Reeb graph

Definition

Given:

- a manifold $\mathcal{M}$;
- a Morse function $f\colon \mathcal{M} \to \mathbb{R}$ with distinct critical values;

the **Reeb graph** of f is the 1-dimensional simplicial complex

$$\mathcal{R}(f) = \mathcal{M}/\sim.$$

$x \sim y$ if they belong to the same connected component of $f^{-1}(f(x))$

The **segmentation map** is the quotient map

$$\Phi\colon \mathcal{M} \longrightarrow \mathcal{R}(f).$$

Reeb graph

Desired algorithm

Input:

- a PL manifold $\mathcal{M}$
  $\rightsquigarrow$ a triangulated mesh $\mathcal{M}$;
- a non-degenerate PL scalar field f on $\mathcal{M}$
  $\rightsquigarrow$ a scalar value $f(v)$ for each vertex v of $\mathcal{M}$.

pairwise different, in order to ensure non-degeneracy; this can be achieved by random perturbations

Output:

- the augmented Reeb graph $\mathcal{R}(f)$.

graph + segmentation map

Time complexity:

- $O(m \cdot \log m)$, where m is the size of the 2-skeleton of $\mathcal{M}$.

#vertices + #edges + #triangles

Parallelizable.

Reeb graph

Geometry of critical points

There are three kinds of critical points:

- (local) **maxima**
  $\rightsquigarrow$ Link^{+} empty;
- (local) **minima**
  $\rightsquigarrow$ Link^{-} empty;
- **saddles**
  $\rightsquigarrow$ Link^{-} or Link^{+} disconnected.

How to detect them on a PL manifold?

Given a vertex v, the **star** of v is the union of all simplices containing v.
The **link** of v is the boundary of its star.

$$\mathrm{Link}^{+}(v) = \{x \in \mathrm{Link}(v) : f(x) > f(v)\}$$

$$\mathrm{Link}^{-}(v) = \{x \in \mathrm{Link}(v) : f(x) < f(v)\}$$

Reeb graph

Significance of critical points

The critical points of f are closely related to the topology of the Reeb graph $\mathcal{R}(f)$.

- **Maxima** and **minima**
  $\rightsquigarrow$ nodes of valence 1 (leaves).
- **Saddles**
  $\rightsquigarrow$ nodes of valence ≥ 2.
  - **Join saddles**: multiple components below.
  - **Split saddles**: multiple components above.

non-mutually exclusive in dimension ≥ 3

Sequential algorithm

Informal description

- Process the vertices of the mesh by **increasing** value of f.
- Construct the Reeb graph $\mathcal{R}(f)$ incrementally.
- While sweeping upwards, keep:
  - the **partial Reeb graph** constructed so far;
  - the current **level set** $f^{-1}(r)$.
- When processing a vertex, **update** the level set and the Reeb graph accordingly.

each connected component
corresponds to an open edge
of the partial Reeb graph

Sequential algorithm

The preimage graph

The level set $f^{-1}(r)$ can be represented by an abstract **graph** G_r:

- **nodes** $\rightsquigarrow$ edges of the mesh $\mathcal{M}$;
- **edges** $\rightsquigarrow$ triangles of $\mathcal{M}$ intersecting $f^{-1}(r)$.

a triangle connects its two sides intersecting $f^{-1}(r)$

Updating G_r

- **Trigger**: update when processing a vertex v

  from $r = f(v) - \epsilon$ to $r = f(v) + \epsilon$
- **Action**: process each triangle $\mathcal{T}$ of $\mathrm{Star}(v)$ separately.
  1. v is the lower vertex of $\mathcal{T}$.
  2. v is the middle vertex of $\mathcal{T}$.
  3. v is the upper vertex of $\mathcal{T}$.
- **Data structure**: the following operations are required;
  - `find` the connected component of a node e;
  - `insert` a new edge between nodes e_1, e_2;
  - `delete` the edge between nodes e_1, e_2;

  $\rightsquigarrow$ offline dynamic connectivity problem $\rightsquigarrow$ **ST-trees**

  support all the operations in $O(\log m)$

Sequential algorithm

The augmented Reeb graph

Reeb graph constructed so far;
has one open edge for each component of G_r

The partial augmented Reeb graph is represented by a pair $(\mathcal{R}, \Phi)$.

