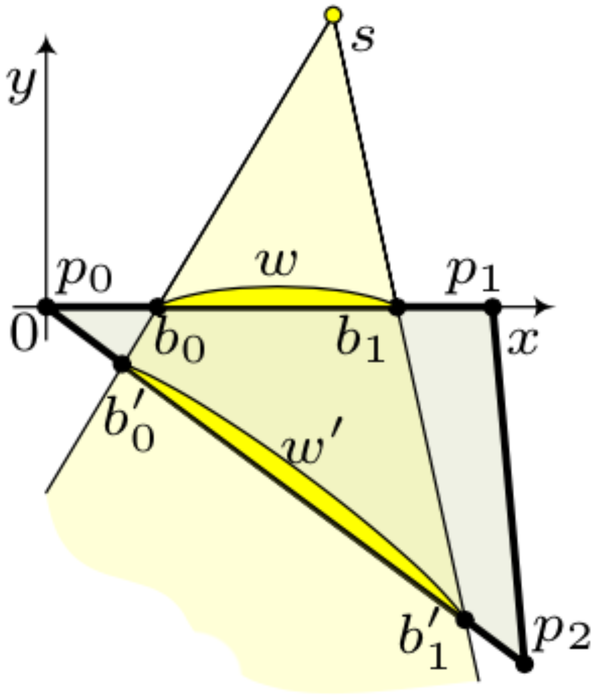
Window propagation

- Given a window w on an edge e_1, propagate its distance field across an adjacent triangle t to define new potential windows on the two opposing edges e_2, e_3



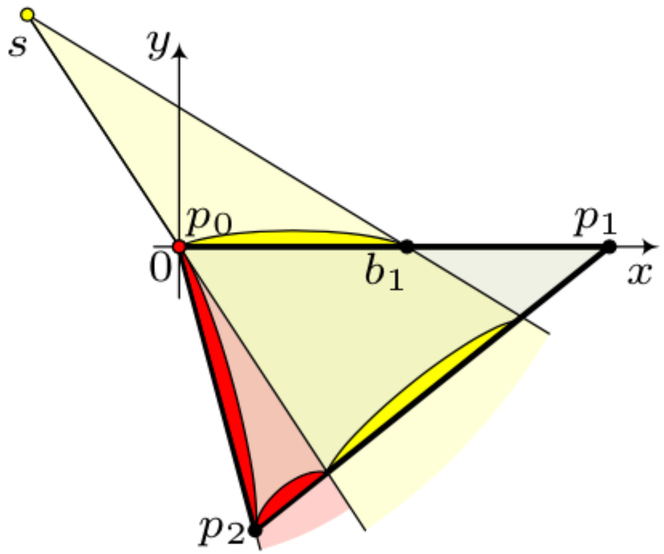
Window propagation

- To define the distance field over w':
- Extend the rays from the pseudo-source s through the endpoints of w and intersect them with the new edge, to obtain the new interval $[b'_0, b'_1]$
- Compute the new distances d'_0, d'_1 from pseudo-source s
- Pseudo-source distance $\sigma' = \sigma$ is unchanged
- Direction τ is assigned to point inside triangle t



Window propagation

- Special case if one of the endpoints of w is a saddle vertex p_0:
  - Consider the other edge $\bar{e}$ of face t incident at p_0
  - If $\bar{e}$ lies outside the funnel, then generate the two new red windows in the figure, having p_0 as pseudo-source
  - Distance σ must be updated $\sigma = \sigma + d_0$



Window propagation

- Each new window w_0 may overlap other windows already computed for the same edge
- Let us suppose window w_1 on edge e overlaps w_0 and let

$$\delta = w_0 \cap w_1 = [b_0, b_1]$$

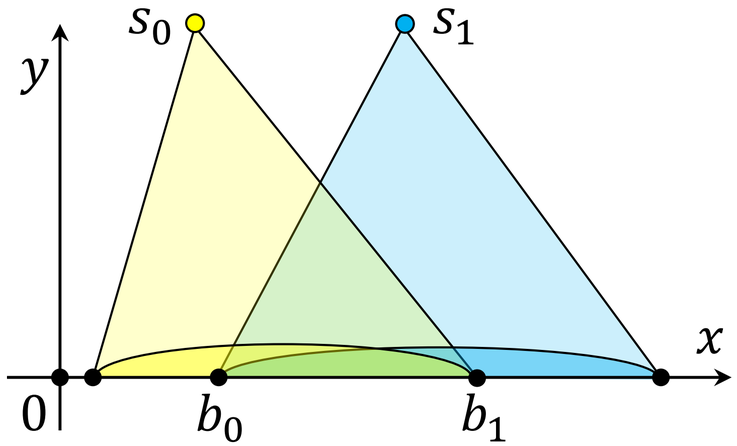
- We must decide which window defines the minimal distance function for each point in δ, and update the windows accordingly

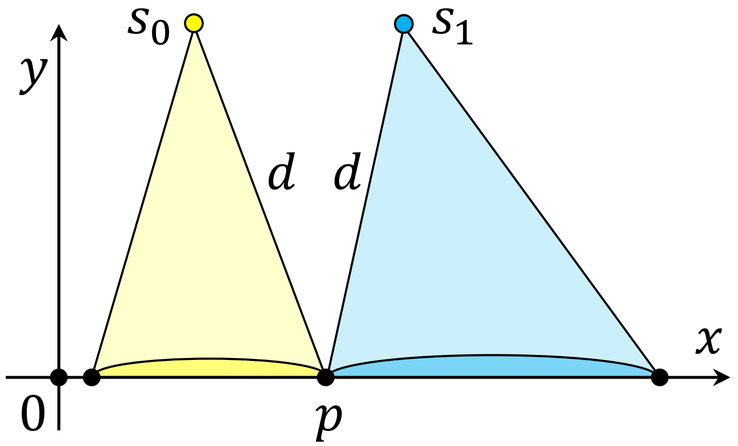
Window propagation

- Find point $p \in [b_0, b_1]$ where the two windows define the same distance

$$\sqrt{(p_x - s_{0x})^2 + s_{0y}^2} + \sigma_0 = \sqrt{(p_x - s_{1x})^2 + s_{1y}^2} + \sigma_1$$

- Solve a quadratic equation to find point p (exercise)





Continuous Dijkstra propagation

- Maintain a priority queue Q of windows
- Priority is minimum distance from source
- Initialization of Q:
  - If s is inside triangle t, then compute one window per edge of t and insert them in the queue
  - If s is on an edge e, then compute one window per edge of the two faces incident at e and insert them in the queue. Also compute two windows on e (no need to insert them in the queue)
  - If s is a vertex, then for every triangle t in the star of s compute one window per edge of t opposite to s and insert them in the queue. Also compute one window for each edge e incident at s (no need to insert them in the queue)

Continuous Dijkstra propagation

- Loop while Q becomes empty:
  - Pop a window w from Q
  - Propagate window w
  - Insert each non-empty window generated by propagation into Q
- Propagation may:
  - Add a new window
  - Modify existing windows
  - Delete existing windows
- In the latter two cases, elements already in Q must be either modified or deleted:
  - Q must support min, member, insert, and delete operations in logarithmic time

Geodesic path construction

- Once all edges are covered by windows representing geodesic distance, it is easy to trace a shortest path from any surface point p back to the source
- For a point p inside a face t:
  - Minimize $|p\text{-}p'|+d_s(p')$ for all p' on the edges of t
  - Jump to the window containing the point that minimizes that distance

Geodesic path construction

- For a point p on a window
  - Find the adjacent triangle t according to direction τ
  - Reconstruct the position of the pseudo-source s in the plane of t and intersect the line to s with the other two edges of t
  - Jump to the intersection point, which is on new window, and repeat
- When reaching a pseudo-source s
  - Explore the windows incident at s until a window with a pseudo-source different from s is found

- Repeat until reaching the source

Time complexity

- n = number of vertices of $\mathbf{M}$
- Each edge may have up to $\mathrm{O}(n)$ windows [Mitchell et al. 1987]
- In the worst case there are $\mathrm{O}(n^2)$ windows
- Each window is processed in $\mathrm{O}(\log n)$ time (access to priority queue and binary search to find overlapping windows)

- Total: $\mathrm{O}(n^2 \log n)$

Geodesic distance between two points

Same algorithm, but we prune the search when a large enough area has been visited