partial segmentation map;
maps each vertex of the mesh to a node or an edge of $\mathcal{R}$

Updating $(\mathcal{R}, \Phi)$

When processing a vertex v:

1. **Let** $\mathrm{Lc} = \left\{ G_{f(v)-\epsilon}.\mathtt{find}([vv']) : v' \in \mathrm{Link}^{-}(v) \right\}$.

   lower components
2. **Let** $\mathrm{Uc} = \left\{ G_{f(v)+\epsilon}.\mathtt{find}([vv']) : v' \in \mathrm{Link}^{+}(v) \right\}$.

   upper components
3. **If** $|\mathrm{Lc}| = |\mathrm{Uc}| = 1$ **then**:
   - $\mathcal{R}$ is unchanged;
   - $\Phi(v) =$ the open edge associated to the lower component.
4. **Otherwise**:
   - create a new vertex w in $\mathcal{R}$;
   - all the open edges associated to the lower components end at w;
   - open a new edge in $\mathcal{R}$ starting at w for each upper component.
   - $\Phi(v) = w$.

Sequential algorithm

Full implementation

input : a triangulated mesh $\mathcal{M}$
a scalar field f on $\mathcal{M}$
output: the augmented Reeb graph $(\mathcal{R}, \Phi)$

begin
  $\mathcal{R}$, $\Phi \leftarrow \varnothing$ [graph], $\varnothing$ [function]
  $G \leftarrow \varnothing$ [ST-tree]
  sort the vertices of $\mathcal{M}$ by increasing value of f
  foreach *v vertex of $\mathcal{M}$* **do**
    Lc $\leftarrow$ `GetLowerComponents`(v)
    `UpdatePreimageGraph()`
    Uc $\leftarrow$ `GetUpperComponents`(v)
    if $|$Lc$| = |$Uc$| = 1$ **then** update $\Phi(v)$
    else `UpdateReebGraph`(v, Lc, Uc)
  end
  return $(\mathcal{R}, \Phi)$
end

Parallel algorithm

Core ideas

- **Sequential**: single procedure sweeping all the vertices sequentially.
- **Parallel**: multiple procedures (local growths) running simultaneously.
  - A local growth is started at every minimum.
  - Each local growth spreads independently with an ordered BFS.
  - Each local growth updates its own preimage graph G_r.
  - **Join saddles**: wait until all involved local growths have reached the saddle, then join them.
  - **Split saddles**: the new open edges in $\mathcal{R}(f)$ are handled by the same local growth.

Parallel algorithm

Local growths

Data structures

candidates are sorted by f value

Each local growth keeps:

- a **Fibonacci heap** θ to store candidates for the ordered BFS;
  can be merged in $O(1)$
- an **ST-tree** G_r to store the preimage graph.
  can be merged in $O(1)$

Join saddles

What if a saddle joins components from different local growths?

- **Detection**: before processing a vertex v, check whether all the vertices in $\mathrm{Link}^{-}(v)$ have already been visited.
  concretely, update an atomic counter visitedLower[v] and check whether visitedLower[v] $= |\mathrm{Link}^{-}(v)|$
- **Stopping**: if not, terminate this local growth.
- **Processing**: otherwise, this local growth is in charge of proceeding;
  - join the priority queues (θ) and the preimage graphs (G_r) of all local growths terminated at v;
  - process v as usual.

Parallel algorithm

Local growth implementation

procedure `LocalGrowth`(v_0, $\mathcal{R}$, Φ)
  θ, $G_r \leftarrow \{v_0\}$ [Fibonacci heap], $\varnothing$ [ST-tree]
  while $\theta \neq \varnothing$ **do**   add $\left|\{w \in \mathrm{Link}^-(v) : w \text{ visited by this local growth}\}\right|$
    $v \leftarrow$ vertex in θ with minimal f value
    update visitedLower[v]
    if visitedLower[v] $< |\mathrm{Link}^-(v)|$ **then**
      append (θ, G_r) to pending[v]   critical section
      terminate
    end
    foreach (θ', G_r') $\in$ pending[v] **do**
      θ.`join`(θ'); G_r.`join`(G_r')
    end
    process v, updating G_r, $\mathcal{R}$ and Φ
    add vertices in $\mathrm{Link}^+(v)$ to θ
  end   just as in the sequential algorithm
end