Input: mesh $\mathbf{M}$, points u and v

- Start computing single source geodesic distance from u
- As soon as v is reached by a funnel, compute its distance $d_u(v)$ with respect to that funnel (this is an upper bound to geodesic distance)
- Stop as soon as the window at the top of the queue has a distance larger than $d_u(v)$

PDE-based methods

The Eikonal equation

- The SSGD problem can be formalized as a PDE:

$$|\nabla \phi|^2 = 1 \quad \text{on } \mathbf{M} \backslash \{s\}$$

$$\phi(s) = 0$$

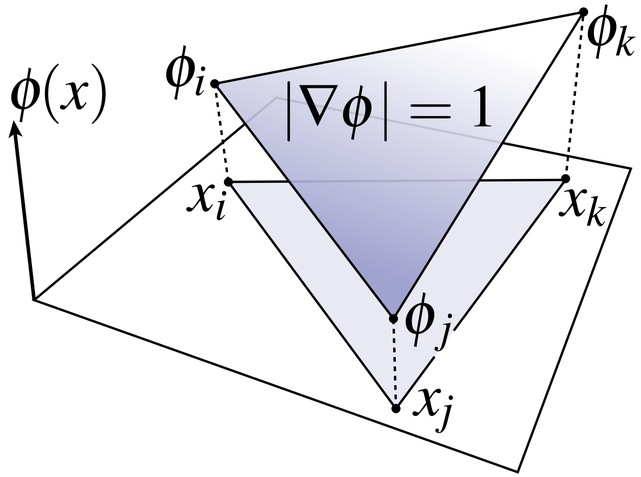
- “the *change in distance per unit distance (i.e., the speed)* along the direction of greatest increase should be 1”
- geodesic lines are integral curves of the gradient of the distance field
- the eikonal equation is a non-linear hyperbolic equation, which cannot be approximated using standard linear finite element methods

.

The fast marching method

[Kimmel and Sethian, 1998]

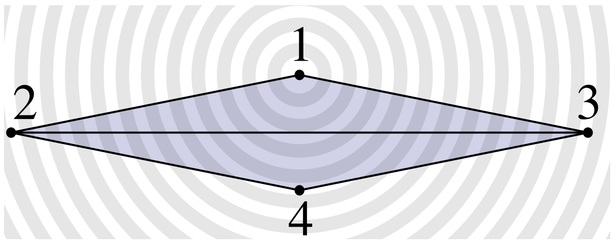
- Follow a Dijkstra-like traversal of the mesh, starting at the source
- Distances are not updated according to paths along edges, but by solving for the linear function that satisfies the eikonal equation
- If the values ϕ_i, ϕ_j at two corners are known, find a value ϕ_k so that the slope $|\nabla\phi|$ of a triangle passing through all three values equals +1

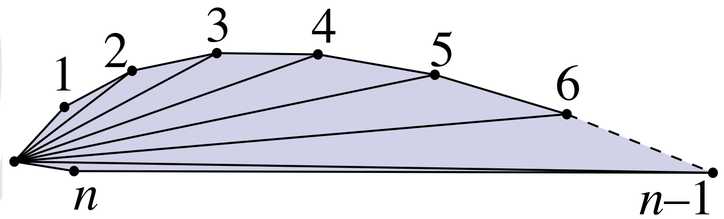


The fast marching method

[Kimmel and Sethian, 1998]

- Linear interpolation of the distance field is inaccurate
  - approximation error increases with the misalignment of the principal direction of the triangle with respect to the (unknown) gradient of ϕ
- The traversal order, given from triangle adjacencies, may fail to analyze closer vertices first, and the problem can be non-local
- Resolving this issue makes the algorithm involved and slow





The Varadhan formula

- The distance field ϕ_x is connected with the *heat kernel* $k_{x,t}$

$$\phi_x(y) = \lim_{t \to 0} \sqrt{-4t \log k_{x,t}(y)}$$

where $k_{x,t}$ is the solution at time t to the heat equation

$$\frac{d}{dt} k_t = \Delta k_t$$

$$k_0 = \delta_x$$

with δ_x a Dirac delta centered at x

- "Start with heat concentrated at x and let it evolve for a short time"

The Varadhan formula

- The heat equation is linear and can be resolved easily
- However, heat decays exponentially with distance, hence the norm of the result becomes soon unreliable:
  - the smaller t the faster the decay
  - the bigger t the worse the approximation of Varadhan formula

The heat method

[Crane, Weischedel and Wardetzky, 2013]

- Compute the heat kernel by resolving

$$\frac{d}{dt}k_t = \Delta k_t, \quad k_0 = \delta_x$$

  for fixed t

- Compute the normalized gradient $X = -\nabla k / |\nabla k|$
- Resolve the Poisson equation $\Delta \phi = \nabla \cdot X$

- We retain just the gradient direction of the heat kernel
- From the eikonal equation, we know that the gradient of the distance function must have norm 1

- We integrate the gradient to obtain the distance field

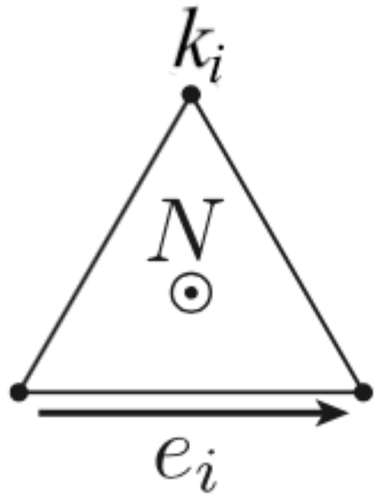
The heat method

[Crane, Weischedel and Wardetzky, 2013]

- To compute the heat kernel, resolve linear system

$$(I - tL)k_t = \delta_x$$

for fixed t, where δ_x is a vector with all zero entries and a 1 at x



- The normalized gradient is computed per triangle

$$\nabla k_t = \frac{1}{2A_t} \sum_i k_i (N \times e_i) \qquad X = -\nabla k_t / |\nabla k_t|$$

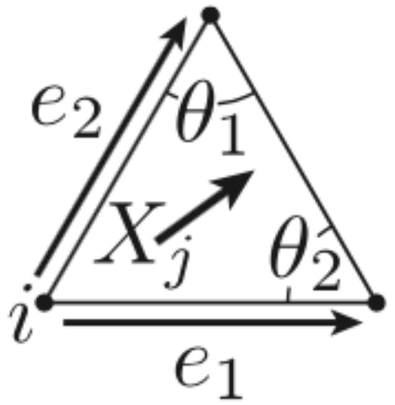
where A_t is the area of the triangle, N is its normal e_i is the edge opposite to vertex i and k_i is the value of the heat kernel at vertex i

The heat method

[Crane, Weischedel and Wardetzky, 2013]

- The divergence associated to vertex i is computed by discrete integration as

$$\nabla \cdot X_i = \frac{1}{2} \sum_j \cot \theta_1 (e_1 \cdot X_j) + \cot \theta_2 (e_2 \cdot X_j)$$



where summation is on triangles t_j incident at vertex i

- The Poisson problem is resolved with the following linear system

$$L_w \phi = b$$

where L_w is the Laplacian stiffness matrix and b is a vector collecting the divergences $\nabla \cdot X_i$ for all i

The heat method

[Crane, Weischedel and Wardetzky, 2013]

Summary:

- We need to solve two linear systems

$$(I - tL)k_t = \delta_x$$

$$L_w \phi = b$$

that are sparse and have the same size of the input mesh (large!)

- They can be pre-factorized once, so that the solution is fast upon changing the source:
  1. solve a $n \times n$ *triangular* system
  2. compute gradients and divergences to build vector b
  3. solve another $n \times n$ *triangular* system

The heat method

[Crane, Weischedel and Wardetzky, 2013]

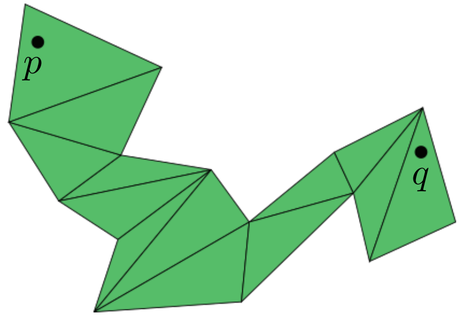
- Pros and cons:
  - Approximated: error ~0.5-1.5% wrt exact polyhedral, but may become large on large meshes
  - Fast after pre-computation: 0.1-0.2 sec on 1M tris
  - Over-smoothed near the cut locus
  - Does not scale well to large meshes
    - Pre-computation becomes very expensive
    - Gradients from heat kernel become too tiny, thus noisy and unreliable even to estimate the gradient direction

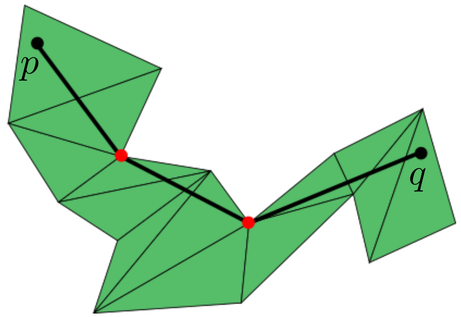
Local methods for PPGP

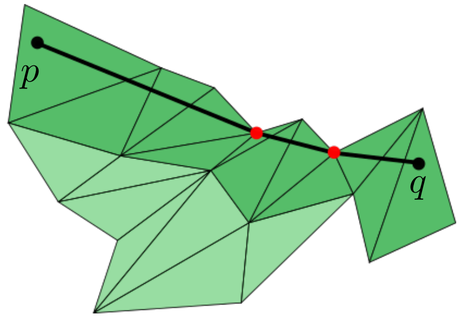
Point to Point Geodesic Path

1. Start with a triangle strip joining p to q (e.g., from shortest path on a graph)
2. Unfold strip to the plane
3. Find shortest path within the strip
4. Straighten the strip to remove turns
5. Repeat 3 and 4 until convergence

- Pro: fast and *scalable*
- Con: finds a *locally shortest* path; step 1 is the bottleneck







Point to Point Geodesic Path

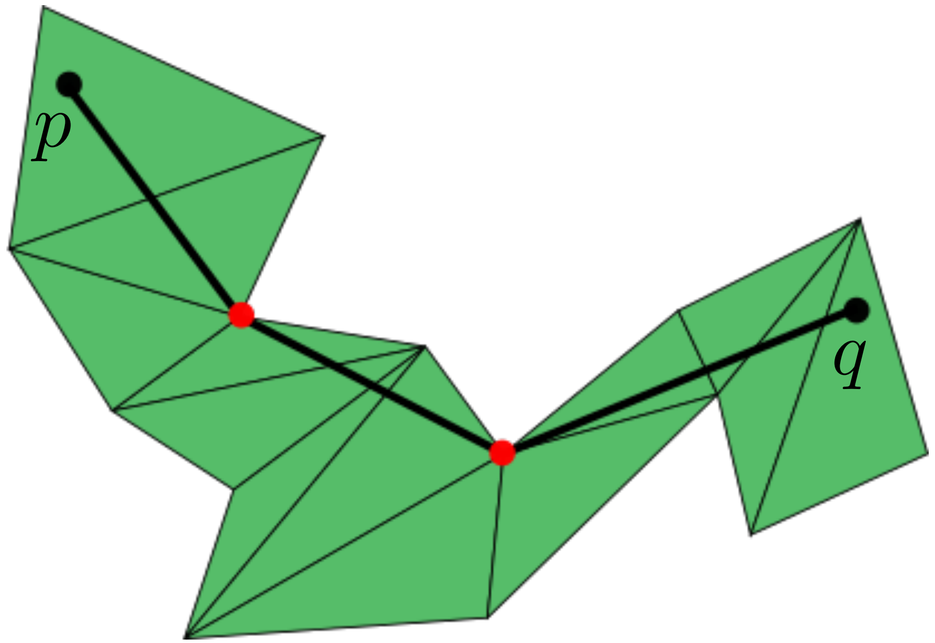
1. Initial triangle strip:
   - usually found with a graph-based method (e.g., from dual graph)
2. Unfolding:
   - one triangle at a time, by using intersecting circles (similar to locating the source in the MMP algorithm)
3. Shortest path in a strip
   - *Funnel algorithm* [Lee and Preparata, 1984]
4. Straightening the strip
   - *Check angle at turns* [Xia and Wang, 2007]

Shortest path in a triangle strip

[Lee and Preparata, 1984]

- Input: triangle strip in the plane, containing p and q in its end triangles, respectively
- Output: shortest polyline connecting p to q inside the triangle strip

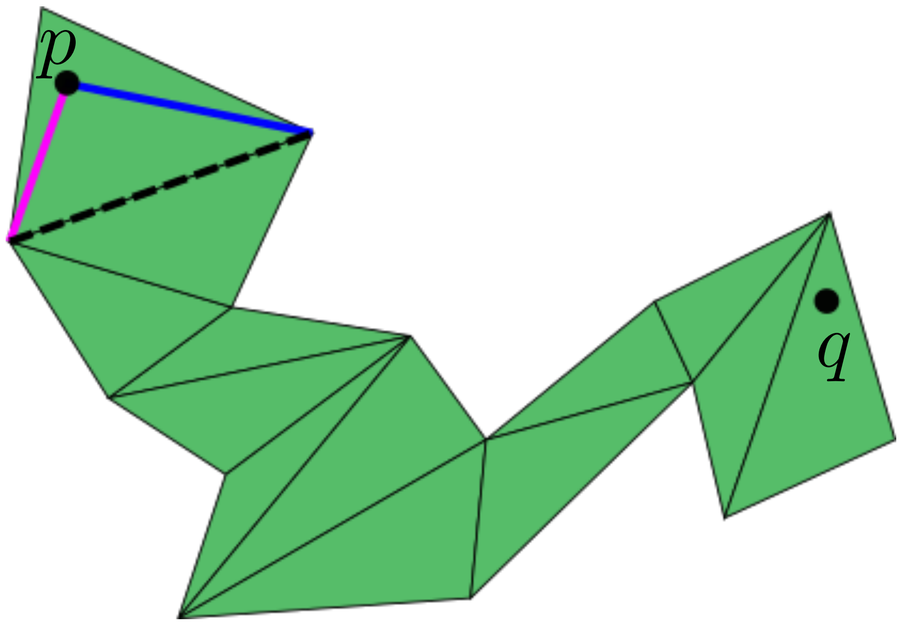
- The path can turn only at reflex vertices



Shortest path in a triangle strip

[Lee and Preparata, 1984]

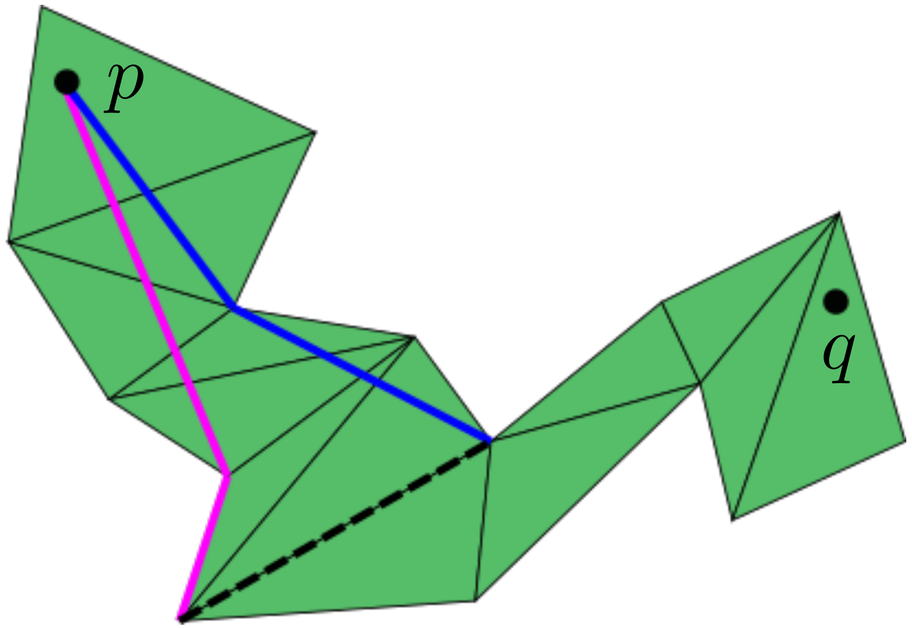
- Initialize funnel: connect p to the endpoints of first transversal edge of the strip; the mouth of the funnel spans all points that can be reached from p



Shortest path in a triangle strip

[Lee and Preparata, 1984]

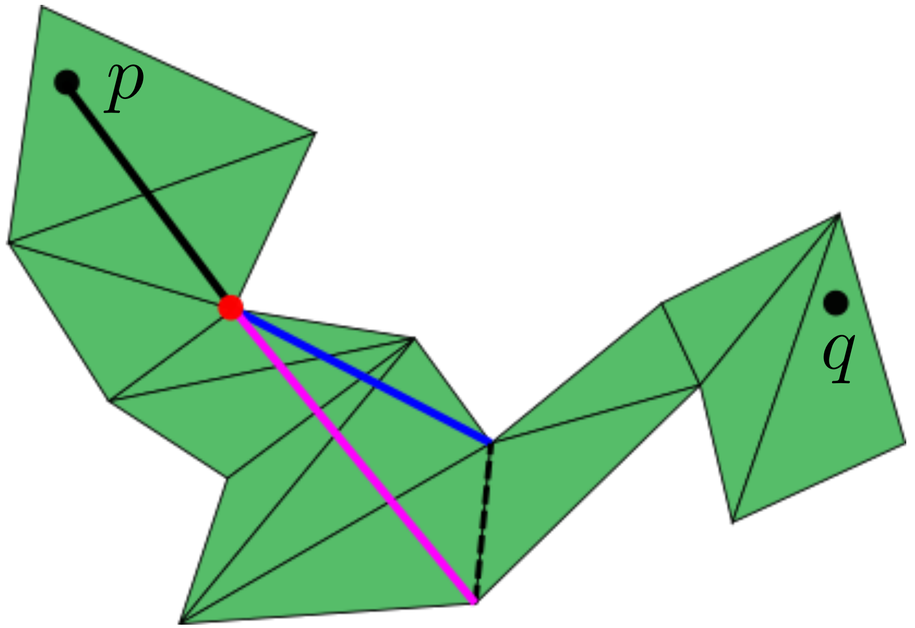
- Expand funnel: move one side of the funnel to the next transversal edge; the sides of the funnel may become concave chains when they overcome reflex edges



Shortest path in a triangle strip

[Lee and Preparata, 1984]

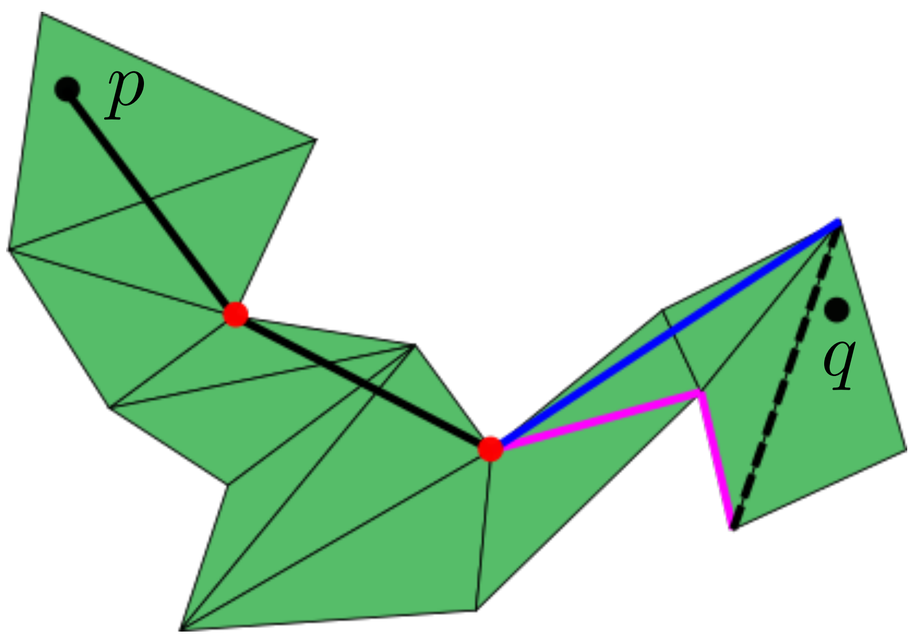
- While advancing, the sides of funnel the may partially collapse to form the beak
- Collapses happen when updating one side overcomes the other side

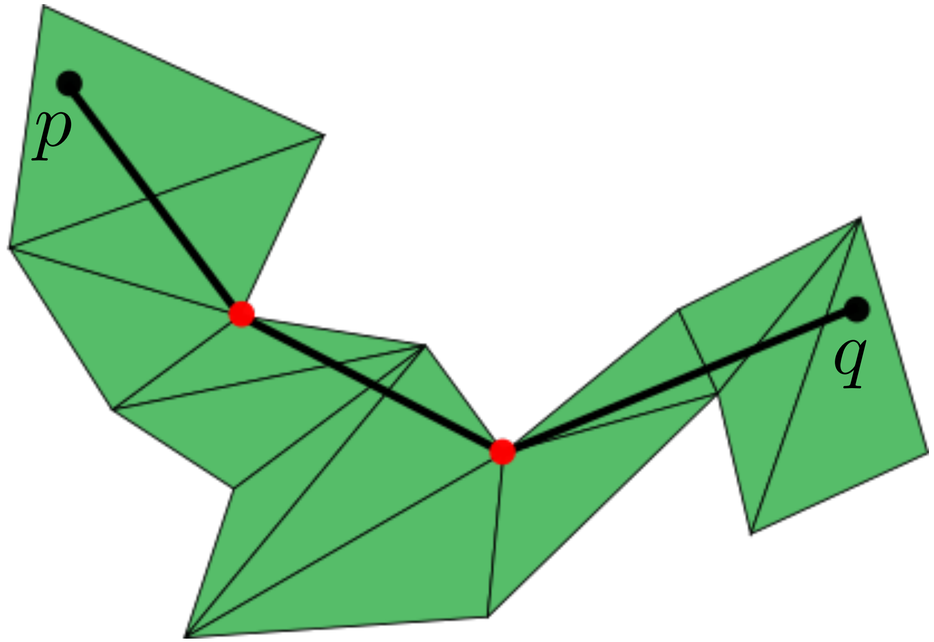


Shortest path in a triangle strip

[Lee and Preparata, 1984]

- The funnel stops advancing when it reaches the last triangle
- The last pseudo-source is connected to q with a shortest polyline running inside the body of the funnel

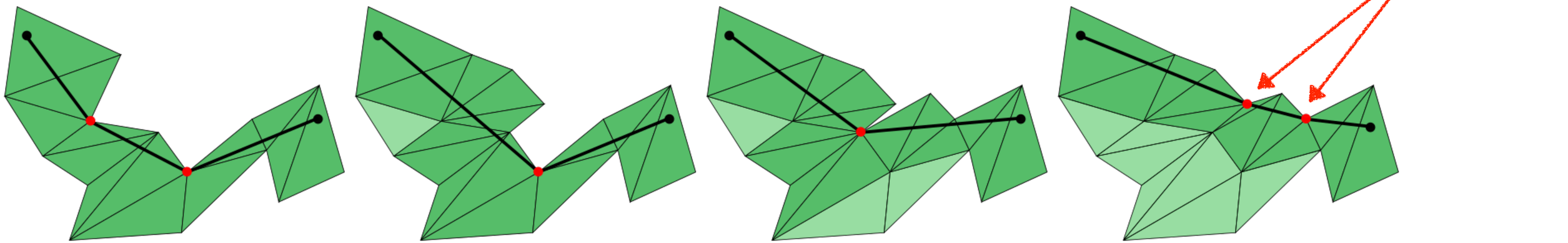




Straightening the strip

[Xia and Wang, 2007]

- A shortest path can bend only at saddle vertices of $\mathbf{M}$
- A turn at a saddle vertex is necessary only if the total angle of the portion of its 1-ring outside the strip is larger than π
- If a bend occurs at v, which does not fulfill the two conditions above, then the path can be shortened by updating the strip with “the other half” of the 1-ring of v



References

A Survey of Algorithms for Geodesic Paths and Distances

Keenan Crane, Marco Livesu, Enrico Puppo, Yipeng Qin

https://arxiv.org/abs/2007.10430

Thank you